EVALife

Research on Complex Systems

Topics of Evolutionary Computation 2002

– Collection of student reports –

Lecturers: Thiemo Krink, René Thomsen, and Rasmus K. Ursem

EVALife, Dept. of Computer Science, University of Aarhus
Bldg. 540, Ny Munkegade, DK-8000 Aarhus C, Denmark
www.evalife.dk

Table of contents

Offspring creation methods - Comparing different implementations of creating the offspring population

Rasmus Kjær and Anne Joost Jensen

Abstract— **This paper investigates different implementations of creating the offspring population from the parent population. Six different variants were implemented and tested on a set of numerical benchmark problems. The most stable implementation was found to be the simple standard implementation. It doesn't give the best result on all the benchmark problems, but seems to give the best overall result, with the smallest variance on the result between different runs, as well as the fastest convergence.**

1 Introduction

Evolutionary algorithms do not find an exact solution to a problem, but derives an approximated solution. A problem, even for very simple Evolutionary Algorithms, is the great variety of possible implementations. To find the implementation that gives the best approximation to a solution a systematic investigation on the effect of variations in the implementation is needed.

This paper introduces various implementations of how to create the offspring population from the parent population. It also shows which implementation of offspring creation to use to get the best result for each of the numerical problems.

2 Operators

To allow comparison between the different variants, and to be able to draw some general conclusions, all experiments are based on the same algorithm, which is then modified according to the specific variant.

In this section the implementation details of the base algorithm will be introduced. The implementation is based on [2]

2.1 Crossover

The crossover used in the algorithm is numerical arithmetic crossover [4].

2.2 Mutation

The mutation used in the algorithm is Gaussian mutation [4].

2.3 Selection

The selection used in the algorithm is Tournemant selection [4].

3 Variants

This section introduces the different implemented variants of creating the offspring population.

3.1 Variant 1

The first variant is a simple standard implementation. Two random individuals are chosen and the numerical arithmetic crossover is run on the two individuals. After this the new population is mutated and selected. The algorithm for this variant can be sketched as follows:

```
double[] newpop = new Individual[popsize]
for (i=0; i<popsize; i++) {
  if (random()<p\_c) {
    Pick two random individuals I and J.
    newpop[i] =
    RNumArititihmeticCrossover.
    crossover({I,J},{0})
  }
  else {
    newpop[i] = pop[i]
  }
}
pop = newpop;
```

3.2 Variant 2

In the second variant, for each individual in the population crossover is performed (with probability p_c) with a randomly chosen individual of the population. After this the new population is mutated and selected. The algorithm for this variant can be sketched as follows:

```
1.double[] newpop = new Individual[popsize]
2.for (i=0; i<popsize; i++) {
3.  if (random()<p_c) {
4.    Pick one random individual I.
5.    newpop[i] =
      RNumAritithmeticCrossover.
      crossover({pop[i],I},{0})
6.  }
7.  else {
8.    newpop[i] = pop[i]
9.  }
10.}
11.pop = newpop;
```

3.3 Variant 3

The third variant is slightly different because it uses both crossover and mutation in one go. As in the second variant, for each individual in the population, only one random individual is chosen. First crossover is performed on the two individuals. From this crossover we get one offspring, and this offspring is then mutated. The next population is then selected.

The algorithm for this variant can be sketched as follows:

```
1.Initialize and evaluate pop
2.while(not done) {
3.Find elite individual
4.// Crossover and mutation in one go
  1.for (i=0; i<popsize; i++) {
  2.  if (random()<p_c) {
  3.    Pick one random individual I.
  4.    newpop[i] =
        RNumAritithmeticCrossover.
        crossover({pop[i],I},{0})
  5.    Mutate newpop[i]
  6.  }
  7.  else {
  8.    newpop[i] = pop[i]
  9.  }
  10.}
  11.pop = newpop;
5.Select next generation with tournament
  selection.
6.Insert elite individual if it was lost
  in selection.
7.}
```

3.4 Variant 4

In this variant there is no selection, but the offspring replaces the parent if it is better. Each individual from the population is first mutated and replaces the original if it is better. Then crossover is performed on the population, with the offspring replacing the worst parent if it is better. The algorithm for this variant can be sketched as follows:

```
1.Initialize and evaluate pop
2.while(not done) {
3.Mutation where mutated replaces the
  original if it is better.
4.Crossover where offspring replaces
  the worst parent if it is better.
5.}
```

3.5 Variant 5

This variant uses standard crossover where two random individuals are chosen. We mutate individuals, but only keep the resulting individual if it is better than the non-mutated individual. The next population is then selected. The algorithm for this variant can be sketched as follows:

```
1.Initialize and evaluate pop
2.while(not done) {
3.Standard crossover.
4.Mutation where mutated replaces the
  non-mutated if it is better.
5.}
```

3.6 Variant 6

In this variant standard selection and crossover is used, with the child replacing the worst parent, if it is better. Standard mutation is then applied to the resulting population and the next population is selected. The algorithm for this variant can be sketched as follows:

```
1.Initialize and evaluate pop
2.while(not done) {
3.Crossover where offspring replaces
  the worst parent if it is better.
4.Standard mutation.
5.}
```

4 Experiments

To test the different variants we have used the numerical benchmark problems shown in table 2 as the test-suite.

Each of the problems were run with a population size of 100 individuals and the experiments were repeated 50 times.

	Variant 1	Variant 2	Variant 3	Variant 4	Variant 5	Variant 6
0	5.246e-2±8.411e-4	3.096±1.322	3.512±1.213	2.764±0.6593	7.125±1.219	5.275±0.952
1	7.625e-2±1.459e-3	1.86e-2±2.288e-2	5.027e-2±0.203	2.108e-2±2.029e-2	13.212±4.897	1.149±1.109
2	7.375±2.495	7.335±2.517	7.452±2.643	7.979±3.236	11.981±8.120	9.247±3.253
3	27.991±34.649	27.726±35.163	49.437±75.232	38.086±66.388	79981.759±201683.889	146.227±500.230
4	7.049e-3±4.379e-3	8.160±5.782e-3	8.922e-3±5.264e-3	4.872e-15±2.728e-14	5.522e-3±4.86e-3	5.14e-3±4.90e-3
5	2.464e-20±2.869e-20	9.263e-16±1.721e-20	1.735e-16±9.744e-17	7.861e-14±3.128e-14	6.02e-3±9.02e-3	6.734e-5±1.702e-4
6	2.263±3.409e-11	2.262±4.218e-7	2.262±2.600e-7	2.262±2.665e-6	-61.067±14.088	2.262±1.868

Table 1: Average best fitness and variance

Function name
Ackley F1 20D
Griewank F1 20D
Rastrigin F1 20D
Rosenbrock F1 20D
Schaffer F6
De Jong F4 (30D)
Ursem multimodal F8 20D

Table 2: Test suite

4.1 Tuning EA parameters

To determine the optimal settings for each variant on each problem we did a systematic tuning of the crossover and mutation rate. The results can be found in appendix 1.

5 Results

The results of running the six variants on each of the benchmark problems are shown in the graphs, table 3. The graphs are showing the average best fitness for each problem on each of the different variants.

Table 1 is showing the average best fitness and standard deviation at the end of the run for each of the variants on all the benchmark problems.

6 Comparisons

In the graphs, table 3, showing the average best fitness, the first variant can be seen to be the variant that converges faster. The first variant gives the best result on the problems Ackley F1, De Jong F4 and Ursem multimodal F8, although not significantly better than variant 2,3,4 or 6 on the latter problem, but with a significantly smaller standard deviation. The standard deviation for the first variant is actually smaller on all the problems. Variant five stands out as the uncontested worst implementation, as the only variant failing to give good results on the Ursem multimodal problem. The result for variant five on Rosenbrock F1 is also extraordinarily bad. Variant two gives the best results on as many problems as variant one, but with a larger standard deviation and not very good results on some other problems. Variant one gives results close to those of variant two on the problems were variant two is the best. As a final note, without any formal testing done, variant four through six runs significantly faster than the other variants. Variant three is somewhat slower. So if common features between the problems on which variant four gives good results (between three and five of the problems, depending of the definition of good) could be identified, this variant could in some situations be prefered.

7 Conclusions

In this paper we have explored six different variants of creating offsprings from a parent population. The different variants have been compaired by examening the table showing the average best fitness and standard deviation of the variants on all the benchmark problems. The first variant is found to be the most stable variant. It gives the best result for at least three of the benchmark problems, compared to the other variants. The average best fitness of the first variant on the other problems is also close to that of the best variant. Except for the Schaffer F4 problem, where the fourth variant is significantly better than the others.

The variants have also been compared by looking at the graphs off average best fitness per generation for the each of the different variants. It can be concluded that even though the first variant does not always give the best re-

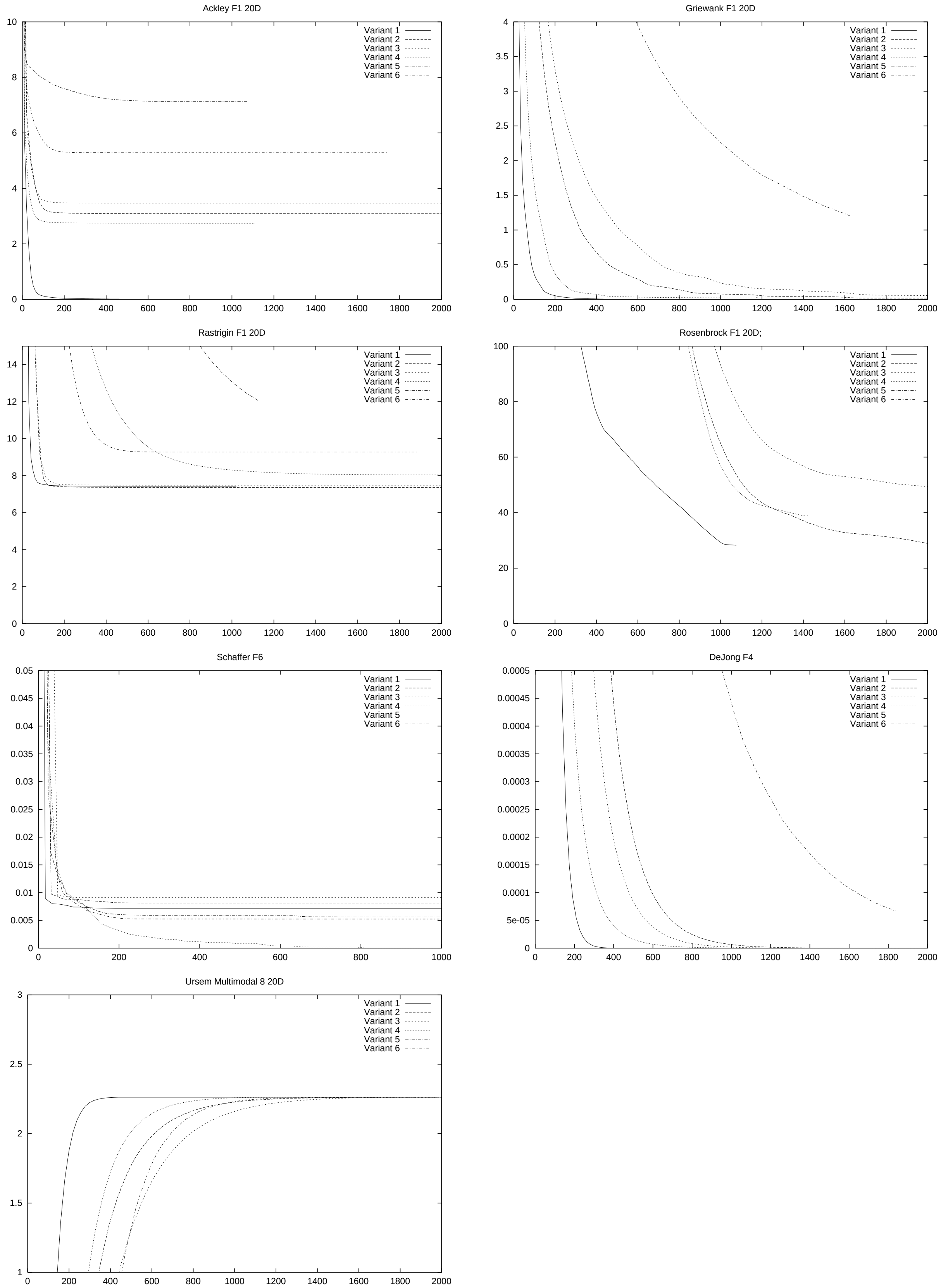

Table 3: graphs of average best fitness.

sult for all the problems, it converges faster than the other variants toward the result.

Taking all things in consideration the first variant of creating an offspring population from a parent population is recommended as a good standard implementation.

References

[1] Thiemo Krink and Rasmus K. Ursem. *Introduction to Evolutionary Computation*, chapter 1. (in prep.), 2003?

[2] Rasmus K. Ursem. Ea-library. *http://www.daimi.au.dk/ ursem/EAdocs/*, 2002.

1 Parameter settings.

Function name	p_m	p_c
Ackley F1 20D	0.7	0.9
Griewank F1 20D	0.6	0.8
Rastrigin F1 20D	0.8	0.9
RosenBrock F1 20D	0.9	0.3
Schaffer F6 20D	0.4	0.3
De Jong F4	0.3	0.4
Ursem Multimodal F8 20D	0.3	0.3

Table 4: Settings for Variant 1

Function name	p_m	p_c
Ackley F1 20D	0.9	0.8
Griewank F1 20D	0.9	0.6
Rastrigin F1 20D	0.9	0.9
RosenBrock F1 20D	0.8	0.8
Schaffer F6 20D	0.4	0.4
De Jong F4	0.7	0.5
Ursem Multimodal F8 20D	0.8	0.9

Table 5: Settings for Variant 2

Function name	p_m	p_c
Ackley F1 20D	0.8	0.9
Griewank F1 20D	0.8	0.9
Rastrigin F1 20D	0.9	0.9
RosenBrock F1 20D	0.8	0.9
Schaffer F6 20D	0.7	0.4
De Jong F4	0.8	0.8
Ursem Multimodal F8 20D	0.9	0.8

Table 6: Settings for Variant 3

Function name	p_m	p_c
Ackley F1 20D	0.9	0.9
Griewank F1 20D	0.9	0.8
Rastrigin F1 20D	0.1	0.7
RosenBrock F1 20D	0.5	0.9
Schaffer F6 20D	0.8	0.1
De Jong F4	0.6	0.5
Ursem Multimodal F8 20D	0.7	0.6

Table 7: Settings for Variant 4

Function name	p_m	p_c
Ackley F1 20D	0.6	0.9
Griewank F1 20D	0.9	0.9
Rastrigin F1 20D	0.9	0.8
RosenBrock F1 20D	0.9	0.9
Schaffer F6 20D	0.7	0.1
De Jong F4	0.9	0.9
Ursem Multimodal F8 20D	0.3	0.4

Table 8: Settings for Variant 5

Function name	p_m	p_c
Ackley F1 20D	0.9	0.9
Griewank F1 20D	0.9	0.9
Rastrigin F1 20D	0.5	0.9
RosenBrock F1 20D	0.9	0.8
Schaffer F6 20D	0.7	0.1
De Jong F4	0.7	0.8
Ursem Multimodal F8 20D	0.3	0.8

Table 9: Settings for Variant 6

Multiobjective optimization

Benjamin Bayart, Poitr Kotlicki, and Michal Nowacki

Abstract—**This report deals with tree differents algorithms used in *Evolutionary Algorthms for Multi-Criterion Optimization.* The goal is to, first describe these algorithms and theirs implementation, and then to compare the results on some differents problems.**

1 Introduction

Many real-world engineering deesign or decision making problems involve simultaneous optimization of multiple objectives. For such issues, the aim is to find a set of compromised solutions - the *Pareto-optimal solutions* -, because there is no single solution due to the conflict between objectives. Thus several algorithms have been proposed so as to deal with such problems as complex test problems or real-world engineering design problems; Then on this paper we focus on tree different algorithms:

- The multi-objective optimizationfor genetic algorithms (MOGA)implemented by Fonesca and Fleming [1].
- The Niched Pareto for genetic algorithms (NPGA) implemented by Horn, Nafploitis, and Goldberg [2].
- The Non-dominated Sorting Genteic Algorithm (NSGA) implemented by Srinivas and Deb [4].

.

2 Multi-objective Algorithm

The principle in the Multi-objective algorithms, due to the fact we have to consider and try to optimize more than one objective, each of them having different individual optimal solution, is to find a set of solutions. As a matter of fact, most of the time we are trying to optimize objectives whcih are in conflict with each other, which means that when one wants to improve the solution for one of the objectives then the values for the other will become worse, and then there is no better individual than other and we will keep this set of individuals, known as the *Pareto front*, as the solution of such problems. Thus the concept of solution for *Multi-Objective optimization* is in fact a set of solutions.

2.1 The Pareto-optimal set

So as to define the *Pareto-optimal set*properly we first need to define the notion of domination for individuals. If we take 2 solutions x_1 and x_2 then they can have one of the two following poosibilities:

- one dominates the other
- none dominates the other

but then a solution, for example x_1 is said to dominate the other solution,x_2, if both the following conditions are true:

1. the solution x_1 is no worse than x_2 in all objectives
2. the solution x_1 is strictly better than x_2 in at least one objective.

Then now we are able to define the **Pareto-optimal set as the set of individuals that are non-dominated in the utter search space**.
Thus the goal of a *multi-criterion optimization algorithm* will be to find that *Pareto* and to maintain the diversity of the population along that *Pareto*.

2.2 Description of such a problem

A general multiobjective optimization problem consists of a number of objectives and is associated with a number of inequality and equality constraints.
Mathematicaly, the problem was written, in [3] as follows:

Minimize/Maximize $f_i(x)$ for i = 1, 2, ..., N
Subject to
$g_j(x) \leq 0$ for j = 1, 2, ..., J
$h_k(x) = 0$ for k = 1, 2, ..., K.

The parameter x is a p dimensional vector having p decision variables.

Solution to this problem is in terms of nondominated or superior points. Namely, we say that the vector x1 dominates vector x2 when vector x1 is partially less then x2. What means that no value of x2 is less then x1 and at least one value of x2 is greater than x1. Vector which is not dominated by any other is defined as nondominated (superior).

That definition, extracted from [5], is used in minimize problems. When the goal is to maximize objectives we find inferior points.
Finally the solution for optimization problem are nondominated vectors, which are known as Pareto-optimal solution
Practically, some solutions are more suitable than others. It depends on environment problem and designer choice. That's our goal is to find entire set of Pareto-optimal solutions.

3 The Multi-Objective GA

In MOGA from the entire population are chosen nondominated individuals and assigned as rank "1". The rest of population is ranked by checking the nondominance of them and comparing with previously chosen nondominated individuals rank. Finally we receive the same rank for two nondominated individuals and rank of dominated individual x is assign as one more than number of all individuals strictly dominate the x. The selection procedure uses these ranks to sort whole population. Next follows an assign fitness by grouping from best to worst rank. Ultimately new fitness counts an average of individuals with the same rank. Then takes place sharing of the objective function values. MOGA uses a niche-formation method to distribute the population over Pareto optimal region. At the end ensues proportional selection.

3.1 Tournament procedure

The tournament is implemented as follows:
ranking_procedure;
sorting_according_to_the_rank;
assigning_fitness;
average_fitness_for_the_same_rank;
sharing_procedure;
proportional_selection;

3.2 Fitness sharing

First we use normalized Euclidean distance:

$$\delta(i,j) = \sqrt{(} \sum_{obj=1}^{N} \frac{x^i_{obj} - x^j_{obj}}{x^u_{obj} - x^l_{obj}})^2$$

, where obj is number of objective,
x^i, x^j are two individuals i and j,
x^u, x^l are upper and lower bound in current front.

This value calculates distance between any two individuals in each front. It's needed in sharing function which presents as follows:

$Sh(\delta(i,j)) =$
$1 - (\delta(i,j)/\sigma_{share})^2$, if $\delta(i,j) < \sigma_{share}$;
0 , otherwise
, where σ_{share} is maximum permissible distance.

Next niche count is calculated by adding the above sharing function values for all individuals.

At least new fitness value equal fitness from ranking procedure divided by niche count.

4 The Niched Pareto GA

The NPGA used a *Pareto domination tournaments.* In this method, a *comparison set* comprising of a specific number (t_{dom})of individuals is picked at random from the population at the begining of each selection process. Two random individuals are picked from the population for selecting a winner in a tournament selection according to the following procedure. Both indiviudals are compared with the members of the comparison set for the domination with respect to objective functions. If one of them is non-dominated and the other is dominated, then the non-dominated point is selected. On the other hand, if both are either non-dominated or dominated, a niche count is found for each individual in the entire population. The niche count is calculated by simply counting the number of points in the population within a certain distance (σ_{share})from an individual. The individual with the least niche count is selected. The effect of multiple objectives is taken into the non-domination calculation. Since this non-dominance is computed by comparing an individual with a randomly chosen population set of size t_{dom}, the success of this algorithm highly depends on the parameter t_{dom}. If a proper size is not chosen, true non-dominated (Pareto-optimal) points may not be found. If a small t_{dom} is chosen, premature convergence may result. Finally one have to find a good value empiricaly. Nevertheless the concept of niche formation among the non-dominated points is an important issue of this algorithm.

4.1 Tournament procedure

The tournament procedure is in fact a tournament selection implemented as follow; Two individuals are chosen randomly trom the population. Moreover one have also to select T_{dom} individuals, chosen also randomly from the population. Then we compare the two first individuals against all the other selected and determine if that two are dominated or not. If one is dominated and not the other one, the latter is selected for reproduction. If neither or both are non-dominated or dominated, we must use sharing - cf next part - to choose a winner.

a simple algorithm of that tournament can be:

function selection
begin
shuffle(rand_pop_index)
individ_1=rand_pop_index[1];
individ_2=rand_pop_index[2];
individ_1_dominated=**false**
individ_2_dominated=**false**
for comp_set_index=3 to 3+t_{dom} **do**
 begin
  comp_individual=rand_pop_index[comp_set_index];
  if S[comp_individual] dominates S[individ_1]
  then individ_1_dominated=**true**
  if S[comp_individual] dominates S[individ_2]
  then individ_2_dominated=**true**
 end
if (individ_1_dominated **end**!individ_2_dominated)
then return individ_2;
else
 if(!individ_1_dominated **and** individ_2_dominated)
then return individ_1
else do sharing
end

4.2 Sharing procedure

So as to maintain a population distributed along the Pareto optimal frontier the fitness sharing procedure is used. It was introduced by Goldberg abd Richardson (1987) and it's aim is to maintain diversity along the Pareto, distributing the population over a number of different peaks in the search space, with each peak receiving a fraction of the population in poportion of the height of that peak. In some other word, one will decrease the *objective fitness value*of an individual depending on how crowded is the neighbourhood of that individual; The more neighbours, the lower new value for the *objective fitness.*

$$f_i' = f_i/m_i$$

where f_i' is the new value and

$$m_i = \sum_{j \in Pop} Sh[d_{i,j}]$$

where $d_{i,j}$is the distance between individuals i and j and Sh[d] is the *Sharing function* such that :

$$Sh[d] = 1 - d/\sigma_{share} \; for \; d \leq \sigma_{share}$$

$$Sh[d] = 0 \; for \; d > \sigma_{share}$$

and σ_{share} is the niche radius, fixed by the user at some estimate of the *minimal separation* desired or expected between the desireable peaks.

Normal fitness sharing

Thus usually after having calculating such a new value for the fitness, we should have 2 new values and then we have to select the individual with the lower fitness, which means that it is the individual with the most efficient fitness value.

Equivalence class sharing

When both individuals compared are non-dominated or dominated and that the two *fitness sharing value* are equal then it is said that two individuals belong to the same *Equivalence Class Sharing*, and then they can be labeled "equally" fit. In such a case we do not apply any degradation according to the niche count on the *objective fitness*, and instead the "best fit" candidate is determined to be that candidate which has the least nuber of individuals in its niche and thus the smallest niche count.
One possible implemetation could be the following:

function selection
begin
.
else if
nichecount[ind_1]$\geqq$ nichecount[ind_2]
then return ind_2;
else ind_1
end

Actually this is done so as to select individuals on the Pareto such that the diversity on it must be the most efficient.

4.3 Sizing niches

The performance of fitness sharing is sensitive to the setting of the niche radius σ_{share} which define finally some kind of Area where to count the neighbours.Thus one can simply define the size of a niche by the dividing of the total area of the frontier,$Area_{pareto}$, by the total number of individuals, such that

$$Area_{pareto}[\sigma_{share}] \simeq Area_{pareto}/N$$

Thus if we are working on a surface of dimansion n-1, we can approximate

$$(\sigma_{share})^{n-1} \simeq Area_{pareto}/N$$

Moreover assuming that we have bounds on our attribute values, we have upper and lower bounds on $Area_{pareto}$ which are the following

$$min(Area_{pareto}) = \sqrt{\sum_{p=1}^{number\_objectives} |a_p^{best} - a_p^{worst}|^2}$$

$$max(Area_{pareto}) < \sum_{p=1}^{number\_objectives} |a_p^{best} - a_p^{worst}|$$

where a_p^{best} and a_p^{worst} are the best and worst values for the search space of the objective p. Thus we can approximate a value for σ_{share}.

5 The Non-dominated Sorting GA

The NSGA is based on an individual's non-domination. The nondominated individuals are first identified from entire population. Then all these individuals are selected to first non-dominated front in that population and assigned a dummy fitness value. An important thing is that all have the same dummy fitness value. After classification they are shared with their fitness value. Sharing is a method of degrading their values. It is obtained by dividing the original fitness value by a number dependent from individuals around them. Next chosen part of population is ignored and the rest of population is identified for the second non-dominated front in the same way. The process is continued until all individuals are selected into a few fronts. Then the whole population is reproduced depending on its dummy fitness values using proportional selection.

5.1 Tournament procedure

The tournament procedure is implemented as follows:

```
current_front = 1;
while (population == classified) {
creating_pareto_front; /* identifying nondominated individuals */
assigning_dummy_fitness;
sharing_in_current_front;
current_front += 1;
}
proportional_selection; /* reproducing according to dummy fitness */
```

5.2 Fitness sharing

First we use normalized Euclidean distance:

$$\delta(i,j) = \sqrt{(}\sum_{obj=1}^{N} \frac{x^i_{obj} - x^j_{obj}}{x^u_{obj} - x^l_{obj}})^2$$

, where obj is number of objective,
x^i, x^j are two individuals i and j,
x^u, x^l are upper and lower bound in current front.

This value calculates distance between any two individuals in each front. It's needed in sharing function which presents as follows:

$Sh(\delta(i,j)) =$
$1 - (\delta(i,j)/\sigma_{share})^2,$ if $\delta(i,j) < \sigma_{share}$;
0 , otherwise
, where σ_{share} is maximum permissible distance.

Next niche count is calculated by adding the above sharing function values for all individuals in current front. At least new fitness value equal group fitness in present front divided by niche count. Group fitness is defined for the first front as some constant value, but next value is minimum of previous one minus constant, positive epsilon.

6 The problems

6.1 The implementation

So as to compare efficiently the 3 differents algorithms, we have used the same functions for the cross-over operation and for the mutation one also. Thus, and due to the algorithm, the only operation which differs between the implemantations of the algorithms is the selection operation.

The code

All the algorithm have been implemented in C. We have chosen to use it because, first of all it was the langage with all know the best and which was the best adapted to do a lot of calculation. Thus 4 principal files was implemented :

- *main.c* file containing the main function and where all the other functions - initialization, selection, cross-over and mutation- are called, used in the main loop for the generation.
- *pop_lib.c* file containing the functions, shared by the 3 algorithms -initialization, cross-over and mutation- and one another function fonction so as to copy the result in a file.
- *problem.h* file containing the description of the 3 problems used for the tests, where the functions, the constraints and the dimension of the problem are described.

Moreover we have also created 3 other files, each of them containing the specific functions for each of the 3 different functions for the selection coming from the 3 algorithms.

How to compare the 3 algorithms

- **same implementation** except the selection operation which is specific to each algorithm,we have use the same implementation for the mutation and cross-over operation so that the difference between the results of the 3 algorithms could only come from their specific part. Thus it is possible to compare efficiently it.

  cross-over operation (arithmetic cross-over)
  r=random(); /* r random number between 0 and 1
  new_gene=r*(old_gene)+(1-r)*(old_gene)
  mutation operator
  gauss=random_gaussien();
  /*random number on a gaussian distribution
  new_gene=new_gene+gauss*(new_gene);

- **mutation and cross-over**
  we first need to determine which are the good values for the cross-over operator and for the mutation one. The fact is that some values could be the most efficient for one algorithm but not for the other; Thus we chose to take the best values for each algorithms and ...

- **comparison between the algorithms**
  then we have registered the fitness values

of the individuals for the 3 different algorithms after 10,50 and 100 generation so as to find which one of the 3 algorithms has got the fastest convergence.

6.2 The 3 main problems worked on

So as to compare the 3 algorithms we have 3 differents problems, described below, where differents sensitive points could have been point out. Moreover we have chosen these 3 ones because we knew some results from multi-objective optimization algorithm applied on, which is quite usefull to first check if the result of one of our algorithm was giving coherent result.

The problems used for the tests

- 1^{st} **problem** :
  1. single-variable problem with 2 functions to minimize
  2. min $f_1(x) = x^2$
     min $f_2(x) = (x-2)^2$
- 2^{nd} **problem** :
  1. two-variable problem with 2 functions to minimize and constraints to fit
  2. min $f_1(X) = 2+(x_1-2)^2+(x_2-1)^2$
     min $f_2(X) = 9x_1 - (x_2-1)^2$
  3. constraints
     $g_1(X) = 225 - x_1^2 - x_2^2 \geq 0$
     $g_2(X) = 3x_2 - x_1 - 10 \geq 0$
- 3^{rd} **problem** :
  1. single-variable problem with 2 functions to minimize and a Pareto front discontinue
  2. min $f_1(X) = x_1^2$
     min $f_2(X) = (1+10x_2)$
     $((1-(\frac{x_1}{1+10x_2})^2-(\frac{x_1}{1+10x_2}sin(8\pi x_1)));$

7 Results

7.1 Graphical results from the 3 problems

So we have obtained, by running the 3 algorithms on the 3 problems the following results - the results have been registered after 10, 50 and 100 generation.

Results for the 1^{st} problem pareto expected

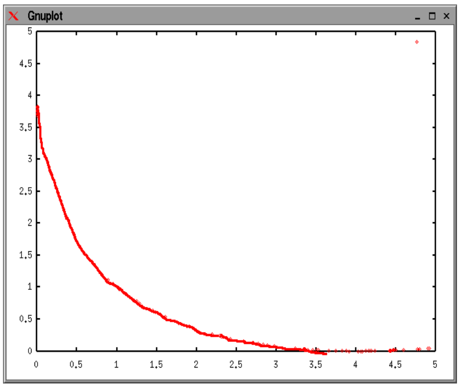

Figure 1: pareto expected for the 1^{st} problem

Results for the 1^{st} problem

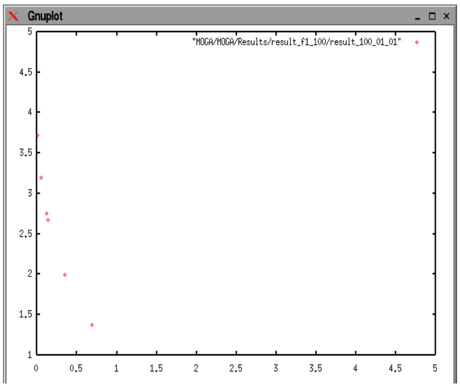

Figure 2: MOGA pareto after 10 generation for the 1^{st} problem

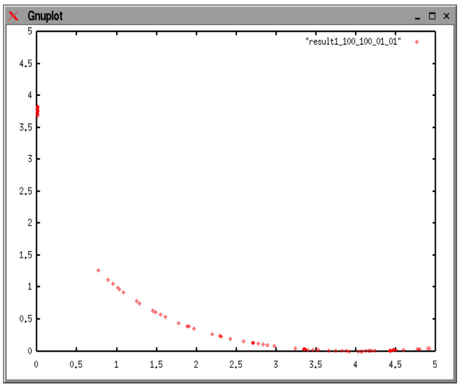

Figure 3: NPGA pareto after 10 generation for the 1^{st} problem

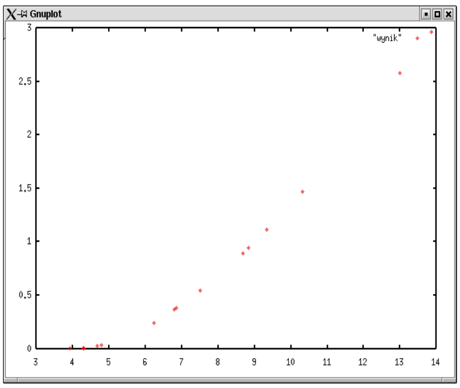

Figure 4: NSGA pareto after 10 generation for the 1^{st} problem.

For first problem 10 generation is definitely not enough to find Pareto front. Only NPGA algorythm reveals the shape of expected plot.

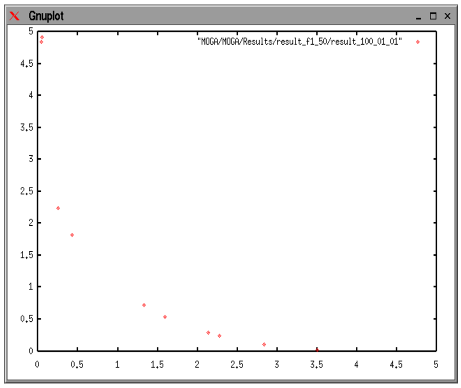

Figure 5: MOGA pareto after 50 generation for the 1^{st} problem

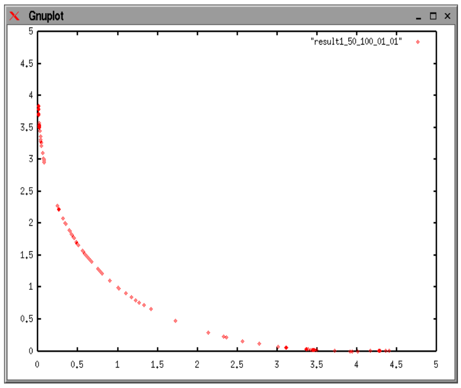

Figure 6: NPGA pareto after 50 generation for the 1^{st} problem

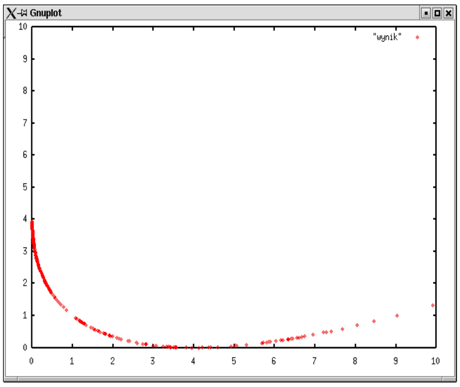

Figure 7: NSGA pareto after 50 generation for the 1^{st} problem

After 50 generations two algorythms generate results similar to expected. Only MOGA has some problems and shows a few Pareto solutions.

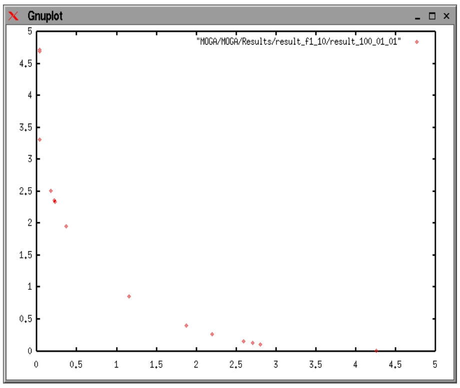

Figure 8: MOGA pareto after 100 generation for the 1^{st} problem

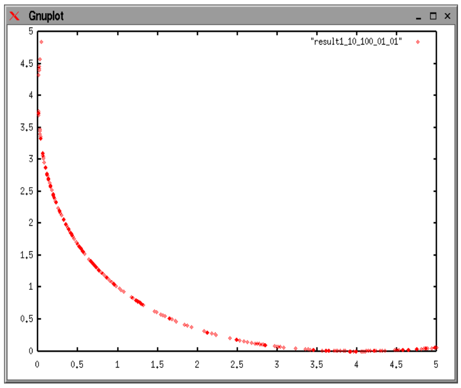

Figure 9: NPGA pareto after 100 generation for the 1^{st} problem

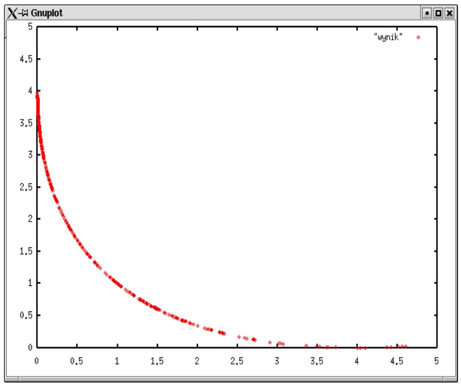

Figure 10: NSGA pareto after 100 generation for the 1^{st} problem

After one hundred generation clearly we can see Pareto front with the exception of MOGA. Points on curve are evenly spreaded out.

Results for the 2^{nd}

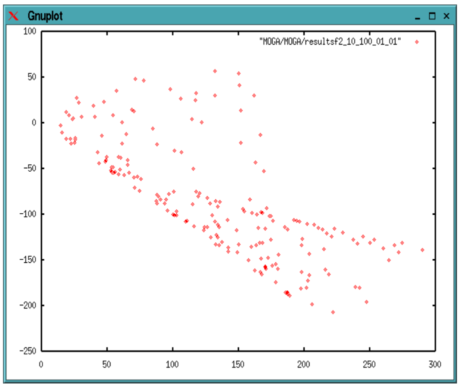

Figure 11: MOGA pareto after 10 generation for the 2^{nd} problem

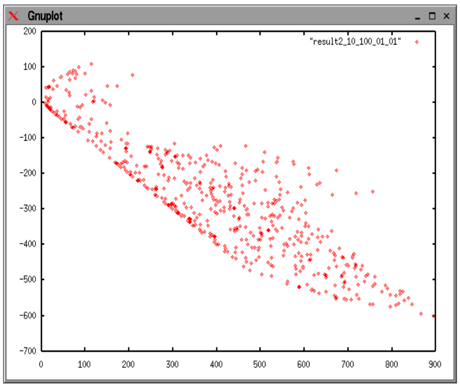

Figure 12: NPGA pareto after 10 generation for the 2^{nd} problem

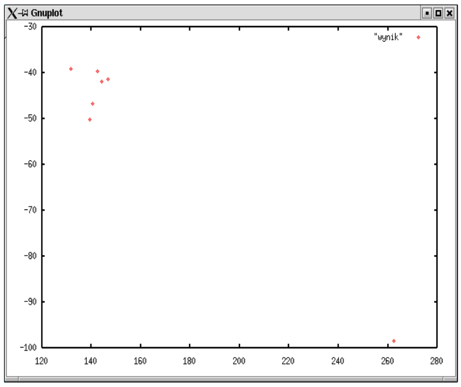

Figure 13: NSGA pareto after 10 generation for the 2^{nd} problem

For second problem after 10 generation MOGA and NPGA present very good results. On NSGA graph are visible only few solutions.

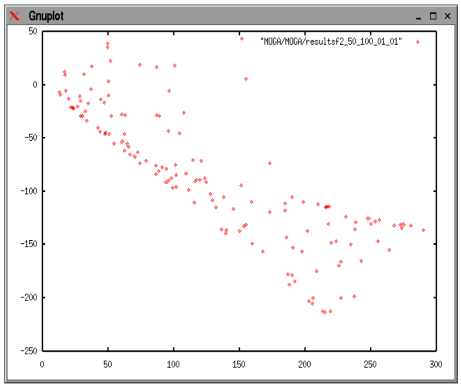

Figure 14: MOGA pareto after 50 generation for the 2^{nd} problem

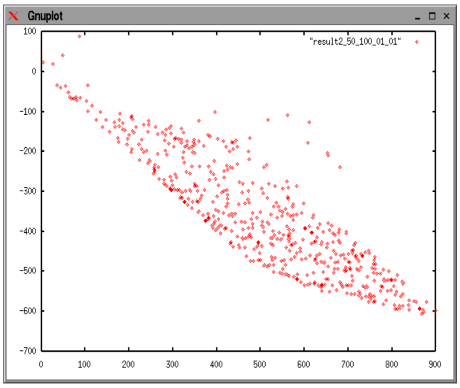

Figure 15: NPGA pareto after 50 generation for the 2^{nd} problem

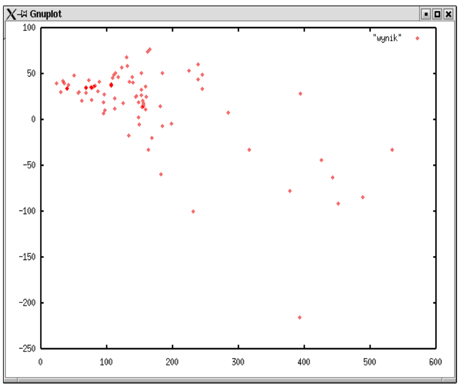

Figure 16: NSGA pareto after 50 generation for the 2^{nd} problem

50 generations don't change much. Two previous algorythms are still very good. From NSGA graph we can read that it found minimum of one objective and many points are close this place.

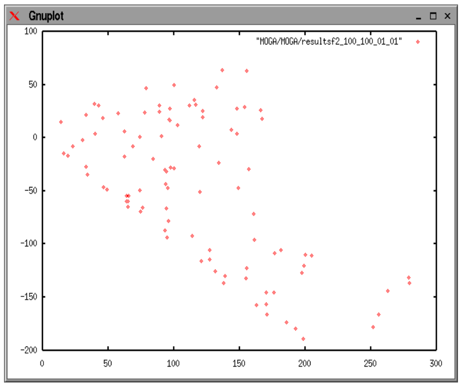

Figure 17: MOGA pareto after 100 generation for the 2^{nd} problem

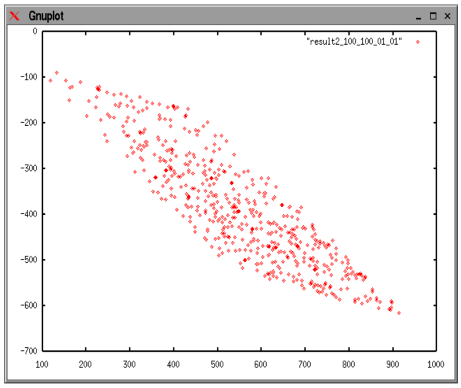

Figure 18: NPGA pareto after 100 generation for the 2^{nd} problem

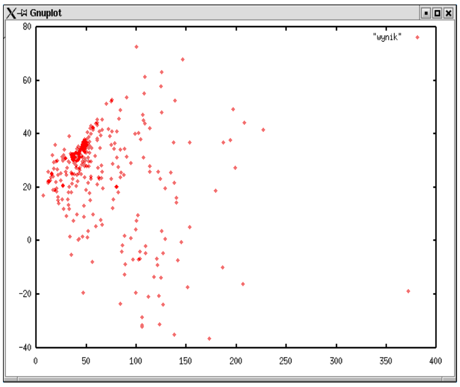

Figure 19: NSGA pareto after 100 generation for the 2^{nd} problem

Last comparison shows that NPGA clearly determine place of Pareto front. MOGA is less accurate. But NSGA is still located in one objective minimum with several dispersed solutions. It's necessary to add that graphs for second problem present solutions of ten runs of all algorythm. It means that there are 10 best fitness values for each individual. We were intend on doing it because of two-dimmensional space problems. Simultaneously only in this problem occured constraint functions which wasn't of no importance for choosing individuals.

Results for the 3^{rd} problem

- **Pareto for the NSGA after 500 generation**

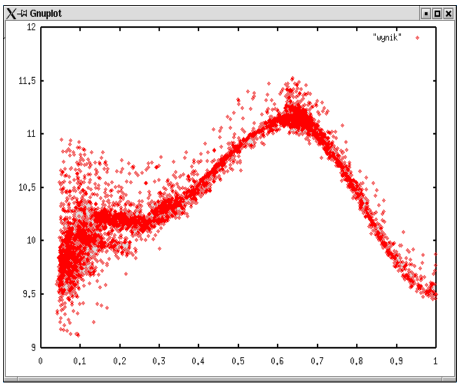

Figure 20: NSGA pareto after 500 generation

In that case only NSGA algorythm recevied some results. Here we run algorithms ten times for each individual, also. Nevertheless our outcome is very impressive.

7.2 Sensitive variables

After having done a lot of tests, we were able to determine what should have need more attention when testing and finally where were the difficult points; Thus we have notice some sensitive points which must be used carefully:

- **mutation and cross over operator**
  As for a simple evolutionary algorithms, the results are sensitive to the value for those 2 variables.

- **specific for the MOGA**

  1. *value for* σ_{share} : this is the only value the algorythm depends on. The smaller value the wider extension in space. It avoids grouping individuals on Pareto front.

- **specific for the NPGA**

  1. *value for* t_{dom} : this is the value for the size of the set of individuals to compare with in the selection operation. If it's too high, premature convergence may result; If too small, there will be few non-dominated points in the population.
  2. *value for* σ_{share} : this is the value determining the size of the area where to count the neighbours of one point for the *niche count* value. Thus if it's too small, it could be possible that none neighbour is found.

- **specific for the NSGA**

  1. *first Pareto front value* : the speed of the convergence will be influenced by this value.
  2. *value for* ϵ : this is the difference between the dummy fitness of the next non-dominated front with the current one.
  3. *value for* σ_{share} : this is the parameter which determine sharing in current front. Thus the greater, the smaller are the new fitness values. And if it's too small then the new fitness value will not change so much and the speed of the convergence will decrease.

8 Conclusions

The two main conclusions that we could point out are the following :

1. **usefull algorithms** Due to the fact that it is possible to find a set of solutions fitting to the variables of the problems, people are able to choose what solution could most fit to their own interests

2. **sensitive algorithms** Nevermind these algorithms are quite sensitive to some of the values for which there is no theorical methods to find what will be the best value and then people could spend a lot of time -and money - before finding the best one...

Meanwhile the work we have done maybe needs to be improved in some parts, as in the research for the good values for the sensitive variables, and in the protocol of comparison, which could be defined also differently.

References

[1] C.M. Fonseca and P.J. Fleming. *Genetic algorithms for multi-objective optimization:Formulation,discussion, and generalization*. 1993.

[2] J. Horn, N. Nafploitis, and D.E. Goldberg. *A niched Pareto genetic algorithm for multi-objective optimization*. 1994.

[3] S. Rao. *Optimization theory and aplication*. 1991.

[4] N. Srinivas and K. Deb. *Multi-Objective function optimization using non-dominated sorting genetic algorithms*. 1994.

[5] K. Tamura and S. Miura. *Necessary and sufficient conditions for local and global nondominated solution in decision problems with multiobjectives*, chapter 27, 509-523. 1979.

Optimizing a Chess Engine with EA-Techniques

Jens Ebbesen and Jesper Hedegaard

Abstract— **This paper introduces an approach for tuning the parameters of the evaluation function of a chess engine based on various evolutionary algorithms.**

We use a technique based on the ELO-rating system used in real chess playing societies for overcoming some of the difficulties with a competetive fitness function. For deciding the simple parameters of the evaluation function we use an evolutionary algorithm and the ELO-technique. This is extended with a Religion Based Evolutionary Algorithm when trying to tune the more advanced parameters.

The test run on our implementation have been incouraging showing clear improvements over the hand coded evaluation function.

1 Introduction

Making computers play chess has been a field of interest since as early as 1947 where Alan Turing specified the first program for playing chess. The chess programs has come a long way since then. In 1988 Bent Larsen became the first grandmaster to lose to a computer in a major tournament and in 1997 IBMs Deep Blue defeated the world champion Garry Kasparov in a famous 6 game match.

Even though the main progress in the preformance of the chess computers is due to much faster machines there is of course also much focus on optimizing the chess engines. Our ideas is to use evolutionary algorithms to improve the evaluation function used for evaluating board positions. The evaluation function consists of some parameters which values is somewhat unknown and it is here we hope that the EA could be used for the fine tuning of these. We have made a distinction between what we call simple and advanced parameters. The simple parameters are those we are certain that are needed (e.g. the material advantage) and the advanced are those which could be useful.

There are other papers [1] describing an EA approach for tuning the weights of the pieces. These have used other techniques and have not gone beyond the simple parameters.

2 Chess engine

To understand the problems we address in this paper it is necessary to have a basic understanding of a chess engine. Therefore we will begin by explaining the basics.

The most simple algorithm describing a chess engine looks like this.

> Calculate all legal endpositions after at most n moves.
>
> Evaluate the endpositions and choose the best one.

One way of improving the chess engine is to increase the number of moves it is able to calculate within reasonable time, or actually to reduce the number of endpositions that needs to be examined in order to find the best possible endposition. Doing this will enable the computer to look further ahead and thus improve its play.

Since this has more to do with finding smart algorithms and representations it is not the focus of this project and we only considered this during the implementation of our chess engine.

The other way is to improve the evaluation function so it is more precise in evaluating a given board. It is this function we will try to fine-tune in this project.

The evaluation function is basically a numerical function. The most simple version simply makes a weighted count of the pieces of both players and computes the difference. Since this is clearly not enough to correctly evaluate the position most chess engines looks at other features as well. These could include king safety, center control, etc.

The weights of both simple and advanced features can be optimized usings EA techniques and this is what we have done in this project.

3 Tuning of simple parameters

The most important part of the evaluation function is the weights assigned to the different pieces. The suggested weights in almost all chessbooks are: (the king is set to infinite):

- Queen: 9
- Rook: 5
- Knight: 3
- Bishop: 3
- Pawn: 1

These should be taken merely as rule-of-thumb values and not all people agree on them. Therefore it is possible that they could be optimized to better fit the true value of the pieces.

Since we need to use some sort of competitive fitness and this can only be achieved by playing games (which takes a great deal of time) we needed a way to ensure somewhat quick and accurate convergence. One simple thing we did was always to include one element which had the values shown above in the start population. The other ideas will be described in the following subsections.

3.1 Selection

In each generation each individual will be compared to another element this is done by playing two games of chess where each player plays one game as white and one as black so to eliminate the first-move advantage. Both individuals are chosen randomly among the elements which has not played in this generation. This means all individuals will face exactly one other individual per generation.

We need to play *popsize* games in each generation which encourages the useage of as few individuals as possible due to the time needed to play a game at the desired depth.

The combined winner (see below for details) of the two games are kept unchanged whereas the loser is mutated.

3.2 The ELO-rating approach

The ELO-approach is basically an aid to the mutation and crossover. It is designed to rank all individuals based on all their previously played games.

The ELO rating system is used by FIDE (The international chess federation) to rank chess players.

It is based on the expected result given the players previous rating. It is calculated using this formula:

```
Rating_new = Rating_old +
   C*(Res_of_game - Expected_res)
```

The constant `C` is normally set to 30, i.e the most you can win in one game is 30 points. The result of the game is 1 if you have won and $\frac{1}{2}$ if the game is drawn (only the winners new rating is calculated, the loser simply loses the same amount as the winner gains). The expected result is given by a Gaussian distribution of the difference in rating with $\mu = 0$.

The ELO-rating will act as a guideline when performing mutation (the higher the rating the lower the mutation) all individuals start with a rating of 1300.

The ELO-rating is also used to determine which individual will be mutated after two individuals have played. Instead of simply looking at the results of the games we check which individual has improved its rating and declare this element the winner. This means that although two individuals have drawn both games we might pick a winner and thereby increasing the pressure on high rated individuals. However, their rating ensures that the consequence of losing is relatively small.

3.3 Mutation

When we mutate an individual we first find the weight where the distance between the winner and the loser is highest, this weight is treated differently from the others and will be described later. On the remaining three weights a simple uniform mutation is performed. The range of the mutation depends on the generation (simple linear decrease) and the rating of the individual.

We have chosen not to use standard crossover since it seemed to make the convergence rate to high which meant our individuals had a tendency to converge to a suboptimal value. Instead we have used something we describe as

Directed mutation which means that instead of the normal mutation you mutate in the direction of the element that defeated it. It is basically arithmetic crossover where the weight is determined on the rating of the losing individual (although there is nothing to prevent the use of a randomly determined weight instead).

As described above we use directed mutation on the weight where the distance is greatest and standard mutation on the rest of the weights. It is clear that increasing the number of weights on which directed mutation is performed will increase the convergence rate and make the process seem more like normal arithmetic crossover. But we have chosen to limit ourselves to one parameter to ensure that we did not experience a large drop in the diversity in the beginning. The decition to always look at the weight with the biggest difference instead of picking a weight at random was because changing this weight would probably have the biggest positive affect on the mutated individual.

3.4 Discussion

We needed an algorithm that would converge fast and toward the global optimum using only a few individuals and a competitive fitness function. These are hard demands but as the following section will show our algorithm fulfils these demands. We believe that one of the main reasons that our algorithm performs as well as it does is the ELO-rating scheme. The reason we experience as fast convergence as we did is probably that our algorithm has some similarities with a PSO with respect to the changes of the individuals between generations. This probably also explains why the algorithm performs as well as it does with so few elements (our main runs have been performed using 12 and 24 individuals).

We have tried to test the algorithm on a few of the standard numerical problems (DeJong F2 and Schaffer F6) by simply replacing the two games of chess with a comparison of the two individuals value on the given problem. We did not spend much time on this so our results were pretty basic and in a few cases the mutation rate was not changed to fit the problem. Still the result we got from this experiment seemed to indicate that with a few modifications it would be able to compete with standard EA's and possibly others as well. A small test was run using 75 individuals which improved the results somewhat.

We have seen indications that there is not one perfect setting for the weights but a few setting which can defeat each other. Whether this is because we still have not found the optimal setting or no such setting exist is hard to say. However all the best individuals were quite similar so although there might not be one perfect setting we could assume that there are a range in which good settings would occur.

4 Results

Our main test was performed using 12 individuals running for 55 generations with a search depth of 5. Only one run was performed due to the time needed to perform such a run.

The initial population had one seeded element. The seeded element was include to ensure faster convergence. As far as we could tell it survived the first five generations which would seem to indicate that it was indeed the fittest at this point but since it was then forced to mutate it meant that there where better settings.

The rest of the individuals was created with random weights in the range $[1 \ldots 11]$.

The average weights for the initial population was:

- Queen: 6.20
- Rook: 4.71
- Knight: 4.40
- Bishop: 4.42

and the standard deviation value for each parameter was:

- Queen: 2.50
- Rook: 2.22
- Knight: 2.38
- Bishop: 2.30

The average and standard deviation for each parameter can be seen in figure 1 and figure 2. At the end of the run the values had converged to the following values:
Average:

- Queen: 8.41
- Rook: 5.34
- Knight: 2.93
- Bishop: 3.10

Standard deviation:

- Queen: 0.28
- Rook: 0.41
- Knight: 0.33
- Bishop: 0.28

As one can see there had been a clear convergence towards the previously mentioned rule-of-thumb values which seems to indicate that the algorithm works. Still there is a clear difference between the two sets of values therefore we chose the let the two settings play each other twice and in both games the player that used the new values won which also seems to indicate that the experiment was succesful.

5 Using Religion Based EA to find features

While there can be little doubt that it is necessary to look at material when deciding if a position is good or bad but there is less certainty with regards to what other features to include. We have handpicked 6 different features some of which we believe have a great importance others which are more doubtful. There have been other experiments [2] in which it has been tried to make an EA find the features from scratch. What we wanted to do was simply have the EA test the validity of the different features and if they were found useful to determine their weights.

5.1 The setup

We have chosen to define the religion as a binary vector of length 6. Where each bit represents whether a given features is enabled or disabled in this individual. We allow for new religions to emerge during the run this is simply done by making an `or-operation` between two vectors thus creating the superset containing both previous religions. The individuals are placed in a grid with wrap-around so that all individuals has the same number of neighbours.

We will now show the pseudo-code for our religion based EA. First we will show the overall algorithm and then some helper functions.

```
initialize population
while not stopCondition do
  for each individual do
     if not (individual used) do
        choose_random_neighbour
        if neighbour has_same_religion
          then share_knowledge
        else try_to_convert_neighbour
        neighbour used <- true
        individual used <- true
     end
  end
end
```

We have chosen to allow an individual to choose between its neighbours without considering whether these have been used or if they have a different religion. This is a bit different than the original algorithm [3]. The reason for this alteration to the algorithm is that we want them to learn from each other without necessarily having to play games. This is done by using their ELO-rating when they have the same religion as seen in the following pseudo-code.

```
share_knowledge:
  if individual1.rating>1350 &&
     indidual2.rating>1350 then
       play_games_and_use_results
  else if max_rating > 1350 &&
          min_rating < 1250 then
            advancedMutation_on_
            lowest_rated_individual
  else if individual1.rating<1250 &&
```

```
          individual2.rating<1250 then
           simple_hard_mutation_on_both
  else simple_mild_mutation_on_both
```

We have chosen to use 1350 and 1250 based on our experiences from the first tests. Here it was clear that individuals with rating below 1250 were far from optimal while individuals with rating above 1350 normally had quite good setting. However, there is nothing absolute about these limits.

The play_games_and_use_results are basically the same as described in the first part of the paper only applied to the six advanced features this time.

The advancedMutation is again just using the approach described in the first part - i.e using directed mutation and so on.

The mild_mutation and hard_mutation are methods applying standard uniform mutation on all six weights. The only difference in the two methods is the mutation range.

Since both mutation operators might set the weight of an enable feature to 0 the religion of an individual might change in other ways than convertion. After the mutation is performed we check if any weights have become 0 and if so changes the religion accordingly.

The most important part of the algorithm is the way in which individuals change their religion the following pseudo-code will show how this is done.

```
try_to_convert_neighbour:
 play_two_games
 if clear_winner then
  if won_both_games then
   if random<0.9 then convert_loser
   else advancedMutation_on_loser
  else
   if random<0.55 then convert_loser
   else advancedMutation_on_loser
 else
  if won_one_game_each then
   if random<0.2 then merge_religions
   else simple_mild_mutation_on_both
  else
   if random<0.8 then merge_religions
   else simple_mild_mutation_on_both
```

The convert_loser method simply changes the religion of the loser. Features which are enabled through this process get their weights from the winning individual instead of being initialized at random.

The merge_religion method performs an `or-operation` on the two religion-vectors thus creating the new religion. As in the case of convertion the weight for all new features are taken from the other individual.

The reason we use random is to ensure some level of diversity, the different limits are set based upon educated guesses but different values could possibly improve performance.

5.2 Discussion

Normally you need quite a large population in order for a RBEA to work reasonably well but as mentioned before the timefactor works as a constraint on the number of individuals we are able to use. However, since our main goal was not optimizing but recognizing good features the problem is not that serious and as you will see in the next section our results proved to be quite good.

One could argue that an individual with the correct features but bad weights might lose to an individual with less optimal features but optimal weights thus preventing us from finding good features. This could be a problem. It is hard to check if this actually happens during the run. In our case we have the advantage of using only few features and our merge operation always creates a superset. Furthermore most of the features must be enabled in the optimal setting. These three factors means that although we might loose good individuals sometimes the chance that others with the same features will be created later on is quite high which limits the possible problem. A way to solve the problem is to allow mutation on religions so we can ensure a higher level of diversity.

There is also a minor issue regarding knowledge sharing within a religion. Currently there is only a true sharing of knowledge if at least one individual has a rating of more than 1350 this means that lower rated individuals might not learn anything at all but will simply be forced to mutate all the time. This might influence the credibility of the averages. The main reason we have chosen to limit the sharing of knowledge this way is as always a time concern to ensure that we could have a relative large population we needed to reduce the number of games plays and this was the most obvious way.

6 Results

Our main run was performed using 49 individuals in a 7x7 grid running for 30 generations again with a search depth of 5. The population was given the weights for the pieces found in the first experiment.

Each individual started with only one of the six features enabled. The weight for this feature was chosen at random between 0 and 1. We chose to start with only one feature for each individual so that all features had the same chances of survival and so that individuals themselves could choose between the possible features.

In the following the religion will be intepreted as the integer value of the binary vector. The features in the vector where as follows (from right to left).

- Bonus for central pawns on 4th or 5th row
- Penalty for knight on boundry
- Penalty for bishop on own 1st row
- Penalty for double pawns
- Penalty for passed pawns
- Bonus for king safety

The distribution among the initial religions where:

- Religion 1: 9
- Religion 2: 3
- Religion 4: 13
- Religion 8: 8
- Religion 16: 10
- Religion 32: 6

After 30 runs the distributions among the represented religions where:

- Religion 51: 41
- Religion 59: 8

One feature is absent in the remaining religions namely the penalty for bishops on own 1st row. When looking at the possible impact of the different features the one concerning the bishop is likely to be the least effective (the only reason we included this feature was to promote good opening play from the players). This meant that in the case that one feature should be removed then it was our most likely candidate.

Since a large portion of the possible religions where represented at one point or another we will not show a graph of the changes in the size of the different religions. Instead we have made a graph showing the number of individuals for each feature (figure 3).

The program ran quite slow in the beginning since there were very few individuals with same religion near each other meaning that nearly all the times an individual tried to convert one of its neighbours. As the number of different religions decreased so did the time it took to run a generation.

The average weights for the six features will not tell us very much in the beginning since all of them were set to 0 in most of the individuals due to the way we initialized the population. A graph showing the change during the generations can be seen in figure 4.

The final averages were as follows

- Central pawns: 0.33
- Knight on boundry: 0.44
- Bishop on own 1. row: 0.0
- Double pawns: 0.03
- Passed pawns: 0.38
- King safety: 0.53

95%-confidens intervals were calculated for five of the six weights to check how good the values were and although the length of the intervals where a bit long they did not give cause for concern with respect to the knowledge sharing.

As with the first experiment we let a player using our new evaluation function play one using the evaluation function that was found to be the best after the first experiment. Each player won the game where they played as white. This could indicate that the impact of these extra features are not that big or that the weights we found still need some finetuning.

7 Conclusions

We have introduced a method to help make an approximate global comparison of individuals in a situation where competive fitness is the only true way to compare the fitness of two individuals. This could improve the use of mutation and crossover in these situations since the amount of information is improved.

As mentioned earlier we tried the approach on a few of the standard numerical problems and got promising results given the amount of time spend tuning the algorithm. This would seem to indicate that the method is actually valid in more cases than just the specific chess problem. It is clear, however, that it is not a very sound approach in cases where a global comparison is possible or where the evaluation is somewhat faster than our chess problem since in these cases it is easy to compare more individuals and thus getting a more precise idea of an individuals fitness compared to the rest of the population.

In our two experiments we chose to fix some of the weights and only try and improve on the others. This was done primarily to improve the running time since you would expect it would take more generations to finetune 10 weights instead of just four or six. With a faster implementation or more time it would be worthwile to allow the EA to change on all 10 weights since there can be little doubt that there are some depency between the two type of weights and this will be missed with our approach.

As mentioned in the RBEA section we had some doubt regarding the spreading of good values within the same religion and there where some small indications from our results that these doubt were valid. Therefore it could be a good idea to make a run where more games were played within each religion which would increase the convergence within the religion and hopefully thus spreading good weights through the population. The only major drawback would be the time such a run would take.

Our main problem during the experiments was the running time of the EA. Unfortunatly there is not much you can do about it the only way to truly improve on this aspect is to improve the running time for playing a game. Luckily we found that even with less than 25 individuals it was possible to produce good results. However there was a clear improvement in the values of the numerical problems as the number of individuals increased. This could indicate that running our first test with more individuals could produce better values but as always with the cost of an increased running time. Moreover since the fitness of the chess pieces is more noicy than that of a numerical problem there might not be the same clear improvement in the values.

It is worth remembering that the values we found where optimized with regards to some external constraints such as the search depth, the specific features included etc. and there is therefore no guarantee that the values will be optimal for any other chess engine.

The evaluation function used in our engine is very basic and might not perform that well against skilled players or other chess engines. The main drawback is the fact that it uses the same evaluation in all parts of the game instead of having different evaluation function in different parts of the game which would probably improved its performance. In a case like ours where the main task was to improve upon the performance of a chess engine it worked quite well and the simplicity was actually an advantage and most of the techniques discribed in this paper could just as well be applied to more advanced chess engines.

References

[1] Graham Kendall and Glenn Whitwell. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 995–1002, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

[2] Henning Korsholm Rohde and Wouter Boomsma. Feature extraction in chess. In Rasmus K. Ursem, editor, *Topics of Evolutionary Computation 2001 – Collection of Student Reports*, volume 132 of *IR*, pages 7–12. Dept. of Computer Science, University of Aarhus, Denmark, 2001.

[3] Rene Thomsen, Peter Rickers, and Thiemo Krink. A religion-based spatial model for evolutionary algorithms. In Marc Schoenauer, Kalyanmoy Deb, Günter Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature – PPSN VI*, volume 1, pages 817–826, Inria hotel, Paris, France, 2000. Springer.

8 Graphs

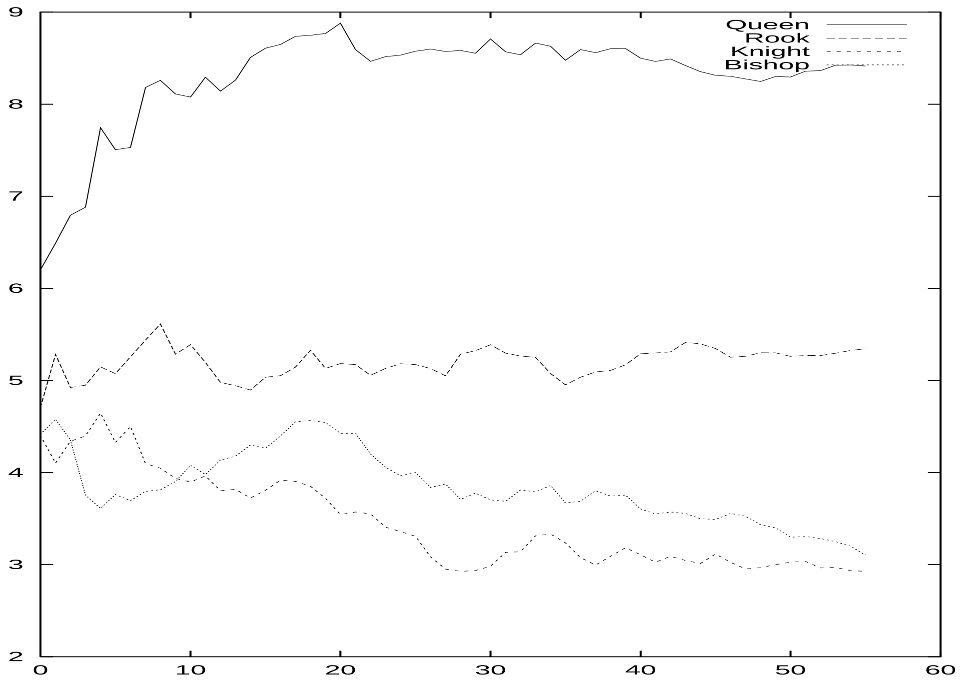

Figure 1: Average of piece values

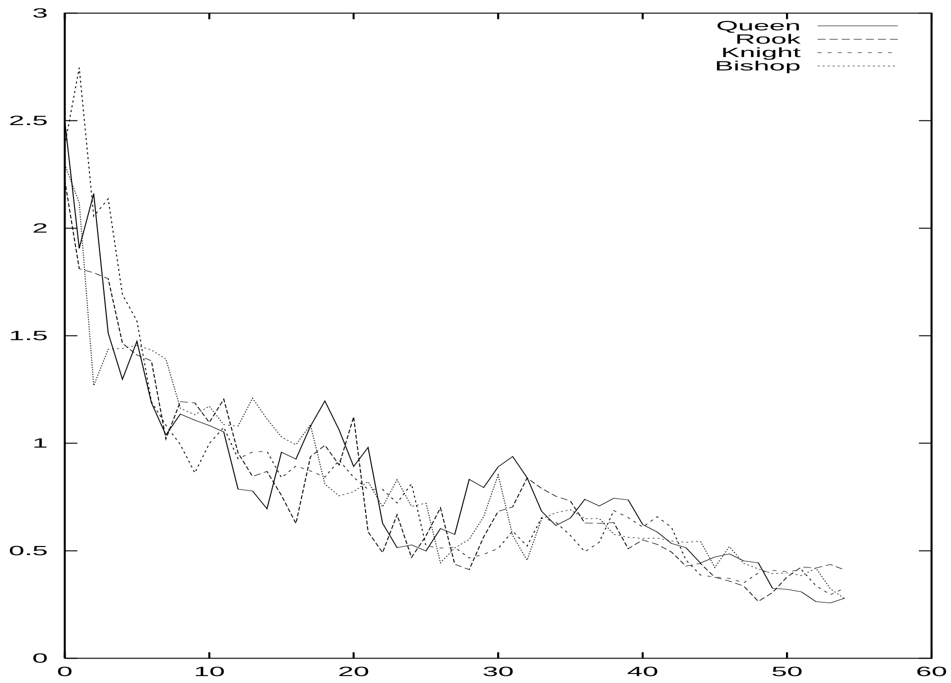

Figure 2: Standard deviation of piece values

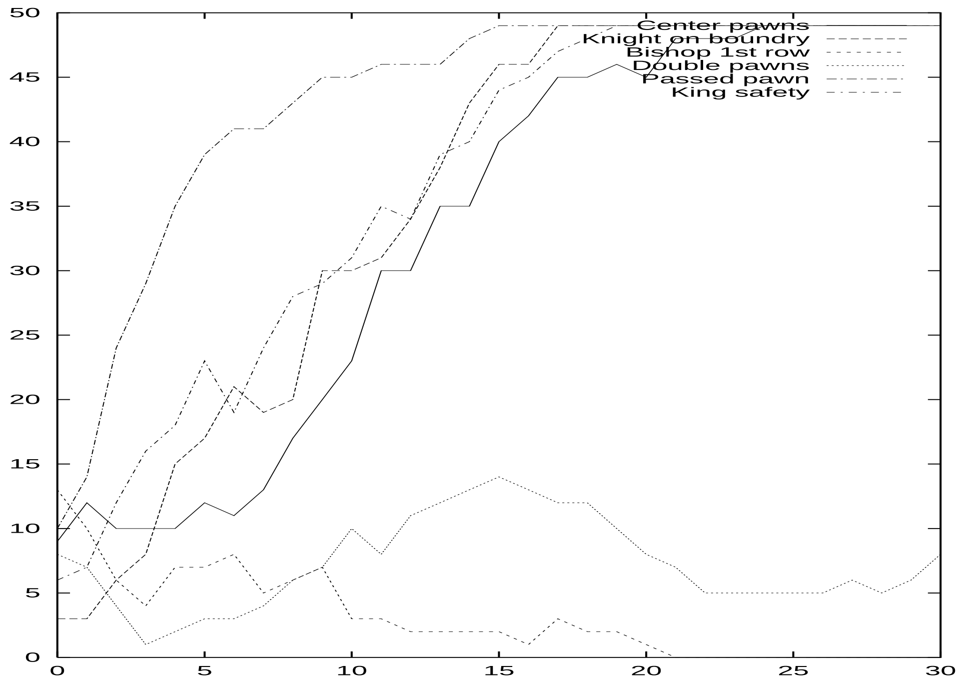

Figure 3: Feature distribution

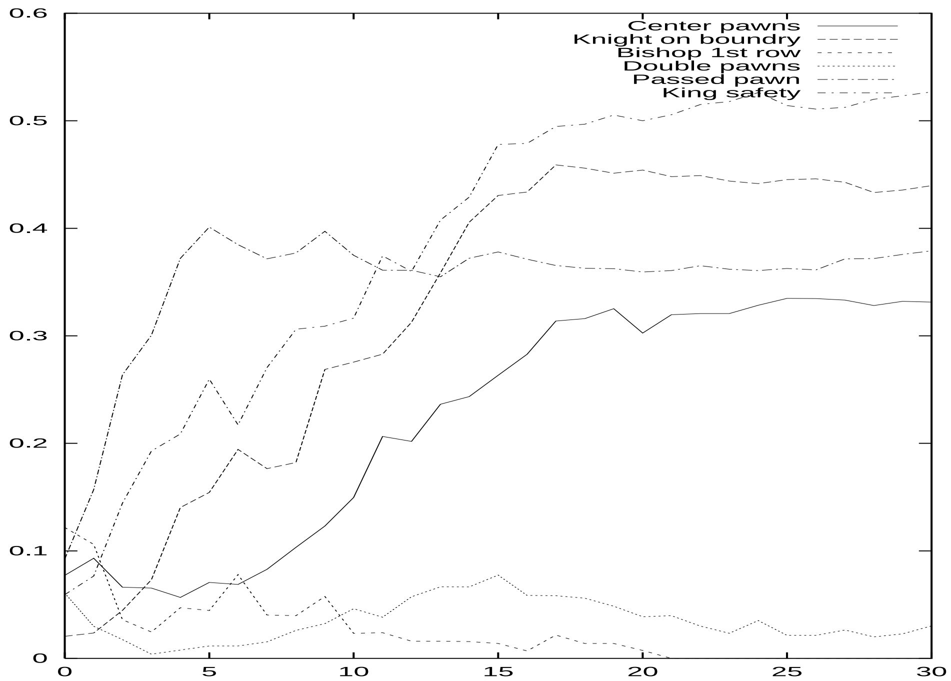

Figure 4: Average of advanced weights

Recognising spam using neural networks

Martin Stig Stissing and Lars Hesel Christensen

Abstract—**This report describes a method for classifying mails as spam or non-spam mails. We have used different approaches training neural networks. The training methods used are based on known techniques such as backpropagation and evolutionary algorithms. A great deal of effort has gone in to finding appropriate features for the network. Having other attempts of identifying spam in mind, we have reused a lot of their ideas as features for our approach. We have used a well known corpus as well as our own homemade collection of emails for training and validating the obtained neural networks. A comparison between our own approach and the approach in [1] is given.**

1 Introduction

This project examines possibilities for using an artificial neural network as a means for recognising unsolicited email, which is also commonly known as spam. We utilise different previous approaches, and incorporate some of their thoughts and ideas as an inspiration for finding useful features as input to our neural networks. The main reason for finding such useful features is to eliminate as much spam as possible, ultimately making spamming an unprofitable career choice. At the present time around 37 percent of all emails on the internet are spam [2]. Handling these takes up a lot of resources, both time, money and bandwidth, and is ultimately a large economical expense for society. Getting rid of these spams can of course be done in a trivial way, that is, simply block every email, but usually you would like to let as many ordinary (legitimate) mails stay untouched and uneliminated at the same time. That is, minimising the number of so called false positives and maximising the number of true negatives.

We have mainly investigated two methods for training neural networks and will describe them in more detail in the following.

2 The neural network

Many different neural networks has been used in the different stages of the project. They all share the property of having an output layer of size one, which is basically the neuron used for determining whether or not a given mail is a spam mail (or at least identified as one). As our ideas for identifying features evolved we upgraded the number of neurons in the input layer and hidden layers. At the present time we have around 40-50 neurons in the input layer, half of that in one hidden layer and one neuron in the output layer. What the values of these input neurons represent are further exploited in the following sections.

2.1 Realization

We have used the Dendrite [3] package for implementing our neural networks. This package allowed us, in a simple manner, to create the standard fully connected networks that we needed. Furthermore it already had one of the standard training methods, namely backpropagation, implemented, which, in short time, allowed us to obtain some preliminary results.

3 The corpora

We have used two different corpora for training and testing the network. The first one used is a well-known spam identification corpus, dubbed the *Ling-Spam corpus*[1] from a moderated list for linguists. It contains 2412 randomly downloaded linguist messages and 481 personal spam messages. All the messages have had their headers, html-tags and attachments removed, so it is only the bare contents of the message that remains. This is actually a disadvantage for our approach, since we utilise many of these as features for our network. The other corpus is a homemade collection of legitimate mails and spams. The legitimate mails are from our own in box, and the

[1] http://www.aueb.gr/users/ion/lingspam_public.tar.gz

spams stem from the *razor-repository*[2] of the institute. These mails are just the raw mails received and therefore contains the full information for our feature extraction. This corpus has 600 mails and 600 spams. This corpus was mainly used in the startup and development phases of the project. The Ling-spam corpus was used to obtain results that was comparable to previous results [1].

4 Feature Selection

As a first attempt of identifying useful features one could, as we did at first, look through a lot of emails, and for instance try to identify some of the words, that seem to capture the essence of spam and likewise for legitimate mails. The presence or absence of these (a limited number of) words could then be fed to the input nodes of the neural network.

In our final approach we have also been inspired by this way of choosing features, but we haven't used a limited number of words and their presence, but instead, inspired by the naive Bayesian classifier from [1], used the *frequency ratio* taking into account the relative number of mails and spams.

$$FR_{word} = \frac{S_{word}}{L_{word} + S_{word}}$$

where L_{word} and S_{word} are the number of legitimate mails and spam mails containing $word$, as features. In reality we only use the most interesting of these, which is the ones where $|FR_{word} - \frac{1}{2}|$ is as close to $\frac{1}{2}$ as possible. We distinguish between whether FR_{word} is close to 1 or 0, and use a fixed number of both as inputs for the network. This should (hopefully) enable the network to be able to correctly classify the mails even though most of the words are "spam-words", but where the mail in fact is legitimate.

We also have some features which are just concrete versions of ones informal description of what characterises (mainly) spam mails

- how many special characters ($, !, #, .) are there (of each kind)
- how much of the mail is capital letters

[2]http://razor.sourceforge.net/

- longest and average length of capitalised words
- longest and average length of run of numerical words
- are there attachments

All of the above mentioning capitalised words was not applicable to the Ling-spam corpus, since all mails in that corpus were in lower case. The attachments presence has also been removed from the corpus.

These features are retrieved from the spam and mail files using a hereto appropriate language.

4.1 Realization

We specifically needed a tool that could scan mails and spams for features, count occurrences of words (for some appropriate definition of words) and finally output some XML which is one of the formats our neural network framework [3] was able to load.

We chose perl as the language for processing and extracting features from our corpora. The reason for this choice is that one of the authors has had previous experience with perl and that perl has some very powerful tools for reading, manipulating and generating strings, XML(eXtended Markup Language) and other various input and output formats.

All in all 10 training sets and 10 validations sets was generated from the corpus.

This was done by dividing the mails and the spams into 10 disjoint sets and then remove each set one at a time and generate a frequency table of word occurrences on the remaining 9 sets. From this frequency table and the 9 sets we also generated a training set. A validation set was generated from the same frequency table using the last set (the one that was removed).

Doing this 10 times, each time taking one of the 10 sets, we created 10 pairs of training sets and validation sets.

5 Training the neural network

When training the neural network we used two different approaches. The classic backpropagation algorithm and an evolutionary algorithm. The backpropagation algorithm, was already included in the dendrite package, so this approach was pretty easy. The evolutionary algorithm trained the weights of the neural network, having each individual as a real valued vector with as many elements as there are connections in the given network (this actually often results in individuals with 1000 elements or more). We let the evolutionary algorithm be mainly self-adaptive, meaning that each individual would have its own probabilities for mutating and recombining, as well as, for each element in the individual, having its own mutation range. This has turned out to work well, and has also (by definition) removed the need to hand tune the parameters for mutation and recombination probability and the parameters for the range of mutation. Another good thing about using an evolutionary algorithm for training (or computing information about) the network is that it could easily be tuned to handle other problem-dependent values or settings of the network, for instance the topology of the network could be examined by letting an evolutionary algorithm propose different promising topologies and hereafter using either of the above training methods as a basis for a fitness function.

When using the backpropagation algorithm (and also when specifying the topology of the network) one has to adjust some parameters (learning rate, momentum, ...) to make the training as efficient and successful as possible. Here we have put some effort in finding the appropriate values for each, but since these could be changing over time, one could argue that (some sort of) an evolutionary approach could be more appropriate here as well.

6 Performance measurements

Regarding classification tasks, performance is for the most measured in terms of *accuracy* (Acc) or *error* ($Err = 1 - Acc$). Adopting the notation and measures (mainly for easy comparison) from [1], we let N_{legit} and N_{spam} denote the total number of legitimate and spam mails to be classified, respectively, and $n_{C_1 \to C_2}$ denote the number of mails from C_1 classified, by a filter, as mails from C_2, where C_1,C_2 $\in \{legit, spam\}$. Then Acc and Err are

$$Acc = \frac{n_{legit \to legit} + n_{spam \to spam}}{N_{legit} + N_{spam}}$$

$$Err = \frac{n_{legit \to spam} + n_{spam \to legit}}{N_{legit} + N_{spam}}$$

But often it is the case that it is many, say λ, times more "costly" classifying a legitimate mail as a spam mail than doing the opposite. Therefore we let the *weighted accuracy* ($WAcc$) and *weighted error* ($WErr = 1 - WAcc$) be

$$WAcc = \frac{\lambda \cdot n_{legit \to legit} + n_{spam \to spam}}{\lambda \cdot N_{legit} + N_{spam}}$$

$$WErr = \frac{\lambda \cdot n_{legit \to spam} + n_{spam \to legit}}{\lambda \cdot N_{legit} + N_{spam}}$$

Traditionally in measuring the performance of a classifier one compares it to a "baseline" classifier, which in this case is taken to be no filtering (letting each legitimate mail and each spam mail pass the filter), to find the *total cost ratio* (TCR). The $WErr_{baseline}$ is then clearly:

$$WErr_{baseline} = \frac{N_{spam}}{\lambda \cdot N_{legit} + N_{spam}}$$

and TCR then

$$\begin{aligned} TCR &= \frac{WErr_{baseline}}{WErr} \\ &= \frac{N_{spam}}{\lambda \cdot n_{legit \to spam} + n_{spam \to legit}} \end{aligned}$$

The TCR could be seen as a measurement of how costly (time-consuming) it is to delete the N_{spam} spams manually, in comparison to how costly it is to use the filter and delete the $n_{spam \to legit}$ classified spams manually plus recover from the $n_{legit \to spam}$ legitimate mails caught by the filter (assuming that each recovery takes λ time). As an example a TCR value of 2 would indicate that using the filter would only be half as costly as not using it.

For the sake of completeness we also present (as [1]) the *recall* and *precision* measures, concepts familiar to the science of information retrieval. The *spam recall* is the basic efficiency of a given filter (how many percent of the spams it correctly classifies)

$$SR = \frac{n_{spam \to spam}}{N_{spam}}$$

and the *spam precision* is intuitively how safe it is to use the filter

$$SP = \frac{n_{spam \to spam}}{n_{spam \to spam} + n_{legit \to spam}}$$

One should clearly strive for SR and SP being 1, implying that all spam is correctly classified and that no legitimate mail is classified as spam, basically that everything has been correctly classified. The TCR has the advantage of incorporating λ, and hereby utilising the difference of cost, giving a more unified measure for comparison of different filters, whatever their milieu might be.

7 Experimental results

We have used 10-fold cross validation during the network training (the *Ling-Spam corpus* was already divided into ten different parts), as a means for maximising the use of the data for training, as well as getting a relatively unbiased validation result. The cross validation simply takes nine of the ten parts and uses this as its training set and then the last part for validating (these sets are further described in 4. This is then done for each of the ten parts. The results of these validations (or actually the average over some runs) of these is then used in calculating the TCR and SR and SP as above. We have used 3 different values (as [1]) for λ, namely 1, 9 and 999 (the training methods have been punished λ times as hard for classifying a mail as a spam as the opposite).

In the following three sections we have listed results for backpropagation (our bp) and evolutionary algorithm (our ea) training methods and also results from a naive Bayesian (NB) approach [1].

7.1 $\lambda = 1$

A value of λ of 1 (indicating that a spam taken to be a legitimate mail is exactly as bad as the opposite) could be the case when the filter never tries to avoid spams from arriving in your in box, but simply makes an indication of whether or not the arrived mail is (believed to be) a spam or not. In this case we obtain the following results.

	SR	SP	WA	TCR
our bp	0.871	0.975	0.975	6.617
our ea	0.805	0.990	0.966	4.908
NB	0.823	0.990	0.969	5.41

Conclusions on these results will mainly be in 9.

7.2 $\lambda = 9$

A value of λ of 9 (indicating that a spam taken to be a legitimate mail is 9 times more acceptable than the opposite) could be an appropriate value if you decide, as an example, to somewhat hide the identified spam messages from the user or if the filter would return the identified spam message to the sender, asking the sender for some sort of resend of the mail.

	SR	SP	WA	TCR
our bp	0.826	0.982	0.993	3.270
our ea	0.703	0.992	0.993	2.910
NB	0.776	0.995	0.994	3.82

We have concluded on these results below as well as in section 9.

7.3 $\lambda = 999$

A value of λ of 999 (indicating that false positives are not accepted) could be the case when the filter simply, without further notice, deletes any message identified as spam.

	SR	SP	WA	TCR
our bp	0.01580	1	0.99980	1.01
our ea	0.465	1	1.000	1.870
NB	0.637	1	0.999	2.86

The reason that the results using the backpropagation training method is so poor is that the punishment is “too” severe. This forces the network to only pay attention to the real mails and the correct classification of these, and thus, in effect it classifies all the spam incorrectly yielding a very low TCR. It should though be possible, through hard and long training, to lower the number of incorrect classified spams without heightening the same for legitimate mails. The evolutionary algorithm succeeded somewhat more in this case, but suffers likewise.

8 Applications

The current state of our mail filtering software is that it is not, in the present form, suitable as a filter for a given mail program. There are several obstacles to overcome and other practical things to consider and (re)implement in order to make a filter that could be used by the common user.

Among these there is the form of the frequency table(as described in the section 4) filter used in our "Bayesian" approach. The filter is in the current form something that is generated once and for all and thus cannot be altered or "trained" on incoming mails.

So one would have to implement this table so it can be easily stored on disk, quickly updated and changed, reflecting the frequency changes generated from incoming mails. This would also require the user to distinguish between "deleting" spams and legitimate mails.

When this is done it should also be considered whether these frequency tables should be personal or if there should be a (large) table for an entire organisation or perhaps both.

It should also be considered to do some pruning on these tables so that they would not keep on growing in size. If they were allowed to do so, they would be an obvious target for a denial of service attack. One method of pruning could be to remove entries in the table that has not been updated for a long period of time. It should be safe to remove such entries, since no mail has in a long time contained words that updated these entries and thus are not relevant for sorting presently incoming mails.

Perhaps it would be an advantage to set a fixed upper limit on the number of entries and throw the oldest ones away.

One could also set a limit for the length of a word, say W, thus making sure that the tables would maximally contain $O(\#letters^W)$ entries.

Furthermore the neural network should be automatically trained according to either the personal database or the organisation database.

All of the above (and probably many other issues) must be considered and dealt with in order to make a program or plug-in that is suitable for a mail reader.

9 Conclusions

This project clearly states that it is feasible using neural nets for identifying spam. Our results are not completely as good as the ones in [1] except for the case $\lambda = 1$, but still they are close. The reason for this is probably due to the limited number of attributes used, we have around 40, where [1] has somewhere between 100 and 300, always using the ones that gave the best results. Another thing that might cause the difference in results (as a consequence of λ) might be that the parameters for the backpropagation were mainly adjusted to our homemade corpus (which also influenced all our selected features) and therefore not so appropriate for the Ling-spam corpus. The self-adaptive evolutionary algorithm also gave more promising (and even better than the backpropagation) results on our own corpus, but it seems that there is some times when it also ends up in a local optimum (just as the backpropagation).

A nice thing about using a neural network is that you can feed it most things, for instance the length of the longest capitalised word, the number of links, and so on, but of course some of these are more informative than others.

Another nice thing about this approach is that it (when implemented properly) would adapt to the never ending flow of mails arriving and the language within them.

What would be nice to spent more time on is getting a better trained network, evolving the backpropagation, the ea and a combination hereof. For instance one could use the ea for finding a good initial net for the backpropagation algorithm. Moreover one should look through more mails and try to identify more important features, since a general observation from the early phases of the project was that the more (relevant) inputs we had the better the filter would perform. One could also use something more general than just single words for the Bayesian approach, for instance two or three word tuples.

Altogether it seems as if this idea on recognising spam could be a fruitful way of continuing the fight against spam.

References

[1] I. Androutsopoulos, G. Paliouras, V. Karkaletsis, G. Sakkis, C. Spyropoulos, and P. Stamatopoulos. Learning to filter spam e-mail: A comparison of a naive bayesian and a memory-based approach.

[2] M. Mangalindan. For bulk e-mailer, pestering millions offers path to profit. 2002.

[3] R. Thomsen and M. Christensen. Dendrite - a c++ neural network framework: A quick introduction.

Comparing mass extinction in evolutionary computation

Mads Østerby Jespersen, Hien Phan, and Hanne Steenberg Rasmussen

Abstract— **This paper describes the performance of mass extinction in evolutionary algorithms. Four different approaches are compared. These are Random immigrants, Fogel's mass extinction algorithm, the Self Organised Criticality algorithm and finally our own extinction operator, Strategic Mass Extinction. We explore both the effect of mass extinction on computationally hard and easy problems, in order to draw conclusions on the settings in which the different mass extinction approaches contribute in evolving good solutions, and the ones where it obstructs the process.**

Our results show, that the use of a strategy to decide when an extinction should occur is overall better than the used of randomly applied extinctions as used in the three earlier approaches.

1 Introduction

The idea of introducing mass extinction in evolutionary algorithms is very much inspired by the research of evolutionary biology. This research has shown that the evolution of animals and plants tend to happen in bursts, followed by long periods of stagnation. It has been suggested that these burst are the results of natural disasters which cause mass extinction of species and give room for new varieties to evolve

Modelling mass extinction in evolutionary algorithms has in some cases been shown to be able to prevent premature convergence. But as far as we know, the performance of these algorithms has yet not been compared to one another. We have also tuned all parameters to get a picture of the full potentials of the algorithms. Some of these parameters has not been tuned in earlier published articles.

"Random Immigrants"[1] (RI) is by far the simplest mass extinction technique of the ones tested. In each generation a fixed percentage of the population is killed and replaced by randomly generated individuals. The individuals to be extinct are randomly selected.

Fogel's mass extinction algorithm[2] (Fogel) is based more closely on observations in biology. Every individual is given a fitness, which is relative to the whole population. In each extinction, the population is subjected to a stress-factor, and only those individuals whose fitness is equal to or greater than the stress factor survive. When all survive, a small "background" mutation of the weakest individual as well as 5 randomly chosen individuals occur.

The Self Organised Criticality algorithm[3][4] (SOC) is inspired by a powerlaw distribution of mass extinctions, which is indicated by the fossil record. In the algorithm the frequency and size of the extinctions are thus distributed by the powerlaw, and the individuals destined for extinction are randomly selected and replaced by mutations of individuals in the surviving population.

The fourth and last mass extinction algorithm (SME) is our own proposal. Our idea is based on observations from the results we got from the other algorithms. We discovered that letting extinctions happen randomly is not the best approach. When no stagnation occurs, extinctions only slow down the process of evolving good solutions. And when an extinction is needed, it may not occur or may not be large enough to prevent premature convergence. The SME monitors the best individual of each generation, and when a stagnation seems to occur, a fixed size mass extinction is triggered.

As we will document in this paper, the SME appears to be overall better than both a standard evolutionary algorithm and the other extinction based algorithms.

2 Strategic mass extinction

Based on our observations on how the other approaches managed to evolve solutions, we came up with an idea of trying a strategy based EA using mass extinction. Below a description of how we thought the algorithm should work and the pseudo code for it will be presented.

The SME tries to perdict if a stagnation has occoured and then responds with a mass extinction in order to get new genes into the population and then it ensures room for evolving new potential good individuals after the extinction by preventing another one in the next n generations.

The function StagnationOccured() compares the fitness of the best individual in P(t) and in P(t-1) and if the fitness has not increased by at least a certain procentage, the population has not made satisfactionary progress and will be punished.

If it is decided that a stagnation has indeed occured, a high procentage of the population is replaced by new randomly generated individuals else mutation occours.

The pseudo-code can be seen in fig. 1.

The reason for waiting some n generations after each extinction before allowing a new extinction, is to insure new individuals time to evolve, before their quality can be estimated. Their fitness in the first generations after the extinction does not say anything useful about their potential.

Extinction do not only happen, when no improvement occur. It also occurs, when the improvement is considered too small. The reason for this is that premature convergence does not necessariy imply, that a local optimum has been found. It is enough, that the computation converges toward a local optimum. In this case a slight improvement in best fitness can be observed in almost every generation for a long time. However, this does not mean that the algorithm is headed for the global optimum, and the use of extinction might be useful in interrupting this potentially premature convergence. The experiments have to some extent confirmed that this is often the case.

procedure *Strategic mass extinction*
begin
  t:=0;
  wait:=0;
  totalwait:=tw;
  killrate:=k;
  minProgress:=m;
  initialize P(t);
  evaluate structures in P(t);
  while (termination condition not satisfied)
  begin
    t:=t+1;
    wait:=wait+1;
    crossover P(t);
    if (*(wait* $\geqq$ *tw) and stagnationOccurred()*)
    then
      kill k percent of structures in P(t);
      replace dead structures;
      wait:=0;
    else
      mutate structures in P(t);
    select structures in P(t) from P(t-1)
    evaluate structures in P(t);
  end
end

Figure 1: Pseudo code for the MSE algotihm.

3 Experiments

3.1 Test Functions

In our experiments we used the following five test functions:

De Jong F4:
$f(\bar{x}) = \sum_{i=1}^{30} x_i^4$, where $-1.28 \leq x_i \leq 1.28$

Ackley F1 (100 dimensions):
$f(x) = 20+e-20\exp\left(-0.2\sqrt{\frac{1}{100}\sum_{i=1}^{100} x_i^2}\right) - \exp\left(\frac{1}{100}\sum_{i=1}^{100} cos(2\pi \cdot x_i)\right)$, where $-30 \leq x_i \leq 30$

Rosenbrock F1 (100 dimensions):
$f(x) = \sum_{i=1}^{100}(100 * (x_i - x_{i-1}^2)^2 + (x_{i-1} - 1)^2)$ where $-100 \leq x_i \leq 100$

Griewank F1 (100 dimensions):
$f(\bar{x}) = \frac{1}{4000} * \sum_{i=1}^{100}(x_i - 100)^2 - (\prod_{i=1}^{100} \cos(\frac{x_i - 100}{\sqrt{i+1}})) + 1$, where $-600 \leq x_i \leq 600$

Rastrigin F1 (20 dimensions):
$f(\overline{x}) = 200 + \sum_{i=1}^{20} x_i^2 - 10 \cdot cos(2\pi x_i)$, where $-5.12 \leq x_i \leq 5.12$

The problems above are minimization problems and 0 is the global minimum for all of them.

3.2 Settings for the algoritms

The implementation is based on the publicly available EvALife java-library[3], and our algorithms will be included in a future release.

In all the algorithms elitism and an annealing mutation parameter is used $(1/(1 + generation))$. Tournament selection is used as selection method.
We tuned mutation- and crossover parameters in all algorithms in the range [0.1;0.9] with stepsize 0.1.

To compare the algorithms with different parameters we used the average of 50, with 200.000 evaluations, runs to find the best.

The tables below presents the settings we found optimal for each EA.

Standard EA settings

	pc	pm
Ackley	0.9	0.9
DeJong	0.9	0.9
Griewank	0.8	0.9
Rastrigin	0.6	0.9
Rosenbrock	0.9	0.9

Table 1: Optimal parameters found for the std. EA.

Fogel settings

$\alpha = 0.4$.
α was tested in the interval [0.3;0.5] with stepsize 0.1

	pm
Ackley	0.1
DeJong	0.5
Griewank	0.7
Rastrigin	0.3
Rosenbrock	0.3

Table 2: Optimal parameters found for Fogel.

	pc	pm
Ackley	0.9	0.8
DeJong	0.8	0.9
Griewank	0.9	0.9
Rastrigin	0.9	0.6
Rosenbrock	0.9	0.9

Table 3: Optimal parameters found for the SME.

SME settings

Wait = 50.
KillRate = 0.9
minProgress = 0.02

Wait was tested in the interval [30;130] with stepsize 10, and KillRate was tested in the interval [50;90] $\cup\{96\}$ with a stepsize of 10. minProgress was tested inn the interval [0.01;0.10] with a stepsize of 0.01

SOC settings

	pc	pm	Sandpile
Ackley	0.9	0.7	7x7
DeJong	0.9	0.9	18x18
Griewank	0.9	0.8	15x15
Rastrigin	0.9	0.7	11x11
Rosenbrock	0.9	0.9	30x30

Table 4: Optimal parameters found for SOC.

SandpileSize was tested with sizes of: 5x5, 7x7, 9x9, 11x11, 15x15, 18x18, 30x30 in order to see if how it effected SOC.

RI settings

ReplacementRate = 0.3

[3]www.evalife.dk/applications.php

	pc	pm
Ackley	0.8	0.9
DeJong	0.8	0.9
Griewank	0.8	0.9
Rastrigin	0.9	0.8
Rosenbrock	0.9	0.9

Table 5: Optimal parameters found for RI.

ReplacementRate was tested in the interval [0.1;0.5] with stepsize 0.1

4 Results

On the graphs on the following pages, we have chosen not to show the performance of the Fogel algorithm, as it was very poor compared to the performance of the other algorithms.

Each algortihm was limited to use 200.000 evaluations in order to make a fair comparison to each EA. This is reflected in the numbers of generations each EA have.

4.1 Ackley F1 100D

In the Ackley F1 problem, the SME seems to be the best. It reaches approximately the same fitness as the RI, but has a faster convergence. Both the standard EA and the SOC display premature convergence to a local optima.

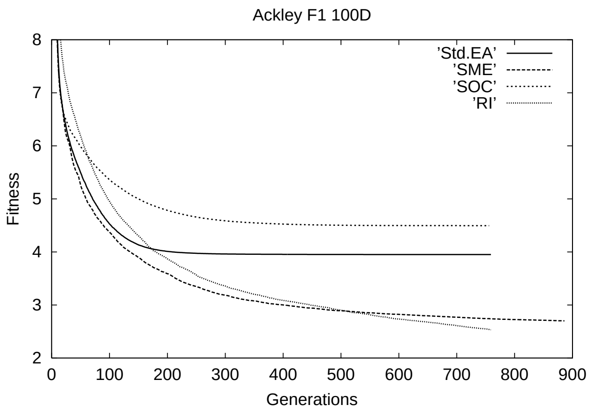

Figure 2: Average best fitness (50 runs)

Algorithm	Avg. best fit.	Std. error
Standard	3.8734	0.0990
RI	2.4804	0.0544
SME	2.6489	0.0660
SOC	4.4075	0.1053
Fogel	18.6556	0.3733

Table 6: Best fitness and standard devation.

4.2 DeJong F4

The DeJong F4 is not a hard problem, and the use of extinction should not be needed, but the results do show something interesting: The use of extinction slows down the convergence of the solutions towards the global optimum. It seems that the SME does the least harm, even though, the performance of it is very close to the one of the standard EA. Fogel's algorithm is the only one that end up far off the optimum.

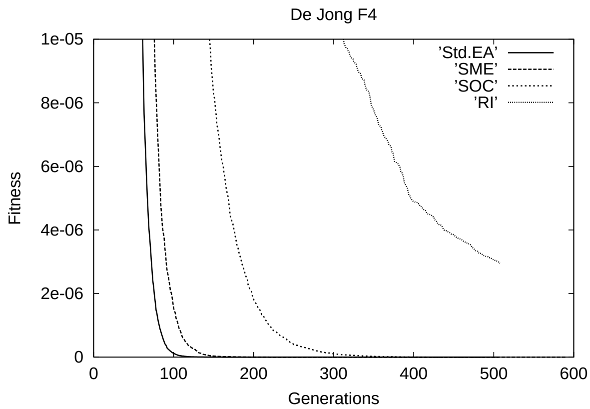

Figure 3: Average best fitness (50 runs)

Algorithm	Avg. best fit.	Std. error
Standard	$3.4113e-12$	$2.3743e-13$
RI	$2.8933e-06$	$2.6922e-07$
SME	$1.3627e-11$	$1.0355e-12$
SOC	$1.4876e-11$	$6.3507e-12$
Fogel	9.7226	0.5150

Table 7: Best fitness and standard devation.

4.3 Griewank F1

The standard EA does very well on the Griewank problem. No stagnation or premature convergence seem to occur, and futher experiments have shown that the algorithm given more time, does find the global optimum. All of the other algorithms have worse performances. The SME does fairly well, followed by the RI, the SOC and finally Fogel.

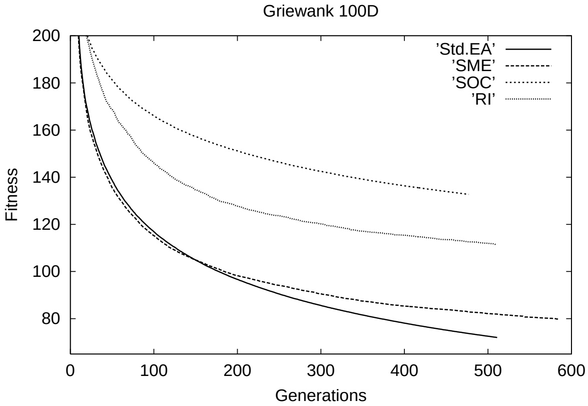

Figure 4: Average best fitness (50 runs)

Algorithm	Avg. best fit.	Std error
Standard	70.6278	1.8840
RI	109.4758	2.2833
SME	78.2786	1.7745
SOC	130.0254	4.4510
Fogel	1568.1473	33.7378

Table 8: Best fitness and standard devation.

4.4 Rastrigin F1 20D

Rastrigin F1 is a very hard problem to solve for a standard EA. Here all the extinction algorithms, except for Fogel's EA, do a better job of finding a good solution. The best one is found by the RI, which converges very fast. The second best solution is found by SME, but this algorithm has a much slower convergence, The fast converging SOC does nearly as well as SME and much better than the Fogel.

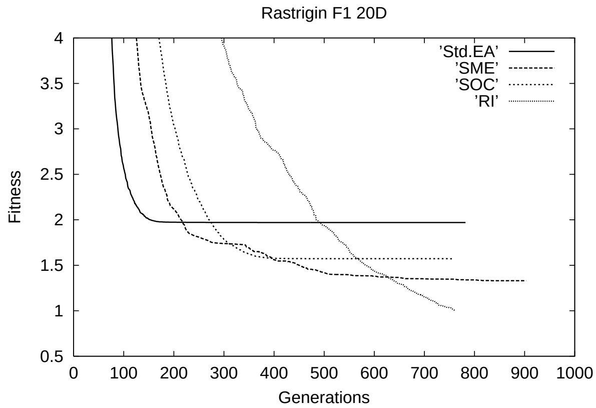

Figure 5: Average best fitness (50 runs)

Algorithm	Avg. best fit.	Std. error
Standard	1.8926	0.1678
RI	0.9794	0.05397
SME	1.3038	0.17659
SOC	1.5031	0.1918
Fogel	98.6568	3.6163

Table 9: Best fitness and standard devation.

4.5 Rosenbrock 100D

The SME and the standard EA both do well on the Rosenbrock problem, but the SME has a faster convergence. The RI algorithm seems to be stagnating, while the SOC is still finding better solutions.

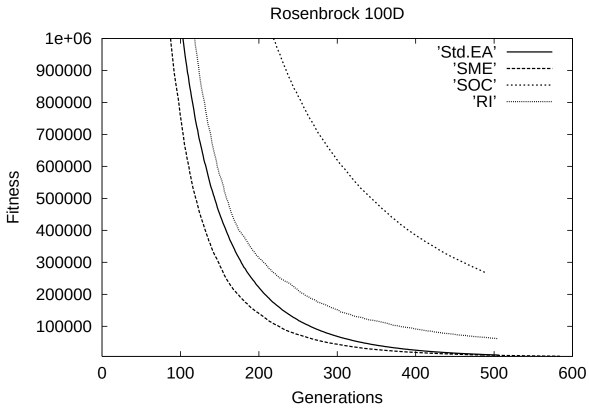

Figure 6: Average best fitness (50 runs)

Algorithm	Avg. best fit.	Std. error
Standard	$9.9828e3$	$3.3717e3$
RI	$6.0282e4$	$2.5688e3$
SME	$6.3677e3$	$5.116e2$
SOC	$2.6096e5$	$2.6482e4$
Fogel	$5.7855e10$	$1.4788e9$

Table 10: Best fitness and standard devation.

5 Discussion

In this paper, we have examined the use of different kinds of mass extinction. The best choice of algorithm is of course to some extent problem dependent, but it appears as if the two algorithms RI and SME generally outperform the other two extinction based algorithms. It is clearly an advantage to use mass extinction on hard problems. The solutions are often distinctly better than the ones produced by the standard EA.

It is difficult to say whether SME or RI is the best algorithm. SME has the advantage of practically never using extinction when the algorithms does not need it. Thus the evolution of solutions is not disturbed unnecessarily and this in some cases improves performance. Also the SME is superior regarding convergence speed.

On the other hand the RI on occation does get closer to the optimum and is in addition to this capable of detecting changes in a dynamic search space. If a new optimum appears during the run of the SME may not discover it. This is where the RI would be expected to excel in particular.

Extinctions happen every generation and kill randomly chosen individuals in every generations in the RI algorithm, but the size of the extinctions is so small that apparently it is too small to prevent at least some of the new immigrants from evolving into good solutions. Though on some problems, the extinction is either insufficient of downright harmful.

The performance of Fogel and SOC was inferior to the performance of RI and SME. Perhaps one conclusion to be made is that large extinctions should be used with caution. In our experiments we discovered that they, if they happen too quickly after a previous mass extinction, might kill promising individuals who have not had the time to develop. And if large scale extinctions happen when they are not needed, they can disrupt a healthy convergence. Elitism reduces the damage done, but does not prevent it all together. In neither Fogel nor SOC is there any guarantee that a series of massive extinctions do not happen in two consequtive generations, although it is unlikely in SOC.

In addition to not happening too often, extinctions should happen when they are needed. There seems to be no advantage in waiting and this will often happen, when randomly occuring extinctions are used.

We believe that the usefulness of implementing biological theories is limited when it comes to mass extinction. It is true, that extinctions seem to be one of the main driving forces of evolution in nature. However, it is also the case that long periods without environmental changes but with persistace of stable species can be observed in the fossil record, and surely there is no need to recreate those in evolutionary algorithms. This is to some degree done in both Fogel and SOC. It also seems reasonable to assume, that randomly killing off quickly evolving individuals in a large scale hardly is a good idea, even though it can be observed in nature. This is almost bound to happen frequently in both the SOC and Fogel, even though the algorithms make an effort to always kill the least fit part of the population.

One of the main results of this paper probably is, that extinctions that happend randomly and killing off a randomly chosen number of species does not seem to be the most promising way to go about designing extinction based evolutionary algorithms. The frequency and size of extinctions need to be controlled very carefully.

When this is said, we believe that there also are other reasons for the poor performance of Fogel. Perhaps there is a not enough exploration taking place due to the lack of crossover. This could be compensated for by perfoming larger mutations. The use of larger and slower annealing mutation variance would perhaps give a more fair picture of the performance of Fogel.

It has been difficult to find good test problems for the experiments. Most problems are not hard enough and does not make the standard EA stagnate at local optimums as it is often seen in "real life problems". It could be very interesting in the future to look into applying extinction on some of these problems.

Another possiblity for future research could be to implement a combination of the SME and RI to exploit the advantages of both algorithms.

References

[1] Cobb, H. G. and Grefenstette, J. F. Genetic algorithms for tracking changing enviroments. In Proceeding og the 5th International Conference on Genetic Algortihms (San Mateo, CA, USA, July 1993), S.Forest, Ed., Morgan Kaufmann, pp. 523-530.

[2] Fogel, G.B., Greenwood, G.W., and Chellapilla, K. Evolutionary computation with extinction: Experiments and analysis. In Proc. of the 2000 Congress on Evolutionary Computation. (Piscataway, NJ, 2000), IEEEE Service CEnter, pp. 1415-1420.

[3] Krink, T., Rickers, P and Thomsen, R. Applying self-organised critecality to evolutionary algorithms. In parallel problem solving from nature - PPSN VI (Berlin 2000), M. Schoenauer, K. Deb, G. Rudolph, X., Yao, E., Lutton, J. J. Merelo and H.P. Schwefel, Eds., Springer, pp.375-384.

[4] Krink, T. and Thomsen, R. (2001). Self-Organized Criticality and Mass Extinction in Evolutionary Algorithms. In: Proceedings of the Third Congress on Evolutionary Computation (CEC-2001), vol. 2, p.1155-1161

Evolving Evolutionary Algorithms

Christopher Mosses and Casper Lund Thomsen

Abstract—**This paper describes a method for evolving parameters of Evolutionary Algorithms by the use of Evolutionary Algorithms. The method involves creating a meta EA which has a population of standard EAs. The standard EAs act as individuals with fitnesses, genomes etc. As a demonstration of the usefulness of this approach, the function for simulated annealing in a standard EA has been parameterized and evolved to develop a better performing function.**

1 Introduction

This is an attempt to use the benefits of an EA on itself. When implementing an EA to work on a problem, there is invariably a set of parameters which must be chosen. The choice is crucial. One set may yield very different results from another. Traditionally the selection of parameters has been based to some extent on intuition, heuristics and hand tuning. Needless to say, the process is often inaccurate, cumbersome and hard to figure out. Additionally, it is hard to back up a claim that the selection is really the best suited.

Our approach attempts to solve some of these problems. The main idea is to create a meta EA (MetaEA) which evolves a population of standard EAs. This automates the hand tuning of parameters, and gives credibility to the final choice: it is the result of a run of an EA. The MetaEA is simple to implement. It is similar to a standard EA, so it is easy to try it out on ones favorite EA.

2 The MetaEA

The MetaEA is similar to a regular EA. It has the expected methods for evolving a population of individuals:

- `mutate`: Mutates the population, using mutation rate parameter p_m. We use the standard Gaussian mutation operator.
- `recombine`: Recombines the population, using crossover rate parameter p_c. We use standard arithmetic crossover.
- `select`: Performs standard tournament selection.

The twist is that the individuals are standard EAs with a few modifications. An EA which is to act as an individual must have the usual properties of an individual, with the following straight-forward interpretations:

- `genome`: The parameters of the EA, such as p_m, p_c, population size etc.
- `fitness`: A measure of how well an algorithm performs with the current genome (parameter set).

The structure of the MetaEA can be described as a hierarchy:

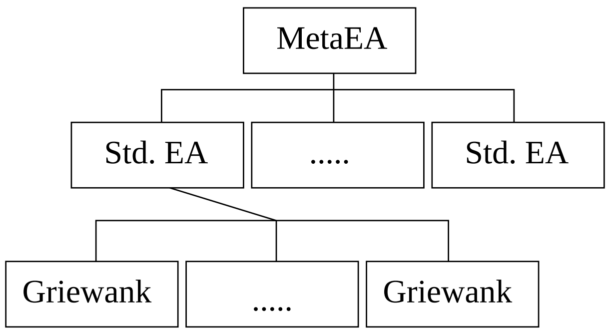

Figure 1: Structure of the MetaEA

In this case, the standard EAs optimize the Griewank benchmark problem, but the problem could be substituted for any other. A list of benchmark problems used in this project is contained in Table 1.

For each generation of the MetaEA, the usual actions of a standard EA are performed: mutatation and recombination of the genomes. This causes the parameters of the standard EAs to change. Then the tournament selection process is carried out. To decide which of two competing EAs is to win, their fitnesses must be compared. The fitness function for an EA does a full run of the EA with the current parameters (retrieved from the genome), and returns the best fitness achieved during the run.

3 Optimized Annealing

To try out the MetaEA approach in practice, we attempted to evolve a good function for simulated annealing. It has been our experience that the choice of annealing function is rather arbitrary. Predominant choices are $\alpha = \frac{1}{1+t}$ or $\alpha = \frac{1}{1+\sqrt{t}}$.

The annealing function is critical for an EA because it must ensure that the search space is explored thoroughly to start with, and it must allow fine tuning towards the end. If it is chosen incorrectly, the algorithm may stagnate early on, or it may miss the global optimum because parts of the search space are ignored. The functions mentioned above give fair results on benchmark problems, but they appear worryingly simple.

To create a new, hopefully better annealing function, we parameterized the functions: $\alpha = \frac{a}{(b+ct)^d}$. The parameters a, b, c and d then constituted the genome of each EA individual. The EAs were set to evaluate a set of standard benchmark problems.

Once the MetaEA yielded results for a, b, c and d, we plugged the values into a standard EA, and ran the benchmarks against other annealing functions.

4 Results

We did the following:

1. We ran the MetaEA on each of the problems. It seemed to converge quickly. Below in Table 1 we display the resulting values of a, b, c and d.
2. We ran the standard EA on each problem, and with various annealing functions, including our own optimized one. The graphs below show a comparison of them, and Table 2 shows the values.

The test setup was the following:

- $p_m = 0.75$
- $p_c = 0.9$
- Size of population of standard EA's for a MetaEA: 20
- Size of population for a standard EA: 100
- Number of generations (MetaEA): 10^4
- Number of generations (BasicEA): 50 times the dimension of the problem

Table 1 shows values for a, b, c and d achieved by running the MetaEA. We noticed during the runs that a, b and c were somewhat dependent on each other, while d was much more stable. Perhaps this is a sign that they are not very crucial for the overall behaviour. It seems that d hovers around 0.5, which is the square root, but it varies according to which problem is being run. There is one fly in the ointment: SchaefferF6. Here we seemed to get evolve a sort of reverse annealing, in that the resulting function is increasing.

Problem	a	b	c	d
AckleyF1 20D	346.3	237.9	425.5	0.5019
AckleyF1 50D	245.9	426.0	492.5	0.6074
AckleyF1 100D	726.8	345.9	677.1	0.6698
DeJongF4 30D	38.3	184.9	545.1	0.7826
Griew. F1 20D	168.2	615.8	666.2	0.3669
Griew. F1 50D	703.6	148.4	727.0	0.4831
Griew. F1 100D	363.9	862.5	379.5	0.4473
Rastr. F1 20D	724.6	555.9	241.6	0.7223
Rastr. F1 50D	201.5	252.8	316.9	0.7744
Rastr. F1 100D	471.3	477.0	354.2	0.9793
Rosenb. F1 20D	709.2	92.3	484.7	0.6163
Rosenb. F1 50D	712.6	815.2	653.1	0.5839
Rosenb. F1 100D	745.8	713.0	884.1	0.5887
Schaef. F6 20D	405.2	521.9	671.9	-2.534
Ursem8 20D	593.1	621.7	331.4	0.7851

Table 1: Parameters for $\alpha = \frac{a}{(b+ct)^d}$.

Table 2 shows the results of running the standard EA on each of the benchmark problems, with each of the annealing functions. We did 20 runs with each configuration and recorded

[4] This may seem low, but the parameters converged rather quickly.

the average best fitness. In many cases the EA using the optimized annealing function outperformed the others by a healthy margin. In other cases it merely matched them, and in a few it was beaten, but not by much. We were happy with the results, because they seem to indicate that using an optimized annealing function yields good improvements.

Problem	$\frac{1}{1+t}$	$\frac{1}{1+\sqrt{t}}$	$\frac{a}{(b+ct)^d}$
AckleyF1 20D	0.1832	0.0010	0.0282
AckleyF1 50D	0.2452	0.1007	0.0020
AckleyF1 100D	0.2409	0.0938	0.0015
DeJongF4 30D	0.0010	1.71E-9	2.1E-15
GriewankF1 20D	0.9197	1.9632	0.0022
GriewankF1 50D	6.2379	5.0405	0.0018
GriewankF1 100D	13.754	11.782	0.0029
RastriginF1 20D	6.7194	0.3502	0.5277
RastriginF1 50D	18.967	1.4982	1.6921
RastriginF1 100D	39.958	2.1496	3.6813
RosenbrockF1 20D	1.95E6	49.4	1.0506
RosenbrockF1 50D	5.06E6	1045.2	7.5876
RosenbrockF1 100D	1.00E7	56619.6	5.1622
SchaefferF6 20D	0.0018	4.85E-4	5.2E-4
UrsemF8 20D	5.2825	32.0787	32.117

Table 2: Comparison of standard EAs with different annealing functions. Values are average best fitness over 20 runs. UrsemF8 is a maximization problem.

As can be seen from the following graphs, we observed several pleasant features of using our optimized function. In many cases[5] we saw a faster convergence, but without the premature stagnation which often comes as a price for it. In addition, we could see that EAs using our annealing tended to continue improving long after the others stagnated. This is an especially valuable trait, because it means one can let the EA run for a longer time and get results that improve correspondingly. Finally, it seems that the higher the dimension, the better we perform compared to the others.

The result of running AckleyF1 50D with the evolved annealing is seen in figure 2. Comparing the three graphs, one can see that the two first converge quickly and stagnate quickly, while EvoEvo converges a bit slower but doesn't stagnate as early.

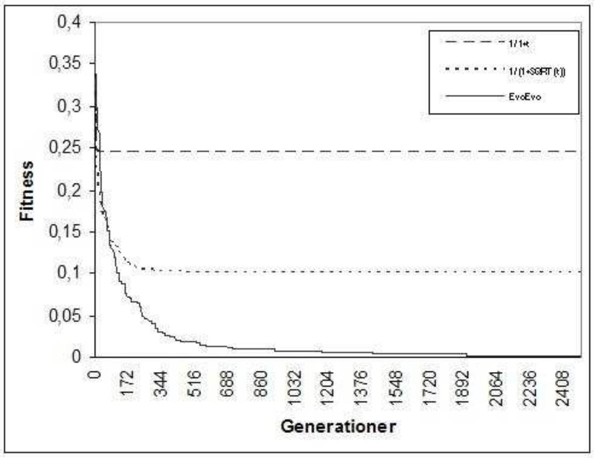

Figure 2: AckleyF1 50D

GriewankF1 for dimensions 20D, 50D and 100D is presented in the figures 3, 4 and 5, respectively. Using $\frac{1}{1+t}$ one stagnates very quickly. This stagnation isn't present when using $\frac{1}{1+\sqrt{t}}$. EvoEvo improves upon $\frac{1}{1+\sqrt{t}}$ with faster convergence and a continued healthy lack of stagnation.

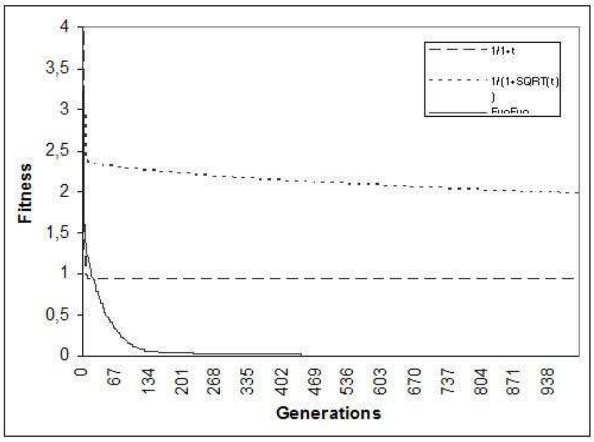

Figure 3: GriewankF1 20D

RosenbrockF1 (figure 6) was the function which differed the most when using the EvoEvo function. There was a factor 10.000.000 between $\frac{1}{1+t}$ and EvoEvo. It is another exam-

[5] We have only shown a few of the problems, the more interesting ones. SchaefferF6, DeJongF4 and RastriginF1 didn't experince any great advantages of using the evolved function.

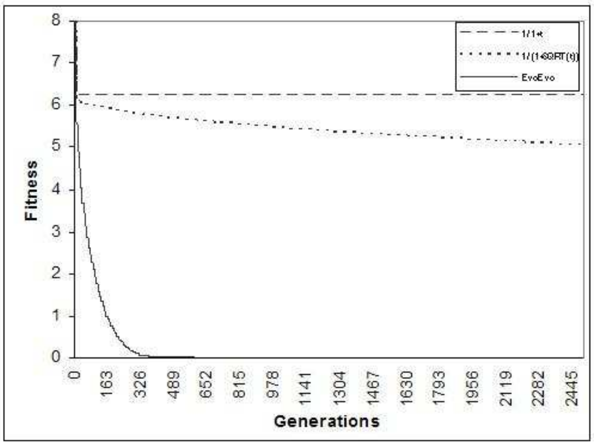

Figure 4: GriewankF1 50D

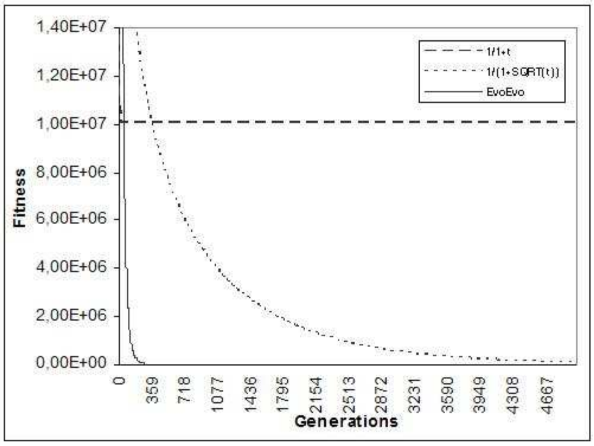

Figure 6: RosenbrockF1 100D

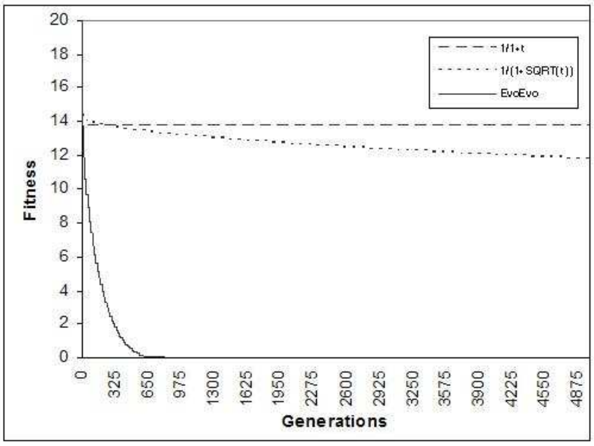

Figure 5: GriewankF1 100D

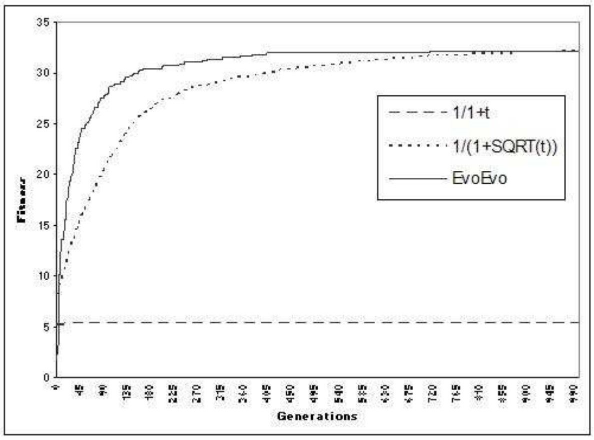

Figure 7: UrsemmultimodalF8

ple of the fast convergence of EvoEvo. $\frac{1}{1+\sqrt{t}}$ eventually starts catching up to EvoEvo, but only very late in the run, and even then it is several orders of magnitude away, as seen in Table 2.

Figure 7 shows the graph for UrsemMultimodalF8 20D. EvoEvo outperforms $\frac{1}{1+t}$ and the only difference between $\frac{1}{1+\sqrt{t}}$ and EvoEvo is the speed of convergence.

5 Future Work

The experiments are promising, and there are many other areas which might benefit from similar treatment. A shortcoming of our parameterization of the annealing function is that it has a rather fixed form. Instead, the function could be generalized further, and different classes of functions could be used. For instance, there might be something to gain from using periodic functions. The function class could be encoded as part of the genome, to be mutated like the others.

Naturally any other parameter can also be evolved, such as population size, the selection operator, the mutation operator etc. For the mutation operator, for instance, it may be possible to find a distribution which is better suited than the Gaussian distribution. We feel that in many cases it is possible to simply parameterize and evolve a standard operator. The operator becomes slightly more complex, but most likely this will have only a negligible effect on performance, compared to evaluating the fitness function.

So far we have managed to find good functions for several problems, but they differ somewhat. An interesting aspect could be to find a generally applicable annealing function as an alternative to the predominant annealing functions currently used in the business.

6 Conclusions

Running time is an issue with a MetaEA. Since it evaluates the population of standard EAs many times, there is a lot of computation going on. For benchmark problems such as the ones tested, this is noticeable. For real-world problems it becomes critical. It simply requires too much time to evaluate the fitness functions so many times. However, that is not the point of the MetaEA approach. The point is rather to evolve EA parameters for a group of similar EAs, which can be expected to behave in the same way. Once the parameters have been found, they can be plugged in and used over and over. In a case where the same EA runs many times on slightly different data, it would also be useful to make the one-time effort of finding good parameters. From the preliminary results obtained, it would seem that there is ample reason to investigate the MetaEA approach further.

A shortcoming is that the evolved parameters differ in quality. Some problems have substantial gains from using the MetaEA while others gain little or nothing. This makes it difficult to give guidelines for when to use the MetaEA and when not to use it. However, we did see that using it never meant a serious deterioration of performance, while sometimes the improvements could be dramatic. This seems to make it a safe bet to try out the MetaEA on new problems.

Simulated Soccer by Neural Networks

Jakob Langdal, Jakob Johannsen, and Ronni Laursen

Abstract—**Soccer is a simple, yet interesting game. Thus, trying to make computers play some kind of soccer simulation game presents an interesting and fun challenge. In this report, we present an evolutionary approach to teaching neural networks to play a simplified soccer simulation game.**

1 Introduction

Apart from the fact that it is fun, the game of soccer possesses many common and interesting game features. These features include strategic positioning of players, cooperation between players, and the combined goals of trying to score as many goals as possible, while at the same time trying to prevent the opponent from scoring.

Many computer soccer simulators have been developed, both for purposes of research as well as for pure entertainment.

We have implemented a simplified model with the purpose of teaching a neural network to play soccer.

2 Representation

Our soccer game is played on a field of NxM squares, each of which can hold an infinite number of players from each team, as well as zero or more balls, depending on how many balls the game is played with. In our runs, we have used N=5, M=4, two players on each team, and one ball.

The game is turn based, and for each turn, each player is allowed one action. The players choose their actions in random order. The following actions are allowed:

- Move
  A move is always allowed. A player can move to any adjacent square, provided that he stays on the field. That is, if the player is standing in square (x, y), he can move to any point (x', y') on the field where $x-1 \leq x' \leq x+1$ and $y-1 \leq y' \leq y+1$.
- Move with ball
  Similar to Move, except that the ball is moved with the player. If a player takes a "Move with ball" action while the ball is not in the same square as the player, only the player is moved.
- Kick
  Passing the ball. A pass can be made to any square on the field within a two square radius of the player, provided that the ball is in the same square as the player. That is, a pass from (x, y) to (x', y') is allowed if $x-2 \leq x' \leq x+2$ and $y-2 \leq y' \leq y+2$, (x', y') is on the field, and the ball's coordinates are (x, y). An illegal Kick action results in no action.
- Shoot at goal
  Trying to score. A shot is allowed only if the ball is in the same square as the player, and if the player is sufficiently close to the goal. "Sufficiently close to the goal" was during our runs defined to be the two squares directly in front of the goal, i.e. coordinates $(0, 1)$ and $(0, 2)$ for $team1$, and $(4, 1)$ and $(4, 2)$ for team 2. An illegal "Shoot at goal" action results in no action.

Note that if the ball is moved by an action, the change is visible immediately, whereas if a player moves, the change is not visible until the end of the turn.

The match ends after a predefined number of turns.

2.1 The Expert System

For the evaluation of the fitness of a particular neural network, we have developed a simple expert system, which we briefly describe in this section.

The expert system uses very simple rules to decide its actions:

If the ball is in the same square as the player:

- If the player is "sufficiently close to the opposing goal":
  Shoot, score and watch the crowd go wild.
- If a friendly player is in a square closer to the opposing goal, and within passing distance:
  Kick the ball to the friendly player.
- Else:
  Move with Ball one square towards the opposing goal.

If the ball is somewhere else:

Always move as directly as possible towards the ball.

3 The Neural Network

In this section we will discuss the structure of our Neural Network. Furthermore we will elaborate on how we represent the field in the network[6].

3.1 Structure

Our Neural Network is composed of 3 layers:

- One input layer with a number of nodes equal to the number of squares in the field times a number of flags representing the state of the square. The number of flags is currently 4.[7]
- One hidden layer with 20 nodes.
- One output layer with 6 nodes where the first 4 represent the possible actions and the last two are interpreted as relative coordinates from the player's position. The coordinates are used to calculate the target square of a "Move", "Move with ball" and "Kick" actions.

[6] Thanks to Thomsen, R. for providing us with a implementation of a easy-to-use Neural Network framework.

[7] See section 3.2

We use a feed-forward neural network.

Of the possible actions the neural network can take, we choose the action with the highest output among the action output nodes, and use the 2 extra output nodes as coordinates for this action.

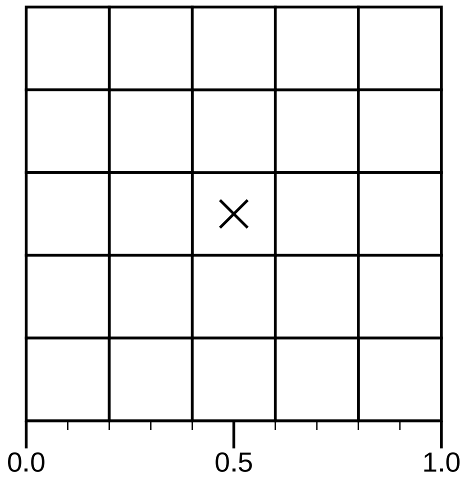

Figure 1: Kick. X is the current player. The ball is in the same square as the player.

For instance, suppose the highest output among the action nodes is found at the "Kick" node, and that the first coordinate node gives output x'. The ball is then kicked to a square with x coordinate as shown in figure 1, and similarly for the y coordinate. For a "Move" or a "Move with ball" action, the output from the coordinate nodes are of course only scaled to match three squares.

3.2 Representation

Our input layer is, as previously mentioned, composed of the number of squares in the field times 4 flags. Before loading the field into the neural network, we change the representation of the field's squares into a quadruple of bits and interpret them as follows:

- If the first bit is set then the ball is at this square.
- If the second bit is set then a player from *team*1 is at this square.
- If the third bit is set then a player from *team*2 is at this square.

- If the fourth bit is set it means that the active player, i.e. the player for which the neural network has to decide an action, is located here.

To avoid confusing the neural network, it always plays *team*2. However, it would be very simple to allow it to also play *team*1, since this only requires a swapping of the second and third bit, and changing the interpretation of the first coordinate output, so that the neural network attacks in the right direction.

4 The Evolutionary Algorithm

In order to train the neural network, we use an EA. Since the neural network uses real-valued weights, and it is these weights we have to improve on, our genomes are simply real-valued vectors. These genomes represent an entire neural network. We get an EA with the following properties.

4.1 Initialisation

Every entry in a genome ranges from -1 to 1, and they are initialised randomly at the beginning of every run.

4.2 Tournament Selection

The selection process is based on a tournament selection between two genomes at a time. These genomes are selected randomly from the population.

4.3 Crossover

Since the genomes are real-valued vectors, we have implemented an arithmetic crossover function. This crossover uses the following formula.

Let x_1 and x_2 be two randomly chosen genomes, then each entry c_i in the offspring is defined as

$$c[i] = wx_1[i] + (1-w)x_2[i]$$

where w is a random number between 0 and 1.

4.4 Gaussian Mutation

To mutate the genomes we have implemented a Gaussian mutation scheme. Every entry in a genome is mutated according to a Gaussian distribution, while assuring that no entry is scaled out of range.

4.5 Annealing scheme

To gradually decrease the mutation rate we have implemented an annealing mutation scheme, so that the variance σ of mutations decreases like this:

$$\sigma = 1/(1+\sqrt{\#generation})$$

4.6 The Fitness-function

Our current fitness function is based on a simple scheme. We take the difference in goals and use this as a fitness for our neural network. To this value is added an award for scoring at all. This scheme should emphasise the neural network's ability for scoring goals.

We have used different fitness schemes due to the fact that a neural network's ability to do well is very much based on the fitness ie. the fitness-function should be developed accordingly.

We have developed a few different fitness-functions but used the one previously mentioned due to it's simplicity and it's ability to do well.

One of our previous fitness schemes also got the difference in goals in fitness. Furthermore the neural network was awarded a constant value for each goal it scored and was penalised for each goal the opponent scored and another, preferred higher, constant value if the game was won and a penalty if the game was lost. As an extra penalty to the fitness we counted the number of illegal actions the neural network took but we also counted the legal actions and gave that a higher value so that the neural network would learn to take correct actions ie. minimising the number of illegal actions.

5 Results (test section)

In this section we describe how we have tested the implementation. We have made test runs for varying parameters for the EA, and with multiple fitness-functions. Furthermore we have tried several neural network configurations, where the size and number of hidden layers was altered. The fitness-values of one of our test-runs is shown in figure 2. It can be seen that the evolution stagnates at a certain point. At this point the neural network beats the expert system 98% of the matches. The reason why it is not winning all matches, is probably due to the randomness in the turn-sequence[8].

5.1 Structural changes

We found that the size of the neural network had little effect on the quality of results. Increasing the structural complexity of the neural network only affected the running-time.

5.2 Parameter tweaking

- We tested the EA with population sizes ranging from 10 to 200 individuals. We found that a population-size of 50 was sufficient to generate reasonable results. Increasing the size had no significant impact, except on running-times.

- Crossover-rates were set to 75% and kept constant throughout all tests. It seemed that *arithmetic* crossover actually improved on the performance of the neural network.

- We experimented with mutation-rates ranging from 10% to 80% and this rate was annealed as explained earlier. It seemed that mutation was almost unnecessary, because of the few local optima in the fitness-landscape.

- The duration of a match was between 100 to 300 turns. This variation had no or little impact, except on running-times.

[8] see section 2

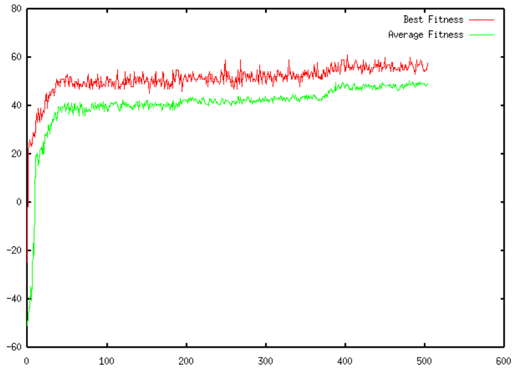

Figure 2: Plot of the fitness-function for a run with 50 individuals and mutation-rate of 10%

6 Problems

In the following section we describe some of the problems we have encountered and some corresponding solutions:

6.1 The 0-0 problem

Due to the way the fitness is given to an individual (ie. the fitness-function), the neural network learns to appreciate a score of 0-0. Actually if the neural network doesn't score a goal in the first x generations the net instead learns to play 0-0 against our expert system (ie. the net doesn't learn to score).

The solution to the 0-0 problem could be solved with a hyper-jump-function implemented in our EA. This would serve to hyper-mutate an individual if stagnation in the population is observed. This function could, of course, also take advantage of an annealing-scheme, only different from the one used in the mutation-function. One could also, as previously mentioned, introduce a penalty to the fitness-value if the scores were equal.

6.2 Expert System

Our implementation of the expert system is rather crude. The idea is much too simple and could easily be extended to react better to some situations. The expert system is deterministic and due to that fact, the neural network could perhaps exploit this weakness and find a winning strategy.

A solution to this problem is mentioned in section 7.2.

7 Future Work

The following section explains some possible extensions to our project:

7.1 Extensions to the Neural Network

The structure of the neural network could easily be incorporated into the EA's mutation-function, so a dynamic structure-change to the net could be developed. The change in topology of the network is a possible advantage and does not require knowledge about the topology from the programmer, because the EA will hopefully choose the best structure.

7.2 Extensions to the game

Instead of training the neural network against a hand-coded expert system, one could introduce a second neural network which then could play against the first net. This has the advantage that a lot of nondeterminism is introduced, hopefully resulting in a well-trained network.

Another possible extension could be to have different fitness-functions for each neural network. The nets could then develop different strategies and hopefully develop into even better decision-makers.

The field's dimensions could be increased along with the number of players. If the field became larger the possible strategies for the neural network would then grow as well. Unfortunately this has a minor drawback. It increases the size of the net substantially and will make the net slower to train. But if one only feeds the network the square, on which the active player stands, and the surrounding squares in a two square-radius, (ie. a moving-window-scheme of size 5*5) the dimension of the field could be of arbitrary size. It would then be sensible to add additional players to each team so that additional strategies could be developed.

8 Conclusions

We managed to train a neural network into a quite sensible soccer player. This neural network outplayed our hand-coded expert-system by far and therefore we achieved our goals.

Measurement-guided EA

Christian Plesner, Jonas Auken, and Laust Rud Jensen

Abstract— **The subject of this paper is monitoring and controlling characteristics in evolutionary algorithms, focusing on one specific characteristic: the diversity.**

1 Introduction

Any EA has a number of parameters that can be adjusted during a run. These usually include mutation rate, p_m, and crossover rate, p_c, but may also include several others, such as an α-parameter for arithmetic mutation, depending of the operators used. There are many approaches to control these parameters to achieve better performance, for instance annealing and several forms of self-adaptation.

Changing those parameters has a direct effect on the characteristics of an EA. Having a low p_m and a high p_c is likely to make the population converge to optima, which means that only a small part of the search space is covered. Similarly, having a high p_m and a low p_c is likely to spread out the population and make it move around a lot. The two characteristics mentioned here, how much of the search space is covered, and how much the individuals move around, are called the *diversity* and the *mobility*, respectively. Many other parameters affect those characteristics – changing an α-parameter will have a great effect on the mobility, for instance.

The goal of changing these parameters is, of course, to make the population behave in a way that increases the chance of finding the global optimum for the specific problem. Usually, the parameters are set to make the population diverse and mobile to begin with, so that the search space is well covered to increase the chance of locating a global optimum. The mobility is then decreased to make the individuals get as close as possible to the optima and not just mutate wildly around them. What is usually wanted when adjusting the parameters, is to control the characteristics of the population to fit the problem.

1.1 Proposition

What we propose, is first of all, to measure these properties. The diversity-measure should give a value corresponding to how spread-out the individuals are in the search space. The mobility-measure should give a value expressing how much the individuals move around between generations. These are just two possible characteristics, but there are many other choices for which characteristic to measure.

With these measures, we will investigate how well the properties can be controlled. That is, can we adjust the parameters of the EA to try to make some property have a given value, when measured with an appropriate measure. This means that we would be able to set a high diversity and mobility in the beginning, and then decrease them. The difference is, that now we control the properties directly instead of through the parameters.

As mentioned before, what we're really after in changing the parameters of an EA is to control the behaviour of the population. If the properties can be controlled more directly, we will be able to strip away one level of indirection. This might possibly improve the performance of existing parameter-control algorithms, especially self-adaptive algorithms, since the complexity of finding good settings might be reduced.

Even if we cannot get the characteristics to act exactly the way we want, it still seems sensible to measure them and letting parameter setting in some way be affected by them. If, for instance, the diversity is low, there is no need to use a lot of energy on crossover if we have a lot of generations left. Otherwise, doing a lot of crossover might be a good idea to home in on the best solution. It has been shown that being aware of the behaviour of a population can give more accurate results with fewer evaluations [1].

2 Measuring diversity

In this project, we have focused on one characteristic: the diversity. In order to control the diversity, we first have to be able to measure it somehow. The measures we use are all taken on populations with real-encoded individuals in a normal search space.

One consideration when it comes to measurements is normalisation, that is, getting values that do not depend too much on the number of individuals, dimensions and other problem-specific factors. We have chosen to normalise with factors that makes the expected diversity of a random population 1. We have found this factor by averaging the measurement over a large number of randomly generated populations. It is probably possible to find an expression for this as a function of the number of individuals, dimensions, etc., but we have not had any luck finding the exact one in any of the cases.

2.1 Midpoint diversity

One diversity-measure is *midpoint diversity*, see figure 1. It is calculated by first calculating the average position of the individuals, and then calculating the average distance of all the individuals to that average. This makes it fairly cheap to calculate, $O(N)$.

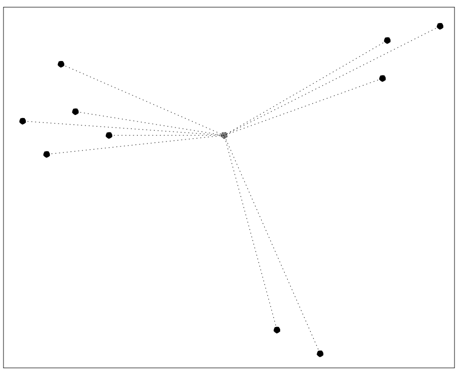

Figure 1: Midpoint diversity

However, this measure can give a high diversity, even if the population is very compact. If, for instance, the population consists of two compact groups of approximately the same size, very far apart, the average will be between them and the diversity will be high. Whether or not this is desirable is a matter of taste.

2.2 Closest-neighbour diversity

The closest-neighbour diversity is calculated by, for each individual, finding the individual that is closest and taking the distance to that. This number is then averaged over all individuals in the population.

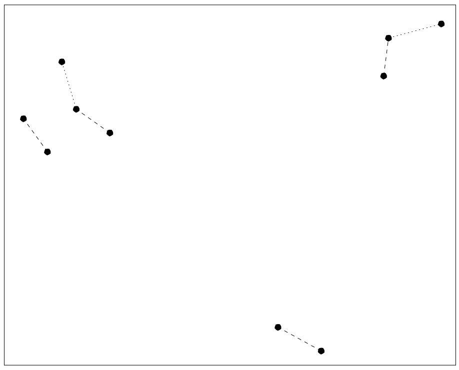

Figure 2: Closest-neighbour diversity

As noted before, we did not manage to find an explicit formula for the normalization factor. We did, however, find a reasonable approximation of this factor (it gives a value that is slightly bigger than the correct one):

$$\sqrt[d]{\frac{V}{\pi n}}$$

where d is the number of dimensions, n is the number of individuals and V is the volume of the search space.

Calculating the closest-neighbour diversity is more expensive than the midpoint diversity – it cannot be calculated in $O(N)$. It is probably closer, however, to what is intuitively understood by diversity.

3 Measuring mobility

One way to measure the mobility of a population is the *closest-neighbour mobility*, which is very similar to the closest-neighbour diversity measure. It is calculated by, for each individual, finding the closest individual in the previous population. We have chosen a normalisation factor that makes the expected mobility between two completely random populations 1.

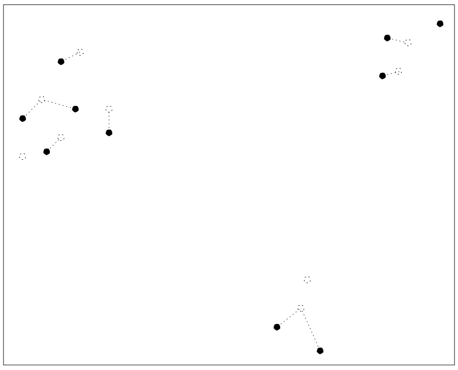

Figure 3: Closest-neighbour mobility

4 Controlling diversity

We have tried different approaches for controlling the diversity. One way is to let the parameters of the evaluation be controlled by the diversity using thresholds, as investigated in [1]. Another is to set the parameters as a linear function of the diversity, and the last option we will describe is a combination of the two.

We have experimented with a number of different problems, which all acted in more or less the same way. The graphs in the following section are all runs on Ackley's F1 of 20 dimensions with a population of 200 individuals over 500 generations. We have also used a range of different operators, but the overall results were the same for those also.

4.1 Thresholds

The main idea in thresholds, as used in [1], is to switch between only mutating and only doing crossover based on the diversity.

The diversity is likely become smaller whenever we do crossover on the population and, conversely, likely to become larger when we mutate the population. We can alternate between doing crossover and mutating, having periods of *exploitation*, where the population is trying to find the best local optima by doing only crossover, and periods of *exploration*, where the population is only mutated, to escape local optima and perhaps find better areas of the search space.

At the initialisation of a population, it is likely to be spread across the problem space – i.e. have a large diversity – and therefore we start off by exploiting, as the population is already nicely spread. We do this until we reach a fixed lower limit for the diversity, the *lower threshold*. Here begins the exploring phase. The population is mutated until we reach a set upper limit for the diversity:, the *upper threshold*. Then we switch back to exploring again. This pattern is repeated during the entire run except for at the very end, where the population is set to only exploit so that it gets a chance to converge completely.

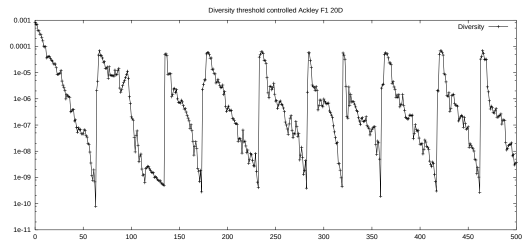

Figure 4: Diversity with threshold control

Using this approproach, we get a certain degree of control over the EA. You cannot set a value and expect the diversity to hit it, but if you choose reasonable higher and lower thresholds, it will most likely stay within those boundaries. In order to do this, you have to have a sense of which values are "reasonable". Choosing thresholds is helped by the fact that the diversity is normalised, but does not have to be any easier than setting p_m or p_c by hand.

Choosing the wrong thresholds will cripple the EA. If the upper threshold is too high, the population will continue to be mutated after the first exploitation phase is finished. If, on the other hand, the lower threshold is set too low, the population may never be mutated and it will quickly find a local maximum – or maybe the individuals will just converge towards the same genes.

Another thing is that one cannot be completely sure the diversity will be smaller with crossover and larger with mutation. With some crossover operators, individuals can actually be pushed further apart and they can sometimes move closer by mutation. Even if this is no problem, the operators may change the diversity too slowly to work well.

When exploring, one would like a large part of the problem space to be covered. This calls for a mutation operator, which ensures that the individuals are spread properly. But, as Ursem have shown, the fitness progress of the best individuals in a population is mostly made while exploiting, therefore it is probably a good idea not to spend too much time on exploring [1]. Instead, one could try and make the mutation operator react directly on the diversity measure: With the midpoint diversity measure, we tried giving individuals a larger tendency to move away from the midpoint (this is also done by Ursem).

Results

As can be seen from the graph, we do get a reasonable degree of control of the diversity. With a mutation operator that lets itself be controlled (as arithmetic mutation with an α parameter), we can actually move the thresholds quite close together and confine the diversity to be within those. The operators have to cooperate, though, since a mutation operator with a large effect will tend to make the diversity break out of the thresholds.

4.2 Linear dependency

Having a low and upper threshold gives some control, but what we are really after is to be able to aim for a single value rather than being between thresholds. Linear dependency is an attempt to do this.

In this approach, we control p_m and p_c by letting them be linear functions of the diversity. A low diversity results in a high p_m and a low p_c. When the diversity increases, p_m is decreased and p_c increased as linear functions of the diversity. When the diversity is low, it will be pushed upwards and when it is high it will be pushed downwards.

Our hope was that the result of these opposite forces would force the diversity to hit a fixpoint somewhere where the increasing and decreasing forces are at an equilibrium. It should then be possible to adjust the location of this equilibrium by changing the exact way p_m and p_c depended on the diversity.

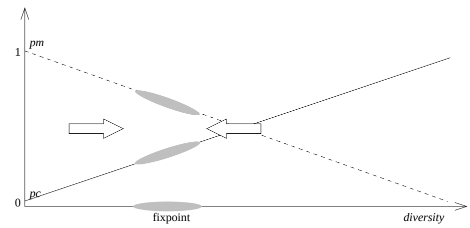

Figure 5: Linear control

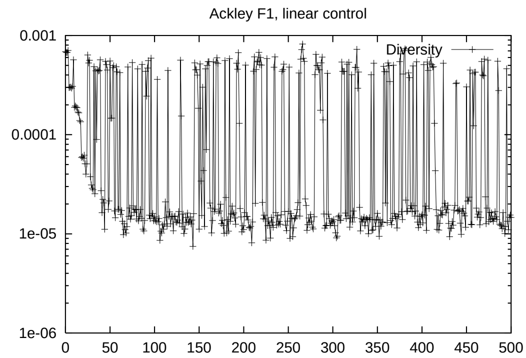

Figure 6: Diversity with linear control

Results

As can be seen from this graph of the diversity over a typical run, the diversity does not really converge. With a clean linear dependency, both the diversity and the parameters change erratically because the coupling is simply too strong. We have tried to introduce an *inertia* parameter, I, which decreases this tendency. When we have calculated a new value for a parameter, p_{div}, from the diversity, we do not just set the parameter p to that value, we take a weighed average instead:

$$p_{\text{new}} = \frac{I \cdot p_{\text{old}} + p_{\text{div}}}{I + 1}$$

This way, the parameters are reacting slower to change. Still, it does not change the fact that there is no fixpoint except with values very close to the natural diversity, that is, the diversity when there is no control.

4.3 Combination

The final approach we tried was a combination of the previous two. We still used two thresholds, but instead of going directly between exploring and exploiting, we changed the parameters gradually between the thresholds.

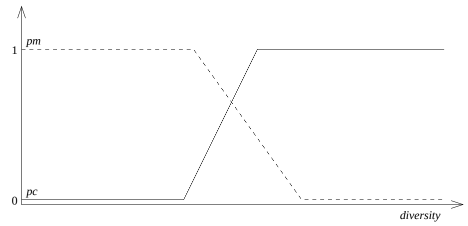

Figure 7: Combined control

We hoped that it would combine the advantages of the two approaches: that the diversity would definitely be between the thresholds and that it would be closer to a fixed value than the with the usual fixpoint control.

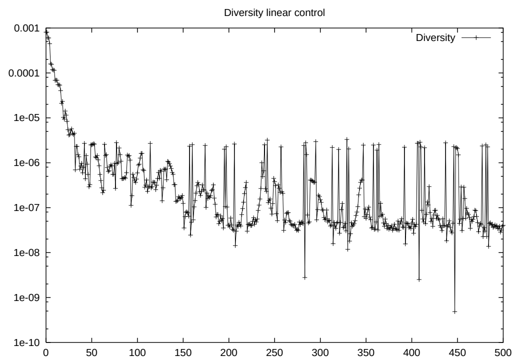

Figure 8: Diversity with combined control

Results

As can be seen from the graph, this did not get us any significant control, but it does actually seem to give a combination of the two approaches: the diversity changes much more rapidly then the linear approach, but it is mostly constrained to be within the thresholds.

5 Conclusion

A defining property of evolutionary algorithms is that they are chaotic. Considering this, it may seem to be not only a bad idea to try to constrain the complexity but also very hard. While we still think it may be a good idea, we certainly agree that it is a hard problem.

We did manage to get some control, as we got the threshold method to behave quite nicely. However, finding the correct values for upper and lower threshold is not at all easier than choosing values for p_m and p_c.

In this project we have only looked at the diversity and only used a few combinations of operators, measures and control strategies. There can potentially be other combinations of those that give much better results.

In this project, we have not been interested in the actual fitness of the individuals of the populations. While our goal was to see how strict a control you could get over a property, it might be that you can get the advantages with only moderate control. It would still be interesting to combine some control over the measurable properties with existing algorithms for parameter control. Our guess is that it would not be unreasonable to expect an increase in performance.

References

[1] Rasmus K. Ursem. *Diversity-Guided Evolutionary Algorithms.* 2003.

Evolving Neural Networks for Classification

Martin Møller Johansen

Abstract— **The traditional way of adjusting the weights of a feed forward neural network is by back propagation. Another way is to use PSOs or EAs, and experiments testing especially the PSO approach are conducted in this paper and compared to back propagation. The experiments are restricted to classification problems.**
Furthermore ways of evolving the number and distribution of nodes as well as the connectivity (ie., given the (fixed) nodes, how should they be connected) have been implemented and tested.

1 Introduction

This paper describes the most common methods for training fixed neural networks and the experience gained when applying these methods to real problems. It also describes how one may use evolution – or, due to time constraints, PSOs – to evolve the architecture of a neural net.

1.1 Neural Networks

A short and informal introduction to NNs can be found in section 12 in [2]. A FFNN is an acyclic weighted graph where each node has an associated activation. Input is fed into the net by presetting the activation of some of the nodes (the input nodes), and the nets output is given by the activation of another subset of the nodes (the output nodes).
The activation α_i of node i is given by

$$\alpha_i = \tau(\sum_j (w_{ji}\alpha_j) - \theta_i) \qquad (1)$$

where w_{ji} is the weight of the edge from j to i, θ_i is a threshold, the sum ranges over all nodes for which there is an edge to i and τ is the transfer function; throughout this paper, τ is the exponential sigmoid which gives the activations range $[0;1]$:

$$\tau(\alpha) = \frac{1}{1+e^{-\alpha}} \qquad (2)$$

In a layered FFNN the nodes are partitioned into a number of layers such that edges are restricted to go from nodes in one layer to nodes in the immediate next layer. Thus the input and output nodes constitute the first and the last layer, respectively.

1.2 Classification NNs

With no further knowledge about the data to be classified, the best one can do is to feed each attribute into its own input node, scaling it into $[0;1]$ in a linear manner (2.4 in [5]). Furthermore there should be one output node for each kind of classification [6]. The class is then determined by which output node is activated, ie. one output node should clearly dominate the other. Ideally, the correct output node j should on data (pattern) p have activation $d_j(p) = 1$ and all the other 0. The goal of training is that the actual output activations $y_j(p)$ should "on the average" be as close to $d_j(p)$ as possible for all p and j.
There are no general criteria for selecting the architecture for a net [6], so one must do an empirical test. This of course calls for an evolutionary approach, which will be explored in 3.2 and 3.3. A too complex net is prone of overfitting to the training data, ie. it fails being general enough to handle yet unseen data, whereas a too simple net may not be expressive enough; for instance, if there are no hidden layers, the net is only able to deal with linearly separable problems (12.1 in [2]).

1.3 Back propagation

Back propagation is described in 12.2 in [2]. The choice of τ in 1.1 makes it particularly easy to implement.
The basic idea is to first adjust the weights leading to the output nodes based on the error of the output, adjust the weights leading to nodes in the last hidden layer based on the output nodes activations etc. The only parameter is the learning rate η. It decides how fast the weights are adjusted: a too low value means

that the net takes a long time to converge, as does a too high value since then the weights may escape local optimas too frequently to ever reach an optimum. However the ultimate success seems quite independent of η as long as it has a reasonable value – I tried $\eta = 0.01, 0.02, \ldots, 1.5$, all of which gave about equal success – so I have not invested much effort into tuning η.

1.4 NNs as individuals

For a fixed architecture, a NN is completely determined by the weights of its edges and its thresholds. Since these are all real valued, it seems natural to let the concatenation of weights and thresholds $\vec{x}$ constitute an individual in either a PSO or an EA. One way of measuring the fitness f of an individual is to record the squared error of the nets output [6]:

$$f(\vec{x}) = \sum_{p} \sum_{j} (d_j(p) - y_j(p))^2_{\vec{x}} \qquad (3)$$

Subscript $\vec{x}$ emphasizes that the nets weights and thresholds are determined by $\vec{x}$. The task is of course to minimize f.

1.5 Combining PSOs and back propagation

Back propagation has the inherent weakness of failing to escape local optimas, which is especially undesirable in the beginning of training where diversity should be encouraged. The ultimate success of the net is entirely dependent on its (random) initialization.
PSOs do not have this weakness, at least not in such a distinct manner, but on the other hand they tend to overshoot their target and hence fail to home in on an optimum. [7] suggests to combine the strengths of the two approaches, ie. use a PSO to evolve (hopefully) better-than-random NNs and then let back propagation work on the best net. This follows the general heuristic of letting an evolutionary algorithm do the initialization and provide a good starting point from where local search can tune in on a more exact solution (sec. 6 in [3]).
An alternative is to do just the opposite: use back propagation to initialize some nets. They may possibly be caught in local optimas, but at least they are (hopefully!) better than random. Then use them as individuals in the PSO. The idea of initializing using traditional methods is described in 7.5 in [2].

2 The Testproblems

Both testproblems are from the UCI Repository [1]. Both are classification problems with all attributes and true classifications known, which makes them particularly useful for introductory experiments.
In testproblem (I) the problem is, given 13 attributes describing a chemical analysis, to determine which (of 3) cultivars growed a certain wine. There are a total of 178 instances. The authors claim that a NN was able to reach a success of 96% using the leave-one-out technique (12.3 in [2]). The problem is considered an easy problem with clear class structures, suitable for preliminary tests.
In testproblem (II) the problem is, given 9 attributes describing the mineral composition and refractive index, to determine the type of a certain piece of glass. There are 7 types, though for one of the types there are no instances. The total number of instances is 214. The authors don't quote how well other classifiers handled exactly this problem of distinguishing between all 7 types, instead the classifications performed have been on a subset of data that fit into a subset of 4 of the classes (those that were windows) which were then partitioned into two (float-processed or not), and the task was then to classify relatively to the latter partitioning, ie. we end up with only 2 classes. This was a bit too close to the level of testproblem (I) for my taste (ie., similar numbers of input and outputs), therefore I chose the (presumably) harder problem at the cost (or benefit) of not being able to do comparisons.

3 Experiments

Throughout this section, *a* $(n_1, n_2, \ldots, n_H)$ *net* is shorthand for a fully connected net with H hidden layers, the first having n_1 nodes, the second n_2 etc. The number of input- and output nodes can be inferred from the testproblem, ie. for testproblem (I) it is 13 and 3, respectively, whereas for testproblem (II) it is 9 and 7, respectively. Since I have no special knowledge about the problem domain, I had no heuristics leading to a more suitable presentation of the input to the net (eg. feeding some input into the net through more nodes). The *success* of a net is the size of the ratio of a given dataset that the net manages to classify correctly.

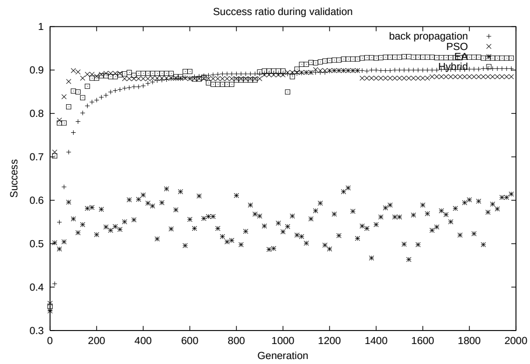

Figure 1: Success ratios for testproblem (I) using a fully connected (10)-net

3.1 Evolving a fixed NN

The purpose of the first experiment was to decide which method was best for training a given (10)-net. This choice of the number of hidden nodes is arbitrary. Five methods were initially used:

- BP with $\eta = 0.6$,
- a PSO with $\phi_1^{\max} = \phi_2^{\max} = 2.0$, $\chi = 1.0$, the inertia $w = 0.7$ and population 20,
- an EA with $p_{\mathrm{m}} = 0.05$, $p_{\mathrm{c}} = 0.9$ and population 100,
- a hybrid of the two first, ie. the PSO for the first half of the epochs followed by BP on the best individual, and
- a PSO where all individuals have first been initialized using BP.

All the parameters were default values, ie. no tuning was performed in advance; hence the comparison is equally fair (or unfair).
The data has been partitioned into two sets of about the same size: a training set and a validation set. The training set is used internally to guide the BP and to compute fitnesses, whereas the validation set is used to test how well the net classifies unseen data – this is of course the more interesting property.

Testproblem (I)

For Testproblem (I), the success rate as a function of epochs is shown in figure 1. The results are averaged over 10 runs, but the standard deviation has been left out to improve readibility. In all cases, except for the first few iterations, it is well below 10%.

BP converges in a slow but safe manner and reaches 89% after 700 epochs; actually it keeps increasing, though insignificantly, ie. it does not tend to get overtrained. The PSO reaches optimality after only 100 iterations, but then tends to decrease again. This may be because it repeatedly overshoots its target and hence fails to converge as also suggested by the oscillations. The hybrid method reaches the best success, so although the PSO by itself performs no better than BP, it clearly gives a better starting point than random initialization. The PSO with BP-initialized individuals does almost as well, indicating that BP conversely gives PSO a better starting point than random; notice that the figure only shows the successes for the PSO-part, ie. even though 2.000 epochs have already been spent on each net during initialization, they are given another 2.000 by the PSO. The EA performs very poorly, perhaps because the number of generations is not nearly large enough.
The methods have been compared based on an equal number of epochs rather than the time spent, thou the latter differs quite a lot: the 10 runs took 2.47 minutes using BP, 24 minutes using PSO, 1 hour and 40 minutes using EA, 14 minutes using the hybrid method (ie., the mean of BP and PSO) and 30 minutes using the BP-initialized PSO (ie., the time taken for BP-training 20 nets (without validating) + PSO). Considering that BP reached (nearly) optimal success after around 700 epochs or 1 minute (for all 10 runs), the PSO reached opti-

mal success after 100 iterations or 140 seconds and the successes were roughly the same, the cost/benefit seems to be highest for BP. The EA performed so poor that it will not be used henceforth.

Testproblem (II)

Figure 2 shows how a (10)-net performed on testproblem (II). Again, the success is an average over 10 runs. This time the two PSOs per-

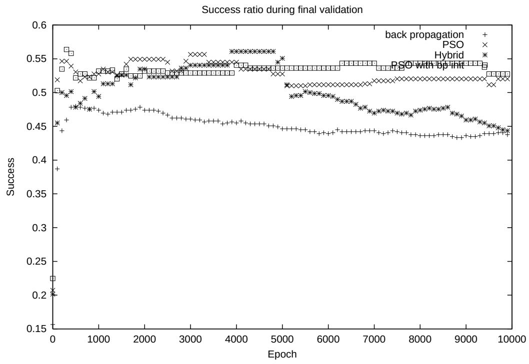

Figure 2: Success ratios for testproblem (II) using a (10)-net during validation

forms best or least bad. It seems as if it makes no significant difference whether the nets are randomly- or BP-initialized, hence the latter seems as a waste of time. As anticipated, the hybrid approximately follows the PSO for the first 5.000 epochs. But as back propagation takes over, the quality rapidly deteriorates, and it ends up just as bad as pure BP.
Notice that back propagation has less freedom: all the thresholds are fixed, whereas for the PSO they are encoded in the individual and therefore they may vary. Hence BP has the additional burden of escaping "unfortunate" initial thresholds, but on the other hand it has less knobs to turn and therefore it may "focus" on the weights. Whether the former disadvantage outweighs the latter advantage is hard to say.

BP and the hybrid method clearly overtrains the net, as is confirmed by figure 3: both of them reach a quite high success of 70% in classifying the training data, whereas the two PSOs reach a success no higher than the success dur-

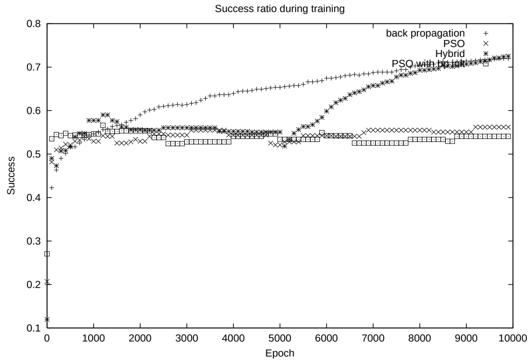

Figure 3: Success ratios for testproblem (II) using a (10)-net during training

ing validation[9]. This indicates that BP is suitable for performing the in most cases not so interesting task of classifying already seen data, whereas the PSO is better at generalizing.
None of the successes are particularly impressive, which may be because a (10)-net is inherently unsuitable for the given problem regardless of weights and thresholds. In the next section a way of evolving the node distribution is described.
Actually the dataset above has been partitioned in three: a training set, a validation set and a final testing set. The successes of figure 2 are for the final testing set to allow for comparison in the next section, ie. the validation set was not used above.

3.2 Distribution of Nodes

For testproblem (I), a fully connected (10)-net seemed to be appropriate, but for testproblem (II) a better architecture was clearly needed. First step is to find another number of hidden layers and distribution of nodes. Rather than performing systematic and exhaustive tests of all possibilities within some range, it seems better (or at least more evolutionary) to use a PSO.
The possible nets are restricted to have no more than 3 hidden layers with no more than 10 nodes in each due to time constraints: the time spent on computing the fitness (see below) rapidly increases with the size of the net.

[9]It should be noted that the two figures are from two different runs

An obvious way to represent such a node distribution as an individual is by making it 3-dimensional and interpret the ith dimension (rounded) as the number of nodes in the ith hidden layer. If the value is zero, it is simply skipped, so eg. $(10, 0, 4)$ represents a (10,4)-net.

When evaluating an individual, a fully connected net with the corresponding node distribution is built and trained. Although back propagation is fastest, it turned out inferiour compared to the PSO on testproblem (II) in 3.1. The PSO also has the advantage of not being so dependent on a lucky initialization. The net was trained for 250 iterations, and the fitness was read off as the success in classifying another dataset (the validation data).

Because the successes differed from run to run, though not as much as for back propagation, the fitness was actually the success averaged over 5 nets with the given node distribution. As a matter of fact the fitness function is not even a function, in that it given the same arguments may yield different results, but the hope is that these fluctuations cancel out when averaging over a sufficiently large number (here: 5).

With the above numbers, one iteration in the "outer" PSO takes around half an hour on a Beowolf.

Since the validation set has an influence on the fitness and hence the evolution of the architecture (ie., it is used for cross validation), a final testing set has been kept separately (2.3 in [5]).

The best individual settled at (8,7) in the 41st iteration and remained there for the next 240 iterations where the PSO was stopped. The fitness (ie., success on validation data) was 42%. The second best individual found was (10,6,10) (fitness 41%).

Figure 4 shows the successes obtained by training with the four methods (excluding the EA) from 3.1. This time the hybrid method performs just as good as the two PSOs, whereas BP is clearly caught in a (worse) local optimum. However in general the net seems no better than the (10)-net (figure 2).

It should be noted that with the representation of node distributions described above, the nets are biased towards more hidden layers:

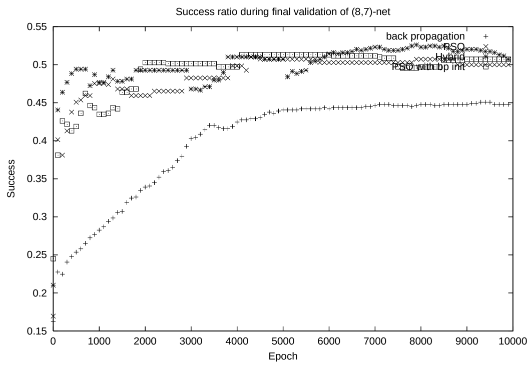

Figure 4: Success ratios for testproblem (II) using a (8,7)-net

the probability of a random net having three hidden layers is $\binom{3}{3}(\frac{9}{10})^3 \approx 73\%$, two hidden layers $\binom{3}{2}(\frac{1}{10})(\frac{9}{10})^2 \approx 24\%$, one hidden layer $\binom{3}{1}(\frac{1}{10})^2(\frac{9}{10}) \approx 3\%$ and no hidden layers 0.1%. So even though zero or one hidden layer is optimal, the PSO may fail to find out, simply because it is a needle in a haystack. One remedy would be to throw in one or more individuals representing nets with zero and/or one hidden layer from start such that the other individuals may be attracted by these in case they do well. Unfortunately this idea (and in fact the problem) crossed my mind too late to redo the experiment, considering that one iteration takes half an hour.

3.3 Connectivity

The default is to use fully connected layered FFNNs, which may seem like a safe choice when one has no particular knowledge of the problem or heuristics for choosing a better connectivity. However evolution may also aid in finding a possibly better connectivity. Two methods are described in 2.3 in [4], a direct encoding method which works on a matrix determining the connectivity of a fixed number of nodes, and a grammatical encoding that determines not only the connectivity, but also the number of nodes.

In this paper only FFNNs are treated and neither of the above two (inherently) respect the feed forward-property. A requirement for this property to hold is that edges are only allowed from a node in one layer to a node in a subsequent layer, which on the other hand means that a node is not necessarily restricted to go between adjacent layers.

One compact (though ambiguous and non-deterministic) way of representing such a connectivity is not to record explicitly whether there is a connection between two given nodes, but rather the probability that two nodes in given layers are connected. For instance, in a net with 1 hidden layer, $(0.5, 0.2, 0.7)$ means that any node in the input layer and any node in the hidden layer are connected with probability 0.5; any node in the input layer and any node in the output layer are connected with probability 0.2; and finally, any node in the hidden layer and any node in the output layer are connected with probability 0.7. The corresponding fully connected layered FFNN, on the other hand, is represented by $(1, 0, 1)$.

More generally, the connectivity of a net with H hidden layers corresponds to an individual with $\sum_{i=0}^{H+1} i = (H+1)(H+2)/2$ dimensions.

The problem of lucky initializations in the fitness evaluation mentioned in 3.2 is perhaps even more distinct here: first, there is the same problem with random weight initializations. Secondly, and probably worse, the actual connectivity is random: only the (expected) frequency of connections between pairs of layers is fixed. Again this is partly remedied by evaluating using PSO instead of back propagation, hopefully making it more insensitive towards weight initializations, and by averaging over a number of nets (5 again, due to time). This way one iteration for the (8,7)-net found best in 3.2 for testproblem (II) takes around half an hour on a Beowolf.

In order not to repeat the "mistake" in 3.2, where the possibly best node distributions with only one hidden layer was a needle in a haystack, I threw in $(1, 0, 0, 1, 0, 1)$ (ie., full connectivity) as an individual from start. However it had a quite bad fitness (a success of 30%) and was hence altered beyond recognition already during the first iteration.

In the 61st iteration, the best individual was found to be $(0.69, 0.70, 0.71, 0.10, 0.36, 0.13)$ with a success of 48%, and it remained so until the 100th iteration where the PSO was stopped. Since there are 7 output nodes (section 2), the expected number of connections going out of each last hidden node is only around one $(7 \times 13\%)$.

Figure 5 shows the success on the final test-

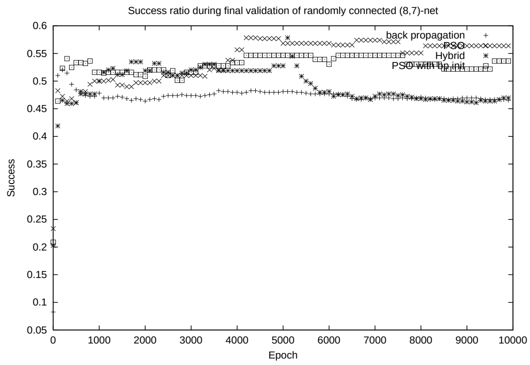

Figure 5: Success ratios for testproblem (II) using a (8,7)-net and best connectivity

ing set when training in the usual four ways. Interestingly enough, the hybrid performs just as bad as BP again – in marked contrast to when the net was fully connected (figure 4). It seems as if BP is simply not ment for training nets where the connections are allowed to "skip" one or more layers. The PSOs manage to get a better success than for the fully connected net, and they are in fact slightly better than for the preliminary (10)-net (figure 2).

3.4 An Alternative Fitness

When evolving the weights in a neural network using PSOs or EAs, the fitness function need not be differentiable. This is in contrast to back propagation where we use the derivative of the error. An alternative to the squared error is to simply measure the success, turning the problem into a maximization problem. An advantage is that we avoid a problem with the squared error fitness, namely that a wrong classification may seem better than a correct classification. The disadvantage is that a "nearly-right" classification gains the same fitness as a "totally-wrong" classification and similarly with "nearly-wrong" and "totally-right" classifications (a totally right classification means that the right output has activation 1 and the rest 0).

Figure 6 shows the success during validation of a (10)-net trained by the above method on testproblem (I). Comparing to figure 1, this method is clearly inferiour. It seems as if the fitness is so coarse that the PSO has no clue as to how the net may be improved. For testprob-

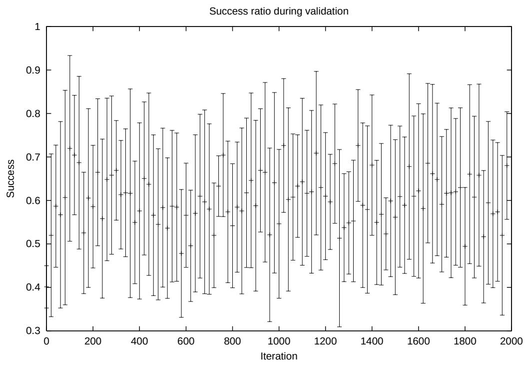

Figure 6: Success ratios

lem (II) using a (10)-net similar conclusions can be drawn. The fitness oscillates around 30%, which is clearly worse than what was obtained using back propagation and the squared error fitness function (figure 2).

4 Implementation

All implementation has been done in Java. As for EAs and PSOs, the EVALife EA library[10] has been used. Since the only neural network I knew of (Dendrite) was implemented in C++, I decided to implement in Java a simple feed forward neural network on my own from scratch as well as a version with back propagation[11]. All implementations performed were straightforward, though a bit time consuming, hence I see no need to describe them in any further details.

5 Conclusions

Implementing and debugging a neural network with back propagation from scratch turned out to be quite time-consuming, so the time available for inventing and implementing innovative ideas has been limited, nor have the ideas turned out particularly succesful.

It turned out that PSOs were best for making generalizing nets, whereas the BPs were best at making nets fit to the training data. However the latter is hardly relevant, so in conclusion PSOs are better though slightly slower at training. EAs are out of the question.
The method for finding the number and distribution of nodes did not manage to find a better net than a first wild guess, and considering the limitations and discreteness on the possible number of hidden layers and nodes one might as well have performed an exhaustive search.
The connectivity as it is defined is continuous and hence more obvious to leave to a PSO since an exhaustive search is infeasible/meaningless. Although the fitness had the "minor" weakness of not even being a function (cf. applying it to the same arguments twice might give different results), it did manage to provide a connectivity slightly better than the default fully connectedness.
Finally, it was found that measuring the fitness of a net as its success in classifying a particular dataset is way to coarse to guide a fruitful adjustment of the weights and thresholds of a net.

[10]Available at `www.evalife.dk/applications.php`
[11]Available at `www.daimi.au.dk/~mmj/ToEC`

References

[1] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998. `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

[2] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, Berlin, 2000.

[3] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1996.

[4] Melanie Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, 2001.

[5] Lutz Prechelt. Proben1: A set of neural network benchmark problems and benchmarking rules. 1994.

[6] C. N. Schizas, C. S. Pattichis, and L. T. Middleton. Neural networks, genetic algorithms and the k-means algorithm: in search of data classification. In J. David Schaffer and Darrell Whitley, editors, *COGANN-92, International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 201–222, 1992.

[7] F. van den Bergh. Particle Swarm Weight Initialization in Multi-layer Perceptron Artificial Neural Networks. In *Development and Practice of Artificial Intelligence Techniques*, pages 41–45, Durban, South Africa, September 1999.

Evolving a Neural Network to Play Connect 4

Peder Herborg and Jesper Fruergaard Andersen

Abstract— **This report describes the different approaches towards training a Neural Network to play the deterministic game Connect 4. Both evolutionary methods such as standard EA algorithms and PSO techniques have been used to develop the neural network, and the more common approach using backpropagation for training have been investigated. We present a hybrid method combining two well known methods into suitable training technique for teaching a neural network to play connect 4.**

1 Introduction

The game Connect four is well known around most of the world. Few people however know how to actually play it well. By the simple rules and looks of the game it seems natural, that a computer should be able to learn to play it better than most humans. Previously good results in the somewhat similar game checkers have been obtained by Fogel and Chellapilla [1]. With this as motivation factor we will try to do the same for the game Connect 4.

It turns out that the two games are not so similar after all, and their approach is not working quite as well with Connect 4. This has to do with the nature of the to games, and thus other methods must be considered for evolving a network to play connect 4.

We wanted to use packpropagation to more directly teach the network some information from games played. We did this by creating a dataset from the board positions in a game played and assigning the start position the value of 1/2. In case of a win the end position got the value 1 and intermediate positions linear value between 1/2 and 1. In case of a lost game the end position was assigned the value of 1/2 and otherwise the same. Realizing that especially the intermediate board positions in the dataset might have gotten assigned wrong values we only trained shortly on the dataset hoping that in the long run the good values would outweigh the wrongly assigned ones.

We combined this with a PSO where the training took place in the games played in the fitness function. We hoped that in the hybrid the training would provide fine tuning while the PSO would provided some diversity.

In the following we will give a quick introduction to the game, the playing strategy and describe the different computer players developed to play this game. Then follows an overview of the different methods applied to this problem, some implementation details and results of the evolution process.

2 The Game

Connect 4 is a fully deterministic game to which the perfect playing strategy is known. It was first solved by Victor Allis in 1988. He showed that the first player can always force a win. However, even with this knowledge it is quite difficult to play the winning strategy even though the 7x6 playing board is quite limited and the search space decreases in size as the game is played. In 1993 John Tromp compiled an opening database for connect 4 classifying all positions after 8 ply [4] as won, lost or draw assuming perfect play from both sides. With this information the search space had been narrowed down enough to solve the rest of the game within reasonable time using clever min-max game tree search with alpha-beta pruning [3], and a set of expert playing rules developed by Victor Allis. So today perfectly playing connect 4 applications based on these results exists. Our goal is to investigate how much of this theory a Neural Network is able to learn.

3 Playing Strategy

Playing the game perfectly is very difficult for humans, and it is by no means a trivial game even though a lot of people think it is more or less randomly decided who wins in the end of a game if it is not decided before that stage.

This is of course by no means random since the game is deterministic, however realizing how the game should be played is non-trivial, and as such expecting a neural network to be able to reach this conclusion is probably too optimistic.

Basically the way to win the game is building up threats so you can later win, or win it directly if your opponent at some stage cannot block all direct threats. If both sides play well, the game wont be decided until it reaches the end game. And here its important to have built up threats on the correct rows. The starting player must have an odd threat in order to win the game. This threat will lead to a win, if the second player has no even threats below it and no odd threats in any other column. If the starting player has no odd threats, then any even threat for the second player will lead to a win. John Tromp has a very detailed analysis of the playing strategy [4].

4 The Players

We tried several different players. Our first player uses an neural network to evaluate each of the up to seven possible moves and selects the move with the highest value, this will be referred to as the network player or pure network player. Our second player implements minimax game-tree search where won and lost positions are assigned fixed values outside the range of possible output values of the neural network and otherwise the neural network is evaluated in the leafs. This player is referred to as the minimax player. In both players the network is fed the board as an array with one entry for each of the 42 cells in the game. An empty cell is represented by a value of 0, a cell occupied by the starting player by a value of 1 and by the second player by a value of -1.

To help the player realize the importance of threats on even and odd rows we implemented a variation of the 2 players where we give the network some extra information. This information is the threats in each cell of degree 2 and of 3 or more. A threat of degree 2 in a cell is where there are already 2 pieces on a row connected to the cell and 4 on a row is possible. Also a threat is not counted if the opponent has a threat of same of higher degree below in the same column. This information is then given for even and odd rows for both the player and the opponent. That is in all 8 pieces of extra information.

We used further 2 players purely as opponents in evaluation. One player always making a random move and one with minimax game tree search and if no winning or blocking move is seen a random move is made.

5 Evolution

Our first attempt on evolving a connect 4 player was inspired by the way Fogel and Chellapilla [1] evolved their checkers player. They used a simple EA where mutation was guided by a self-adaptive parameter vector for each player and steady-state selection. In each generation each parent created one offspring that was a mutation guided by the self-adaptive parameter vector. Each individual was then evaluated by playing games against random individuals We tried two variants of this, one with pure random mutation and one where each player had associated a self-adaptive parameter vector. One thing to note about their approach is that they didn't use any crossover at all.

The neural networks evolved with the EA did learn some about connect 4, but didn't play as well as we had hoped. This led us to use a PSO and backpropagation as described in section 6. The PSO was implemented as described in a tutorial by X. Hu [2].

In general evaluation was done by letting players play against each other or a random player and assigning the value 1 to a win, 0 to a draw and -2 to a lose.

6 Backpropagation

We had many considerations on how to use backpropagation for actually training our neural network. To start with it wasn't considered as an option, since we had no data to train the network. However, given the fact that training the network purely using evolutionary algorithms didn't produce the desired result, a different approach was needed. First a subset of the opening database that Tromph [4] compiled was used for training and validation.

This gave decent evaluations of first 8 moves. However, the database consists of 67557 different board positions so uniquely identifying the classification of every single position was a difficult task.

This was also not the tuning we needed in order to improve the overall networks performance, so instead backpropagation on a single game was used. After each played game backpropagation will push the network towards identifying the current position as good or bad depending on the outcome. This radically improves the networks performance, due to the fact that after making a bad move, it is less likely to do it again next time a similar position arises. However only the last move is taking into account so in order to compensate for this we extended the backpropagation algorithm to evaluate all positions prior in the game, leading to the final result. The entire game is used as training set with the game result gradually decreasing towards the beginning. This enables the network to better learn not only when not to make a loosing move, but also to avoid bad distribution of its pieces which will eventually lead to a loss in most cases.

7 Hybrid

Now we have two training methods for our neural network. We then combined them in a hybrid solution where we used our PSO to evolve some individuals. In the fitness function, which plays a number of games to obtain a fitness values, we turned on training of the neural network using backpropagation. In this way we hoped that the PSO would select good networks and provide some diversity while the backpropagation training did some fine-tuning on solutions.

8 Experiments

8.1 EA

Experiments with the minimax player using the EA without the self-adaptive parameter vector resulted in a player that won about 80% of the games against the random minimax player. With the pure network player the result was about 90% win.

We then tried the random player given the extra information on the game and this resulted in a player that only won about 50% of the games. It appeared that with this approach to learning the player wasn't able to figure out how to use this information or even just disregard it.

8.2 PSO

These experiments were done using PSO and no backpropagation training of the players. The result presented in figure 1 was obtained using the pure network player. The evaluation was done for each player by letting it play 500 games against the random player. Each game on the graph is a 1000 games. We see that the PSO after about 30 iterations reaches about 85% win and then doesn't change much.

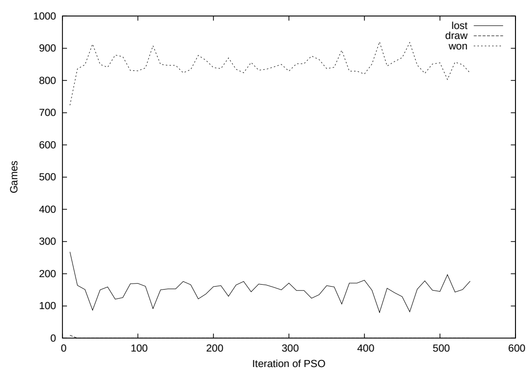

Figure 1: Without minimax.

Figure 2 shows the result of letting the minimax player play against the random minimax player. It wins about 55% of the games but doesn't really improve much.

All results using the PSO were obtained with a population of 25 individuals. Experiments were also done with a population of size 10 which showed significantly worse results.

8.3 Training with Backpropagation

The results of training the network using pure backpropagation vary quite a bit. This results are very dependent on the starting initialization of the network weights, and of course the

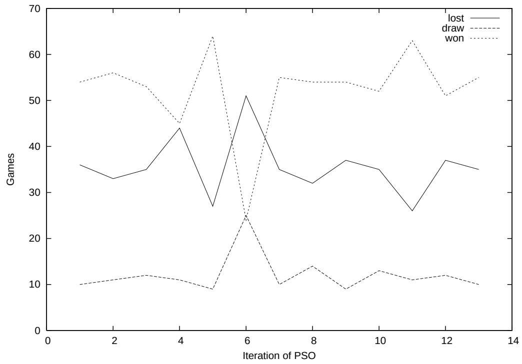

Figure 2: With minimax.

network learningrate and number of games. In figure 3 you see the results of backpropagation over a period of 100000 games. Each interval is 1000 games.

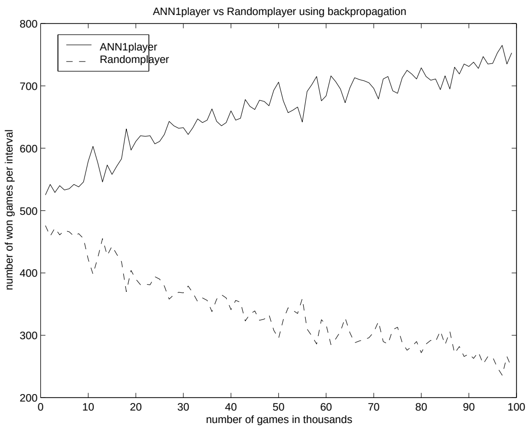

Figure 3: Learning rate on 0.0001.

As it can also be seen on the figure 3 and figure 4, the final result is very dependent on the initial weights and how fast the network learns. With higher learning rates than 0.001 the backpropagation does more harm than good, because the weights are changed too rapidly for the network to stabilize. In both of these tests a random net was initialized, and then the network was set to play 100000 games vs a Random Player using backpropagation on the final result of the games to change the weights. Next we look at results for the same scenario, but where all the game positions leading to the final one are used to alter the weights using

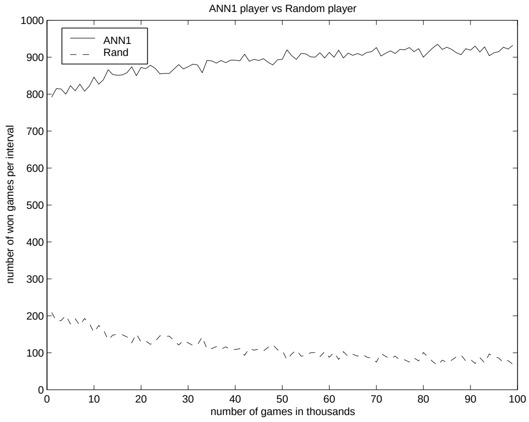

Figure 4: Learning rate on 0.001.

backpropagation.

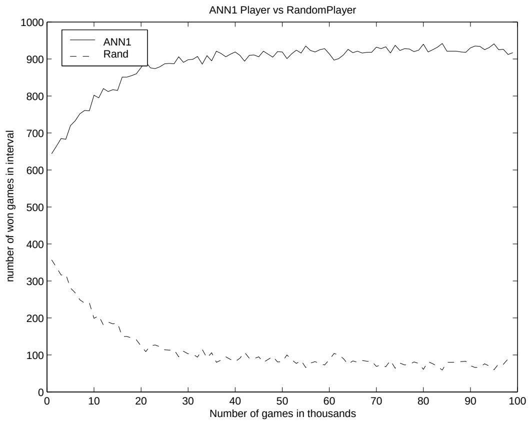

Figure 5: Learning rate on 0.0005.

In figure 5 the statistics show more or less the same as in the two previous ones. It converges a little faster due to the bigger learning set and ends up winning appx 90-95% of the games. The best result was obtained with a trained network using the early EA algorithm, which was then finetuned using backpropagation over 100000 games. This network has a winning statistic of roughly 97.26 percent over 10000 games.

8.4 Hybrid

In all experiments with the hybrid, backpropagation was training on all positions in the game played. In the backpropagation maxepochs was set to 5 and both learningrate and outputlayer_learningrate was set to 0.01.

Two experiments were conducted with the pure neural network player, with the only difference that the experiment in figure 6 500 games was played for each player in the fitness function and the experiment in figure 7 5000 games was played.

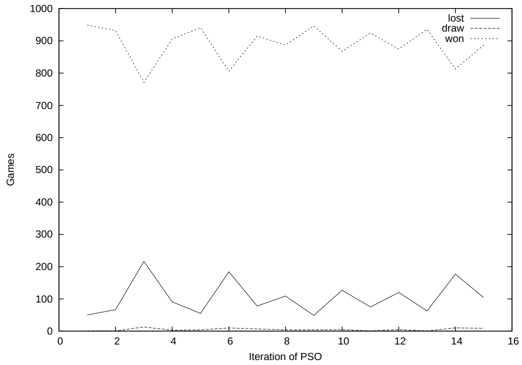

Figure 6: Without minimax, 500 evals

In the experiment with 5000 games played in each iteration, the best net found won 984 out of 1000 games and only lost 14 games.

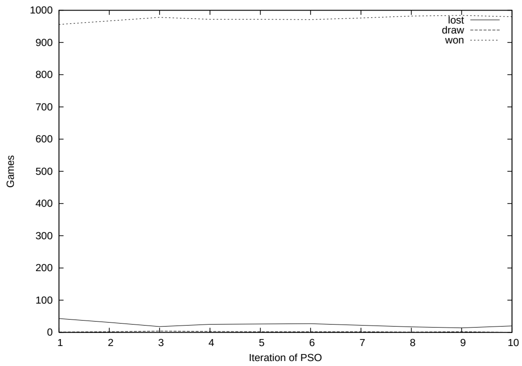

Figure 7: Without minimax, 5000 evals

With the minimax player the best net found won 91 out of 100 games and lost 7.

Unfortunately the hybrid is very time consuming per iteration so we haven't been able obtain result over many iterations. One iteration in the experiment shown on figure 8 takes more than 6 hours and the one on figure 7 more the 5 hours on a P4 2,4 GHz.

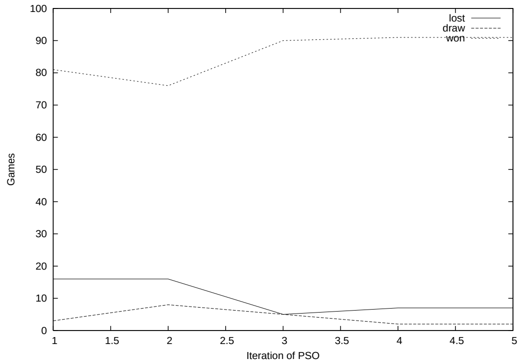

Figure 8: With minimax, 100 evals

9 Conclusions

As earlier stated connect 4 has a winning strategy for the starting player. Using the algorithms described we have been unable to train a network to play perfectly and thus winning every time. Considering the fact that this game was not solved until 1988 and that very few humans are actually able to play the winning, it is not surprising that the network has not been able to learn this either. The network player we have developed using PSO in combination with backpropagation has a fairly good understanding of the positioning of the game, and combining this with the ability to look a few moves ahead it is playing more or less like an average human player. Using the opening database mentioned earlier combined with bigger search trees the network is playing almost at expert level and is very difficult to beat. It seems however that the neural network alone is not able learn to play the game at the level of an average human unless extra expert knowledge is used.

10 Future Work

We have tried out many different techniques, but have yet to be tweaked before they show truly promising results. It would be interesting to experiment with different sizes of the network layers, to see which size is needed to achieve the best results. Furthermore, the combination of PSO and backpropagation need some more tuning to truly take advantage of the hybrid approach. To get more reliable results, training on a minimax player would also be desirable.

References

[1] K. Chellapilla and D. B. Fogel. Evolving neural networks to play checkers without expert knowledge. *IEEE Trans. Neural Networks*, Vol. 10:6:pp. 1382–1391, 1999. http://www.natural-selection.com/NSIPublicationsOnline.htm.

[2] X. Hu. Tutorials (pso/ea). http://web.ics.purdue.edu/ hux/tutorials.shtml.

[3] A. Andersen A. Nazir G. Tanaka J. Keijser and R. Iverach. Tree searching algorithms. http://pages.cpsc.ucalgary.ca/ andre/533/final/minmaxtutorial.html.

[4] J. Tromph. 8 search ply opening book for connect 4. http://www.cwi.nl/ tromp/, 1993.

SOC and the Sandpile EA

Bjarne Ridderberg and Thomas Jakobsen

Abstract—**This report describes some further investigations about Self-organized Criticality and the Sandpile EA. This includes certain variations concerning the usual EA-operators as well as the specific structure of the Sandpile EA.**

1 Introduction

1.1 SOC

The Sandpile EA that we describe in this text heavily relies on the concept of *self-organized criticality* (SOC) which was first introduced by Bak in [1]. SOC is — roughly speaking — a special state of a system that can be characterized neither as *stable* nor *chaotic* but somewhere in between.

What makes the SOC theory really interesting is the fact that many physical systems in nature can be very nicely described as systems in this state of self-organized criticality. Examples of such systems are earthquakes and some properties of stock markets, but the canonical example is — undoubtly — an ordinary pile of dry sand into which a one keeps dropping new grains of sand.

Bak [1] did various observations regarding the nature of those SOC-systems and found that while stable systems — eg.. a flat beach — is almost not responding to small impulses (no earthquake when dropping a sand grain on a flat beach), and chaotic systems are responding randomly, the relationship between the size of disturbance and the resulting effect in SOC-systems can be described by a *power-law distribution*. In the case of the sandpile, this means that when you are dropping sand grains onto a big sandpile, sometimes you will get a very large avalanche.

For further details on the SOC theory, see [1].

2 The Sandpile Model

In [1] Bak shows how to make a simple implementation of a model that imitates a sandpile. Because this model — as described later — is very important for our project, we implemented it in order to gain some insight before we made any extensions.

The model basically consists of an N * N size array of integers where each integer represents a number of sand grains in the sandpile. From time to time one number is increased, and we then check if the numbers exceed the *critical value*. If so, an avalanche is triggered, causing the sand grains to propagate recursively to each side in the array. Grains that propagates outside the array leave the system.

By measuring the number of grains in the pile and the avalanche size over time we got the results shown in figure 1. Here we see,

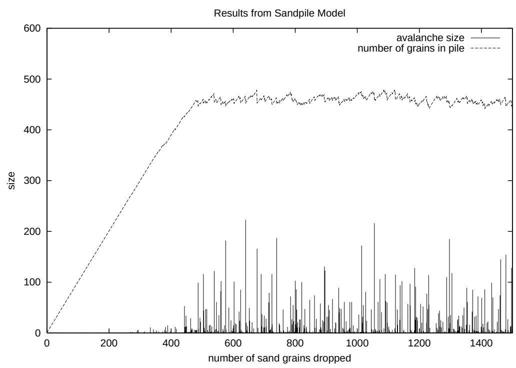

Figure 1: Data from Sandpile Model

that in the first phase, the number of grains increases almost proportional, but after a while — when the pile is filled up — avalanches tend to increase in size, and the number of grains stagnates. When this happens, the system has entered the state of self-organized criticality.

We also verified that the results shown in figure 1 follow the Power Law Distribution described in section 1.1. This was true (except some stochastic noise) as seen in figure 2 where the correspondence between avalanche size and the frequency with which these avalanches occur describes a straight line when plotted in a double exponential coordinate system. (This is generally true for Power Law Distributions.)

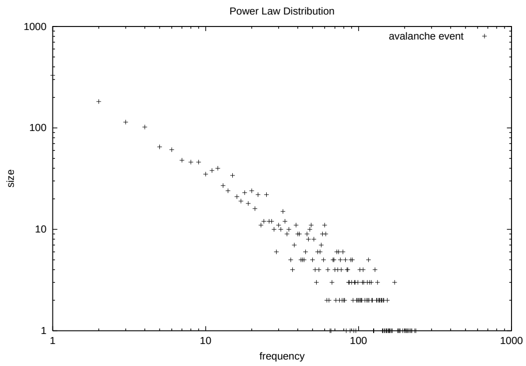

Figure 2: The Power Law Distribution

3 The Sandpile EA

The advantages and disadvantages of the Sandpile EA and of our modifications of this model compared to other EAs can best be explained in relation to the general concept of *exploration* versus *exploitation* described in [3] page 15. The general observation is here that when tuning EAs for better performance it always comes to a trade off between

exploration The various ways of maintaining diversity of the population in order to avoid premature convergence to local optima.

exploitation The various ways of increasing the selective pressure by favourizing the more fit individuals in order to gain fast convergence and to fine-tune the result.

3.1 SOC and the Sandpile EA

While SOC has previously been used to tune single operators like mutation (see [2] and [4]), the Sandpile EA goes one step further and incorporates SOC by using Bak's Sandpile Model as described in section 2 as a basis for its *entire structure*. That is, the EA is itself a sandpile, and the individuals are themselves the sand grains.

So, the Sandpile EA works by maintaining a population of individuals in an array and then — from time to time — dropping a new individual into some place in the array. If the number of individuals in one cell in the array exceeds a certain critical value, this triggers an avalanche during which some individuals propagates around in the array and some individuals might even propagate out of the array.

This structure has the following implications on the Sandpile EA

- *Spatial Structure*
  The Sandpile EA is a spatial EA and thus possesses the same advantages as other spatial EAs, eg. like the Diffusion EA described in [4]. The main motivation for having a spatial structure is to maintain population diversity by dividing the population into subgroups — called *neighborhoods* — that have only limited interaction and thus avoiding premature convergence to local optima.

- *Mass Extinction*
  The Sandpile EA also includes the notion of mass extinction in that there is a certain probability that individuals propagates out of the array and thus leaves the system. In [2] SOC was used to control mass extinction of the population, and this was found to improve their results. With the Sandpile EA, we also use SOC to control mass extinction in that we believe that it will improve our results.

4 Our extensions

In order to investigate the Sandpile EA we implemented some different operators for the EA. (These ideas were partly supplied by René Thomsen from the EvALife group.) This includes

- Different kinds of propagation
- Different kinds of selection
- Local uniform crossover
- Gaussian mutation

4.1 Different Kinds of Propagation

Besides using the standard randomized propagation described in Bak's Sandpile Model [1], we implemented what we call **elite propagation**. In this kind of propagation, each time an avalanche is triggered, we make the most fit individual propagate towards the center of the sandpile. This causes the elite of the population to be centered in the middle of the sandpile after a while — and makes it much more likely for low fit individuals to propagate out of the bounds of the sandpile and thus to get "killed".

In other words, this kind of propagation increases the *selective pressure* on the population. In fact, elite propagation as described above always keeps the most fit individual alive, so using this propagation method causes the Sandpile EA to have **elitism**.

4.2 Local Crossover

In accordance with the idea in other spatial EAs — eg. the Diffusion EA — we don't allow individuals in different neighborhoods to interact, and thus crossover is only going on between individuals in the same neighborhood. In the crossover-phase we therefore scan through each cell and if an individual in this cell happens to participate in a crossover, its crossover mate is selected from the surrounding neighborhood.

We use the uniform crossover operator that is supplied in the RealNumericalGenome class of the JAVA framework made by R. Ursem of the EvALife Group.

4.3 Different Kinds of Selection

As in the crossover phase, selection is also implemented locally. We do this by scanning through each cell in the array; if a cell contains N individuals, we choose N individuals *from the neighborhood* (which also includes the cell itself) which then replace the old individuals.

We have implemented two different ways of selecting these N new individuals. These are

1. Tournament Selection
   When using tournament selection we hold a small competition for each individual in the sandpile. In this competition two individuals selected randomly *in the neighborhood* is compared on basis of their fitness and the one with better fitness is taking the original individuals place in the new version of the sandpile. This selection operator is a choice towards further *exploration* — because some not so fit individuals sometimes replaces more fit individuals.

2. Elite Selection
   In elite selection there is no competition based on randomly chosen individuals in the neighborhood. Instead the N most fit individuals — the "local elite" — are taken from the neighborhood and used for replacement of the original N individuals.

   However, after a while there will often exist many individuals with the same fitness in the sandpile, so in order not to let this elite fill up the complete sandpile far too quickly, we don't just pick the N best individuals, but rather the N best *different* individuals. This causes the new version of the sandpile after selection to still contain some lesser fit individuals, though.

   Elite Selection is, contrary to tournament selection, a choice towards further *exploitation* — because the better fit individuals are always favourized and will relatively quickly dominate the sandpile.

Like the elite propagation described above, the elite selection operator automatically gives the EA the **elitism** feature.

4.4 Mutation

We use standard normal distributed mutation with constant variance. It could have been nice to try out some kind of annealing of this variance, but this is left as a future improvement.

5 The Experiments

5.1 Grid Size

One interesting property of the Sandpile EA is the size of the array that holds the individuals

— that is, the size of the sandpile. To investigate how this size affects the performance of the EA, we tested different grid sizes for AckleyF1 in two dimensions using simple propagation and tournament selection. We would have preferred to test various grid sizes also for other problems and operators, but this would give too many test cases.

Figure 3 shows our results regarding grid size. It seems to suggest (well, you might have to look for a while), that the larger the grid, the better result. If the grid size is small, though, the EA seems to converge faster at first, but it is more likely that it is going to have difficulties escaping the local optima. The limit beyond which there seems to be a good chance for secure convergence towards the global optimum is this AckeleyF1-case a grid of 14 * 14 cells, but it is likely that this exact size is problem dependent.

The results suggest that one always should set the grid size as large as the performance time allows — and especially, one should avoid really small grid sizes.

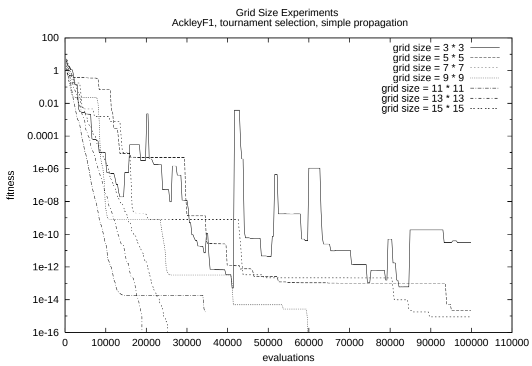

Figure 3: Grid Size Experiments

5.2 Critical Value

Another interesting parameter is the *critical value* for the Sandpile EA. This is the value that determines how many individuals are allowed in each cell in the array before an avalanche is triggered. As figure 4 shows, a large critical value increases the convergence speed a little. But on the other hand, if the value becomes too small (i.e. less than 4) there is a chance that the EA will have difficulties escaping the local optima and thus perform poorly. In Bak's Sandpile Model [1] the critical value is set to 4.

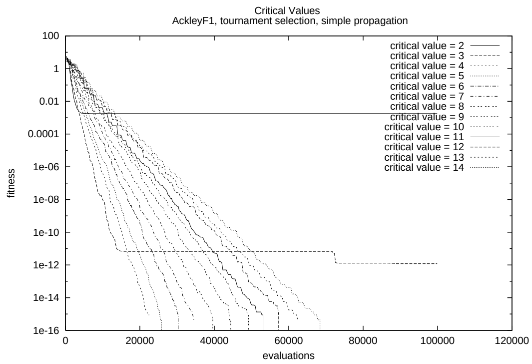

Figure 4: Critical Value Experiments

6 The Experiments

The problems the Sandpile EA has been tested on, were at first relatively easy problem, like DeJong F1. Later the problems became more advanced, to see if was better than the Diffusion EA.

The test problems used were:

- Ackley F1:
  $f(\overline{x}) = 20 + e - 20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right)$

- Griewank F1: $f(\overline{x}) = \frac{1}{4000}\sum_{i=1}^{n}(x_i - 100)^2 - \prod_{i=1}^{n}\cos\left(\frac{x_i - 100}{\sqrt{i}}\right) + 1$

- Rastrigin F1:
  $f(\overline{x}) = \sum_{i=1}^{n} x_i^2 - 10\cos(2\pi x_i) + 5$

- Schaffer F6:
  $f(x, y) = 0.5 + \frac{\sin^2 \sqrt{x^2+y^2} - 0.5}{(1+0.001(x^2+y^2))^2}$

The Ackley F1, Griewank F1 and Rastrigin F1 were in 20 an 100 dimensions.

A common feature of all the test problems is that they are good at revealing how efficiently an EA is at solving multi-modal problems. They all have extremely many local optima, while the global can be rather difficult to home in on.

6.1 Extensions

The different extensions we have made to the Sandpile EA have been compared to each other on the different test problems, to see which is better, and if there is a better at all. Before we started, we expected the combination of elite selection and elite propagation to have an advantage over the tournament selection at least. As the results will show, this is not always the case.

Tournament vs. Elite selection

The tournament and elite selection are different, as described in 4.3, as to what they try to achieve. While tournament tried to maintain **exploration**, elite selection is much more oriented towards **exploitation**.

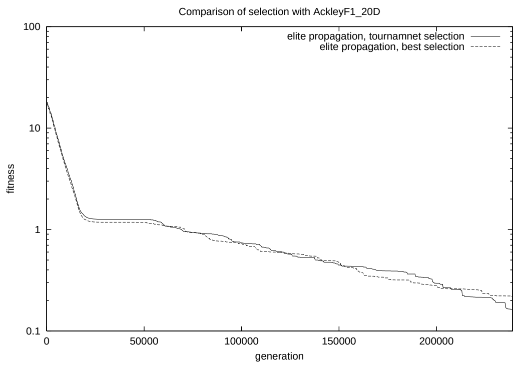

Figure 5: Comparison of selection methods in Ackley F1 20 dimensions

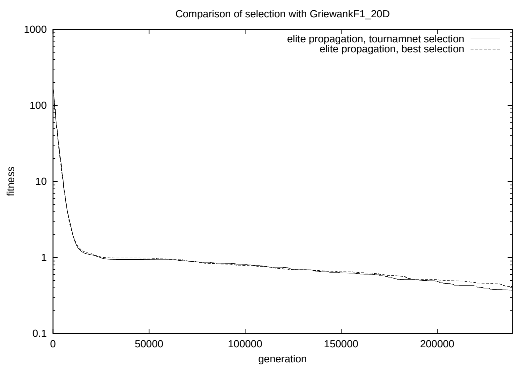

Figure 6: Comparison of selection methods in Griwank F1 20 dimensions

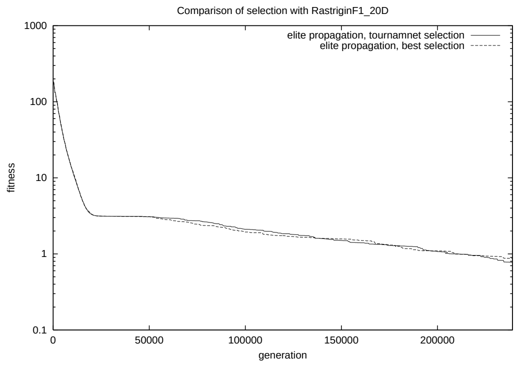

Figure 7: Comparison of selection methods in Rastrigin F1 20 dimensions

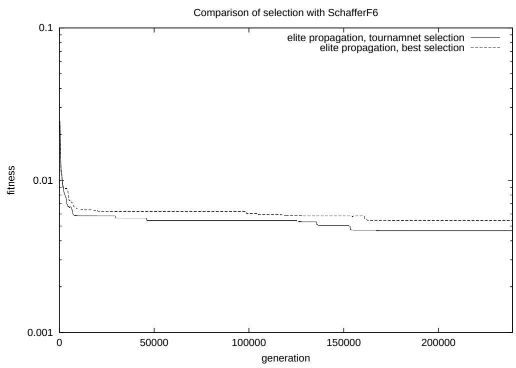

Figure 8: Comparison of selection methods in Schaffer F6

As can be seen from the graphs in figure 5, 6 and 7, in the 20 dimension case the tournament selection was better than elite selection on all the test problems, and also on the Schaffer F6 problem as seen from the graph in figure 8. The advantage might not be large in the cases, but overall tournament was better.

With 100 dimensions the picture was a little bit different.

Here the situation, as seen on the graphs in figure 9 and 10, is a bit different. The advantage of using tournament selection is not as evident as in the case of 20 dimensions. As for the Ackley F1 in 100 dimensions the result was the same.

As the results have shown, tournament selection might not be such a bad idea. Especially with the problems in 20 dimensions the

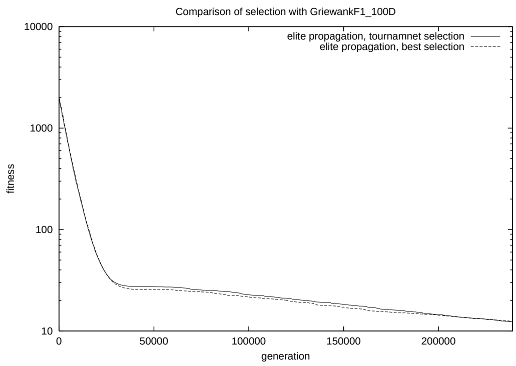

Figure 9: Comparison of selection methods in Griewank F1 100 dimensions

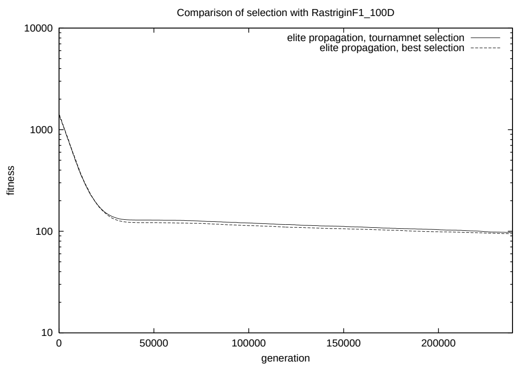

Figure 10: Comparison of selection methods in Rastrigin F1 100 dimensions

difference was evident, but in the higher dimensions the difference was not as clear.

Simple vs. Elite propagation

The difference between simple and elite propagation is, as with tournament and elite selection, also in what they are trying to achieve. Simple propagation is trying to achieve **exploration** while elite propagation is trying to achieve **exploitation**.

The difference in the results between simple propagation and elite propagation is not very noticeable, as seen from the graphs in figure 11, 12, 13 and 14. It does appear, though, that the elite propagation is slightly better than the simple propagation.

In the 100 dimension cases, the difference become a bit more evident, as shown in figure

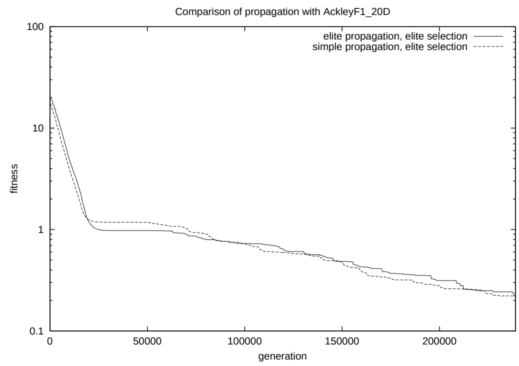

Figure 11: Comparison of propagation methods in Ackley F1 20 dimensions

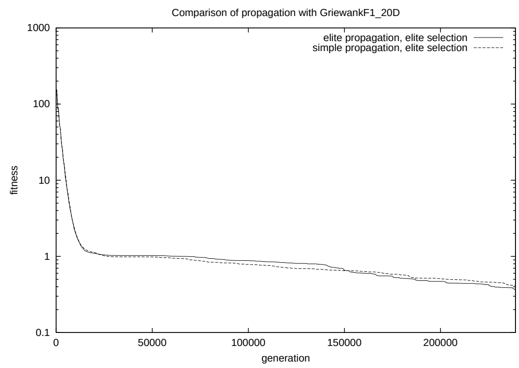

Figure 12: Comparison of propagation methods in Griwank F1 20 dimensions

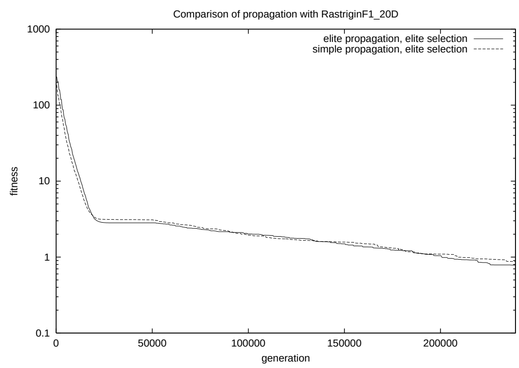

Figure 13: Comparison of propagation methods in Rastrigin F1 20 dimensions

15 and 16. We belive this might be caused by the fact that in simple propagation, the best individuals might “drop of the edge”, while the

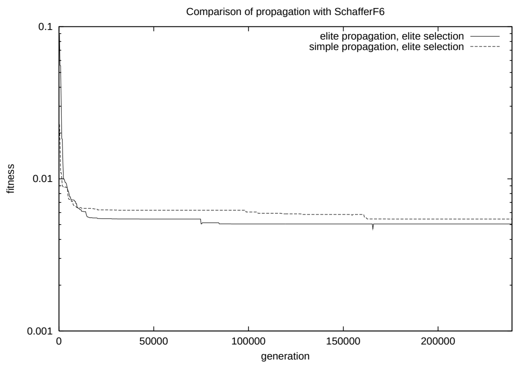

Figure 14: Comparison of propagation methods in Schaffer F6

elite propagation makes sure the best individuals stay in the grid.

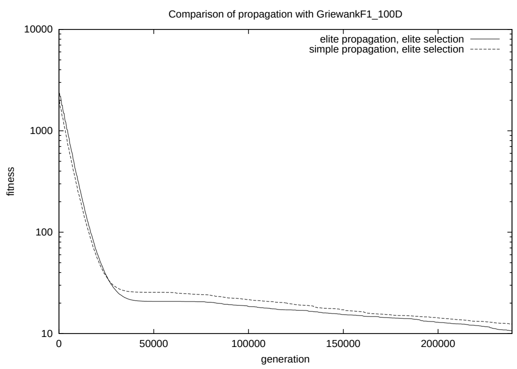

Figure 15: Comparison of propagation methods in Griwank F1 100 dimensions

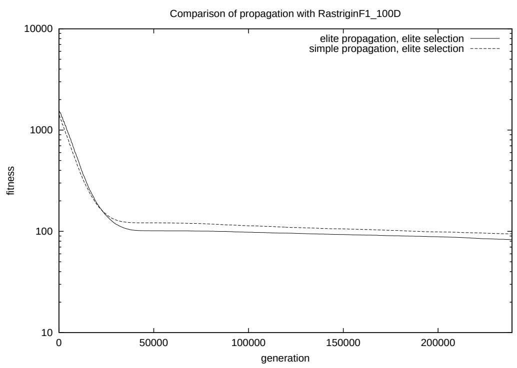

Figure 16: Comparison of propagation methods in Rastrigin F1 100 dimensions

6.2 Sandpile compared to Diffusion

One of our goals from the beginning was to see if the Sandpile EA was better than the Diffusion EA. This is because the Diffusion EA is more or less a simplified version of the Sandpile EA. It was also the basis we used for implementing the Sandpile EA.

The setup for the comparison was made to make it as fair as possible. One of the problems is that in the Diffusion model, the number of individuals is constant, while in the Sandpile model the number of individuals change all the time. To compensate for this we decided to use the number of evaluations as the basis for comparison, instead of using the number of generations. The Sandpile still has more individuals, but the Diffusion now has the opportunity to run for many more generations. The size of the grid we used was $15 * 15$, so the Diffusion EA has 225 individuals while the Sandpile EA will have a somewhat larger number.

The Diffusion EA has been compared with all the different variations described in section 6.1, and the best result has been chosen. This might seem a bit unfair, but as seen earlier, the difference between the variations have not been significant. Otherwise, the two EA's use the same operators.

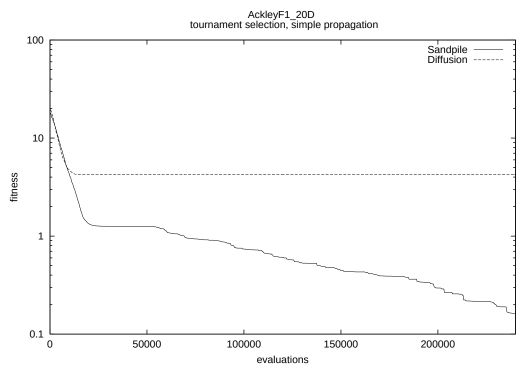

Figure 17: Comparison of Diffusion and Sandpile in Ackley F1 20 dimensions

The graphs in figure 17, 18, 19 and 20 clearly speak for them self. In all the other cases, the results were similar, and the graphs show a good representation of the results.

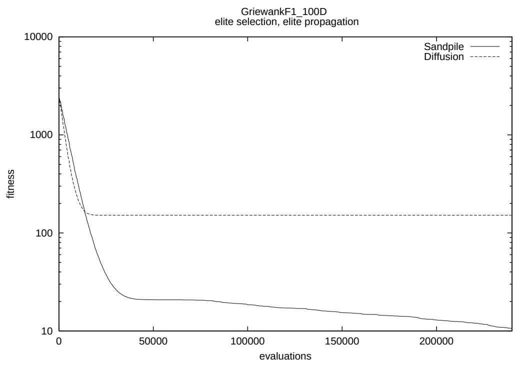

Figure 18: Comparison of Diffusion and Sandpile in Griewank F1 100 dimensions

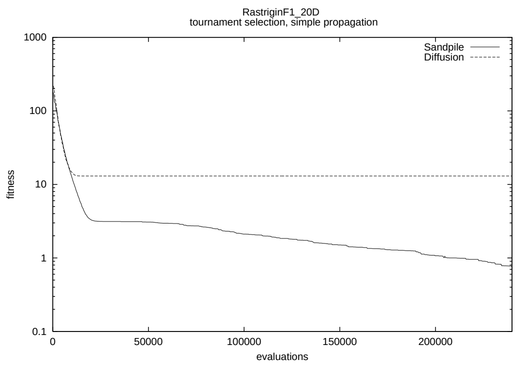

Figure 19: Comparison of Diffusion and Sandpile in Rastrigin F1 20 dimensions

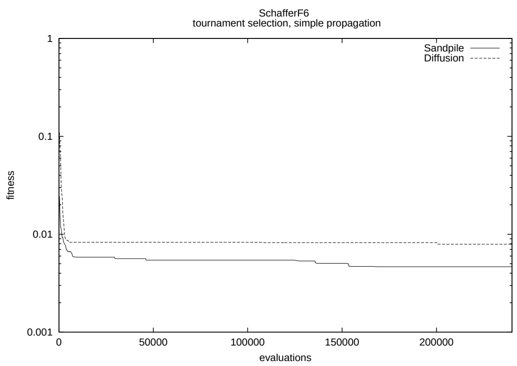

Figure 20: Comparison of Diffusion and Sandpile in Schaffer F6

6.3 Future improvements

In our investigations with the Sandpile EA we didn't care to implement the ideas in the most optimal way with respect to execution time. This hasn't affected the results, but it could anyway be nice to optimize the various data structures in the implementation.

One factor that could improve our results is the use of annealing of the mutation operator. We used plain Gaussian mutation with constant variance, but varying this variance is generally known to improve the performance of most EAs.

Due to a bug in the JAVA framework that we have been using, there have been too many calculations (the bug caused all individuals to re-calculate their fitness after the mutation phase even though only some individuals got their genomes mutated and thus needed re-calculation). Therefore the number of evaluations is probably set too large in our results — but because the bug has been affecting both the performance of the Sandpile EA and the Diffusion EA, we believe that the comparison between these is still valid, but one could improve the results by re-doing the experiments without the mentioned bug.

7 Conclusions

It seems as though the concept of SOC that have been implemented in the Sandpile EA is generally improving the performance of the EA.

With respect to the performance of the operators that we implemented there seems to be no general answer to, which of them are best. In our tests the tournament selection showed slightly better performance, but this might be due to the fact that all of the test problems that we have selected are multi-modal and thus are better solved by maintaining population diversity.

Our experiments have shown that the Sandpile EA performs better than the Diffusion EA in all the chosen test cases. Especially in the harder test problems — those of higher dimensionality — the difference became evident.

References

[1] P. Bak. *How Nature Works: The Science of Self-organized Criticality.* Copernicus, New York, 1996.

[2] T. Krink, P. Rickers, and R. Thomsen. Applying self-organized criticality to evolutionary algorithms. *EvALife Group*.

[3] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolutions Programs*. Springer, New York, 1999.

[4] P. Rickers, R. Thomsen, and T. Krink. Applying self-organized criticality to the diffusion model. *EvALife Group*.

Variance in Gausian mutation

Jeppe Brønsted and Strange From

Abstract—**This paper describes testing of various ways of controlling the variance of the Gaussian mutation operator in a standard EA[12]. It is part of the BREC [1] project describing basic elements of evolutionary computation.**

1 Introduction

In a standard EA the variance of the Gaussian mutation operator is an important factor. This variance can be controlled in several ways.

Traditionally annealing have been used to fine tune solutions near the end of the run, but it is unclear whether this is a good idea. Low variance makes it difficult to escape local optima. Other methods could lead to better results.

The goal of this project is to investigate alternative ways of controlling the variance of the Gaussian operator.

2 The Variance Functions

The tested variance functions can be divided in to 3 subgroups; deterministic annealing methods, deterministic non-monotonic methods and randomized methods.

In all the functions below, t is the generation and $maxgen$ is the maximal number of generations.

2.1 Deterministic Annealing Methods

It might be a good idea to let the variance decrease towards the end of the run. If the EA performs well it will be close to a good solution, and low variance is needed to fine-tune the solution.

The methods described below are all monotonic decreasing functions.

- $f(t) = 1 - \frac{t}{maxgen}$
- $f(t) = \frac{1}{t+1}$
- $f(t) = \frac{1}{\sqrt{t+1}}$
- $f(t) = -\log t + \log maxgen$

2.2 Deterministic Non-monotonic Methods

Sometimes the EA falls into local optima. To escape these, higher variance of the mutation operator is needed. It might be a good idea to "stir up the pot" once in a while.

- $f(t) = 1 + \sin \frac{2\pi t}{100}$
- $f(t) = 6 \sin \frac{2\pi t}{100}$
- $f(t) = \sin \sqrt{t}$

2.3 Randomized Methods

The idea here is to use stochastic distribution to control the variance.

Below $U(a, b)$ is the uniform distribution and $N(a, b)$ is the Gaussian distribution. The tested functions:

- $U(0, 1)$
- $|N(0, 1)|$
- $P(a, 1) - 1$ where $P(a, b) = \frac{b}{\sqrt[a]{U(0,1)}}$
  Low variance most of the time, middle variance more seldom and high variance rarely.

2.4 Graphs

Graphs for some of the functions can be seen in figure 1.

3 Experimental Setup

In all our tests we used the following parameters:

- Population size: 100
- Number of runs: 50

[12] As specified in [1]

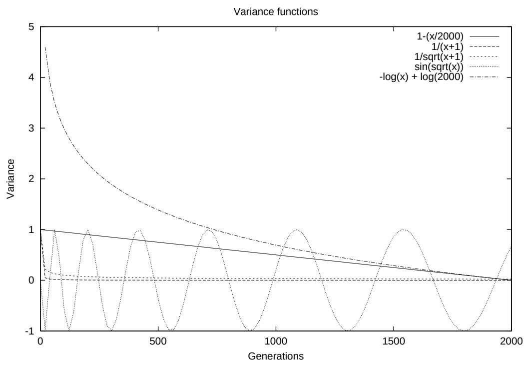

Figure 1: Variance functions

- Probability of mutation: 0.75
- Probability of crossover: 0.9
- Standard tournament selection with two individuals.
- Arithmetic crossover.
- Gaussian mutation with our 10 variance functions.
- Individuals: floating-point numbers.
- We used elitism.
- Parameter for method 8 (Powerlaw distribution. See section 2.3) was fine tuned for every problem.

3.1 The Test Problems

All our tests was performed on the 7 following standard problems:

Ackley F1:
$f(\overline{x}) = 20 + e - 20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right)$

Rastrigin F1:
$f(\overline{x}) = \sum_{i=1}^{n} x_i^2 - 10\cos(2\pi x_i) + 5$

Schaffer F6:
$f(x,y) = 0.5 + \frac{\sin^2 \sqrt{x^2+y^2} - 0.5}{(1+0.001(x^2+y^2))^2}$

De Jong F4:
$f(\overline{x}) = \sum_{i=1}^{n} x_i^4$

Ursem Multimodal F8:
$f(\overline{x}) = 2\cos\left(2\pi \prod_{i=1}^{n} x_i\right) - 4\left(\sum_{i=1}^{n}(x_i+1)^2\right) + \frac{2\sum_{i=1}^{n}\cos(2\pi x_i)}{n}$

Rosenbrock F1:
$f(\overline{x}) = \sum_{i=2}^{n} 100\left(x_i - x_{i-1}^2\right)^2 + (x_{i-1}-1)^2$

Griewank F1:
$f(\overline{x}) = \frac{1}{4000}\sum_{i=1}^{n}(x_i-100)^2 - \prod_{i=1}^{n}\cos\left(\frac{x_i-100}{\sqrt{i}}\right) + 1$

In every test but one, the maximum number of evaluations was 200000. The one that differed was Schaffer which was run with a maximum number of evaluations of only 50000.

3.2 Tuning the parameters

It proved to be difficult to find the optimal value for the parameter in the powerlaw distribution method (See section 2.3). The values varied greatly from problem to problem demanding time consuming fine tuning. The values we found to be the best were:

- De Jong: 350
- Griewank: 93
- Rastrigin: 47
- Rosenbrock: 33
- Ursem: 5
- Schaffer: 13.4
- Ackley: 47

This was the only fine tuning we did. Further fine tuning could have been accomplished by doing a parallel displacement of the variance functions according to each problem. Another thing we could have done was a fine tuning of the *pm* and *pc* but this was out of scope.

4 Results

Graphs showing the average best fitness for all the problems and tables showing standard deviation and the average best fitness have been made. The tables can be seen in section 6.

4.1 Overview of the various methods

1. $f(t) = 1 - \frac{t}{maxgen}$
2. $f(t) = \frac{1}{t+1}$
3. $f(t) = \frac{1}{\sqrt{t+1}}$
4. $f(t) = 1 + \sin \frac{2\pi t}{100}$
5. $f(t) = 6 \sin \frac{2\pi t}{100}$
6. $U(0, 1)$
7. $|N(0, 1)|$
8. $P(a, 1) - 1$ where $P(a, b) = \frac{b}{\sqrt[a]{U(0,1)}}$
9. $f(t) = \sin \sqrt{t}$
10. $f(t) = -\log t + \log maxgen$

4.2 Graphs

As stated earlier we have made some graphs showing average best fitness for the 7 test problems. In each figure you can see 10 different graphs indicating the average best fitness for the 10 variance methods listed above. As can be seen in figure 2-8 the overall picture is that methods 2, 3 and 8 have the best performance. Although methods 2 and 3 in some cases outperforms method 8, it is more stable. Methods 2 and 3 alternates in being the best/second best, but usually the picture is that when method 2 is best/second best, method 3 performs worse than method 8 and vice versa. The other methods performs about the same, and in general quite badly in the light of method 2, 3 and 8.

Figure 8 is showing the results for the Ursem Multimodal F8 problem. This problem is in contrast to the other problems a maximization problem. Some times the fitness is negative and since the y-scale is logarithmic in the figure, the graphs looks rather peculiar.

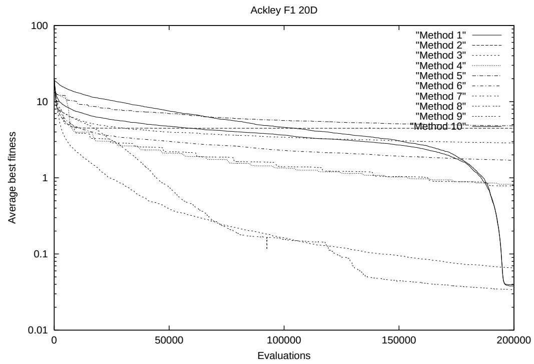

Figure 2: Ackley F1 20D

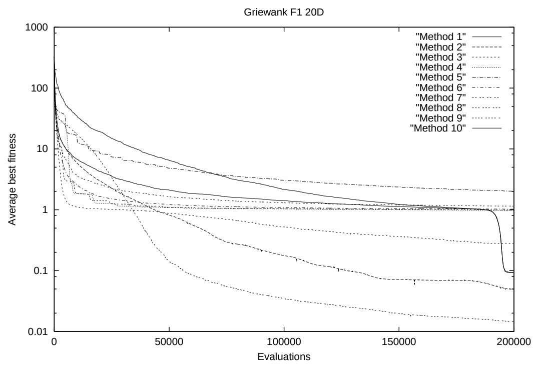

Figure 3: Griewank F1 20D

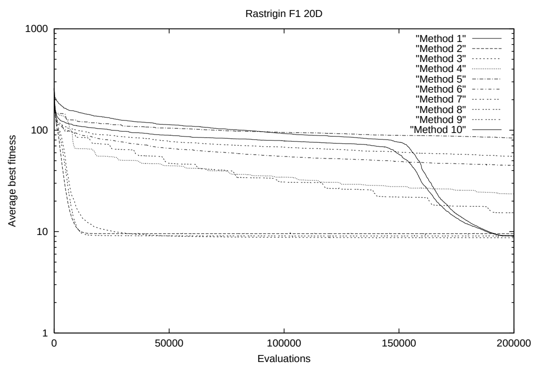

Figure 4: Rastrigin F1 20D

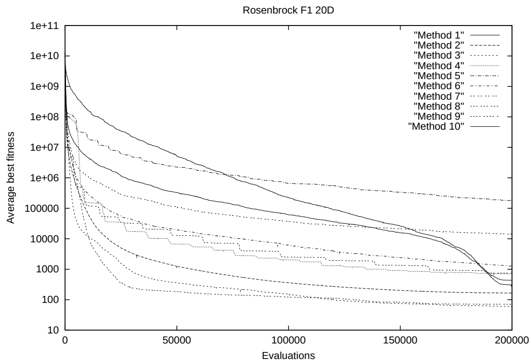

Figure 5: Rosenbrock F1 20D

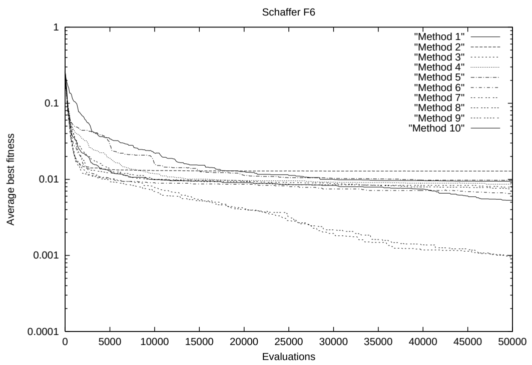

Figure 6: Schaffer F6

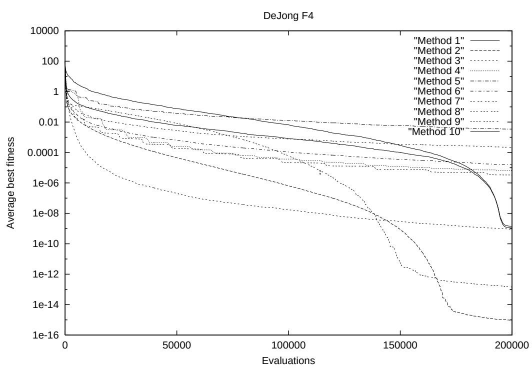

Figure 7: DeJong F4 30D

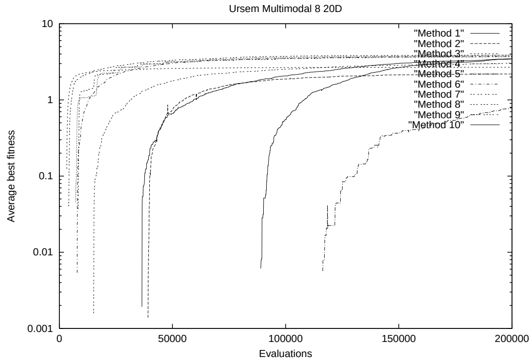

Figure 8: Ursem Multimodal F8 20D

5 Conclusions

As expected we found that annealing is a good idea for controlling the variance of the Gaussian mutation operator. The only method worthy of comparison was the powerlaw distributed function (See section 2.3).

In most of the problems it performs surprisingly well and we believe that with a more thorough fine tuning we would have been able to beat the traditional annealing methods in all the test cases.

The downside of the powerlaw distributed function is that using it is very time consuming because it has to be fine tuned to every individual problem, since it is, we believe, not possible to find a parameter that performs well for all the test problems.

The comparison between the different problems might not be so fair as one could want; Maybe better result could have been obtained for the traditional methods had we fine tuned these to the problems as well. Maybe a parallel displacement of the function could result in a better performance.

6 Tables

Below we have made some tables showing the standard deviation and the average best fitness for each of the 7 test problems. In every table std. dev. and avg. best fit. is shown for each of the 10 variance methods.

Method	Std. Dev.	Avg. Best Fit.
1	0.00875	0.03934
2	1.20916	4.47585
3	0.01497	0.06550
4	0.24612	0.82311
5	0.53654	4.81762
6	0.50886	1.68506
7	0.41188	2.86741
8	0.00763	0.03401
9	0.16417	0.78246
10	0.00724	0.03795

Table 1: Standard deviation and best fitness for Ackley F1 20D

Method	Std. Dev.	Avg. Best Fit.
1	0.03384	0.09294
2	0.11983	0.04908
3	0.20105	0.27665
4	0.04937	0.97741
5	0.35460	2.02150
6	0.02961	1.02392
7	0.04438	1.14800
8	0.01061	0.01454
9	0.04118	0.98915
10	0.03305	0.09183

Table 2: Standard deviation and best fitness for Griewank F1 20D

Method	Std. Dev.	Avg. Best Fit.
1	3.16942	9.08238
2	3.25366	9.55162
3	2.64982	8.69933
4	7.64360	23.56287
5	13.19176	83.45560
6	14.12091	44.95599
7	13.56803	55.36914
8	2.94250	9.05964
9	6.61828	15.30562
10	2.41274	9.16255

Table 3: Standard deviation and best fitness for Rastrigin F1 20D

Method	Std. Dev.	Avg. Best Fit.
1	1100.55124	435.72378
2	622.46991	164.94376
3	74.49928	59.22292
4	902.83254	708.23961
5	133773.09687	178497.75531
6	966.01889	1272.04836
7	12744.27191	14017.20607
8	126.13227	70.89854
9	857.97196	744.31075
10	720.09033	308.43812

Table 4: Standard deviation and best fitness for Rosenbrock F1 20D

Method	Std. Dev.	Avg. Best Fit.
1	0.00388	0.00530
2	0.00919	0.01282
3	0.00127	0.00094
4	0.00244	0.00862
5	0.00153	0.00971
6	0.00391	0.00649
7	0.00333	0.00791
8	0.00204	0.00098
9	0.00319	0.00746
10	0.00160	0.00942

Table 5: Standard deviation and best fitness for Schaffer F6

Method	Std. Dev.	Avg. Best Fit.
1	7.47714e-10	1.09569e-09
2	9.48212e-16	9.28387e-16
3	5.14403e-10	9.04622e-10
4	4.96415e-06	6.57930e-06
5	0.00208	0.00339
6	9.84535e-06	1.53331e-05
7	0.00014e	0.00021
8	9.27773e-14	1.44819e-13
9	1.72030e-06	3.32142e-06
10	1.00288e-09	1.39924e-09

Table 6: Standard deviation and best fitness for DeJong F4 30D

Method	Std. Dev.	Avg. Best Fit.
1	0.42899	3.47528
2	0.90084	2.10898
3	0.72140	2.68487
4	0.09704	3.84904
5	0.52930	0.78674
6	0.39119	3.72360
7	0.55918	3.03573
8	0.06087	3.87137
9	0.39638	3.68622
10	0.31226	3.44854

Table 7: Standard deviation and best fitness for Ursem Multimodal F8 20D

References

[1] Rasnus K. Ursem. Brec projects. 2002.

BREC project: Comparison of selection operators

Niels Thrane

Abstract— **This report contributes to the BREC project, which is concerned with basic research in evolutionary computations. In the field of evolutionary algorithms there are several different selection operators. And while most of them do work, they have not been thoroughly compared in terms of performance. This report compares four of the most common selection methods on seven different well-known benchmark problems.**

1 Introduction

It is well known that mutation and cross-over operators have a big impact on the quality of the solutions that an evolutionary algoritm can discover. However, when it comes to selection operators the picture is not nearly as clear. Essentially, there are two kinds of selection operators, namely one that chooses individuals for recombination, and one that selects the most fit individuals for the next population. In these experiments the recombination selection is always randomized, whereas the fitness selection is one of these:

- $(\mu + \lambda)$: From μ parents λ offspring are created and the next population is chosen to be the μ most fit individuals from the whole lot.
- (μ, λ) : From μ parents λ offspring are created and the next population is chosen to be the μ most fit individuals from the λ offspring. Obviously, $\lambda \geq \mu$.
- Stochastic tournament selection with replacement: 2 individuals are chosen at random from the entire population. The best individual is only chosen with a certain probability. This process is repeated `popsize` times.
- Deterministic tournament selection with replacement: 2 individuals are chosen at random from the entire population. The most fit individual of the two is copied to the next population. Essentially, this is a special case of the stochastic variation with the probability of selecting the most fit individual always equal to 1. Again, this proccess is repeated `popsize` times.

2 Model

All of the experiments have been carried out with elitism enabled and using floating point representation. The parameter setup for the EA was:

- Population size (μ): 100
- Probability of crossover: 0.9
- Probability of mutation: 0.75
- Mutation operator: Gaussian with $1/(1+\textit{generation})$ as variance.
- 200,000 evaluations (50,000 for the Schaffer benchmark).

All of the above parameters were fixed during all the experiments. Better solutions might have emerged if one had fine-tuned these operators.

3 Benchmarks

The seven benchmarks that were used in the tests are listed here:
Ackley F1, 20 dimensions:

$$f(x) = 20 + e - 20 \exp\left(-0.2\sqrt{\tfrac{1}{20}\sum_{i=1}^{20} {x_i}^2}\right) - \exp\left(\tfrac{1}{20}\sum_{j=1}^{20}\cos\left(2\pi x_j\right)\right)$$

Griewank F1, 20 dimensions:

$$f(x) = \frac{\sum_{i=1}^{20}(x_i - 100)^2}{4000} - \prod_{i=1}^{20}\cos\left(\frac{x_i - 100}{\sqrt{i}}\right) + 1$$

Rastrigin F1, 20 dimensions:

$$f(x) = 200 + \sum_{i=1}^{20} x_i^2 - 10\cos(2\pi x_i)$$

Rosenbrock F1, 20 dimensions:

$$f(x) = \sum_{i=2}^{20} 100\left(x_i - x_{i-1}^2\right)^2 + (x_{i-1} - 1)^2$$

Schaffer F6:

$$f(x,y) = 0.5 + \frac{\sin^2 \sqrt{x^2+y^2} \; -0.5}{(1+0.001(x^2+y^2))^2}$$

De Jong F4, 30 dimensions:

$$f(x) = \sum_{i=1}^{30} i x_i^4$$

Ursem multimodal F8, 20 dimensions:

$$f(x) = 2\cos\left(2\pi \prod_{i=1}^{20} x_i\right) - 4\left(\sum_{i=1}^{20} (x_i+1)^2\right) + \frac{2\sum_{i=1}^{20}\cos(2\pi x_i)}{n}$$

All of the following results are from data collected during 50 runs for each set of parameters.

4 Parameter tuning

4.1 Stochastic tournament selection

For stochastic tournament selection I needed a probability setting for the tournaments. I immediately discarded the fitness values of the individuals as a measure for the probability. The reason was that two very low fitness values – say 1 and 3 – would give 75% probability for the most fit individual, whereas high fitness values – say 101 and 103 – would give almost fifty–fifty odds even though the difference in fitnesses is the same. Therefore, I settled on a fixed probability setting. Not surprisingly, values below 0.5 did not give good results and a higher probability for selecting the most fit individual always gave better results. This also meant that deterministic tournaments (probability 1 of selecting the most fit) were always better.

4.2 $(\mu+\lambda)$ and (μ,λ) selection

For $(\mu+\lambda)$ and (μ,λ) a good λ value proved similarly easy to find. In all of the experiments a very high setting consistently gave good results. For $(\mu+\lambda)$ this means a λ setting of between 1.6 to 2.0 times μ. Because (μ,λ) selection discards the parents in each generation the best setting here was between 2.6 and 3.0.

Higher settings were also tested, but they did not improve the solutions noticeably. In fact, in some cases the solutions were slightly worse.

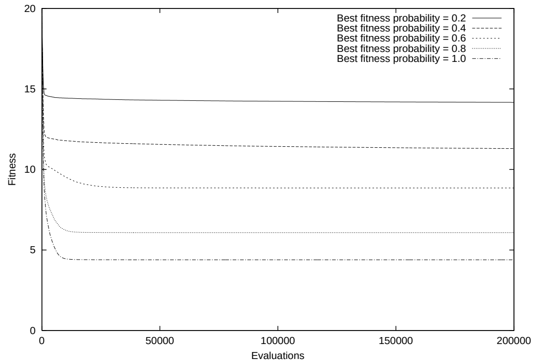

Figure 1: Stochastic tournament selection for Ackley F1 20D.

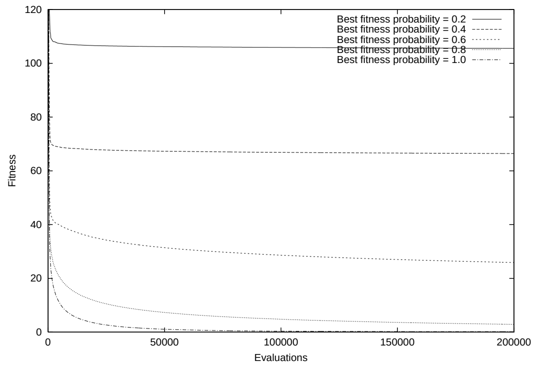

Figure 2: Stochastic tournament selection for Griewank F1 20D

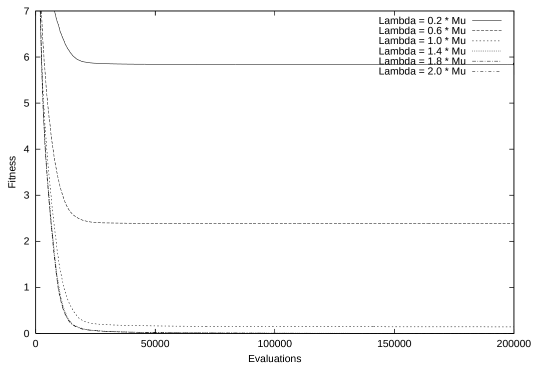

Figure 3: $(\lambda+\mu)$ selection for Ackley F1 20D.

5 Results

Below are the results from the comparison of the different operators.

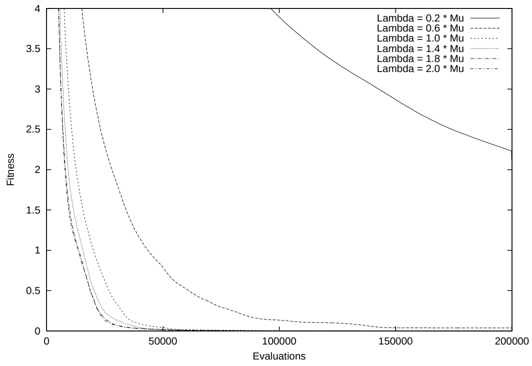

Figure 4: $(\lambda + \mu)$ selection for Griewank F1 20D

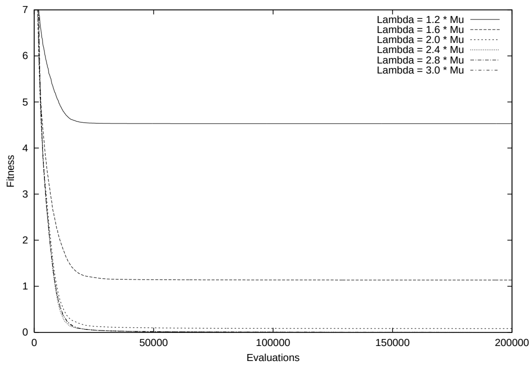

Figure 5: (λ, μ) selection for Ackley F1 20D.

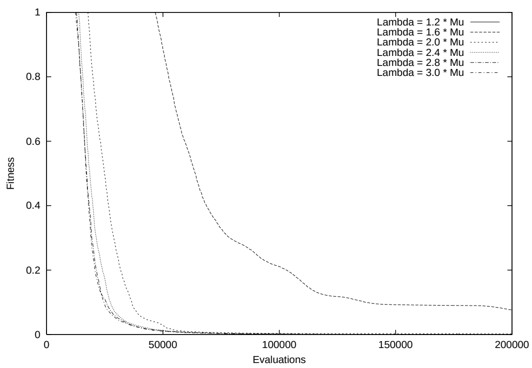

Figure 6: (λ, μ) selection for Griewank F1 20D

5.1 Graphs

Below are graphs for all seven benchmarks that compare (λ, μ), $(\lambda + \mu)$ and deterministic tournament selection. The most successful parameter setup was used in all cases.

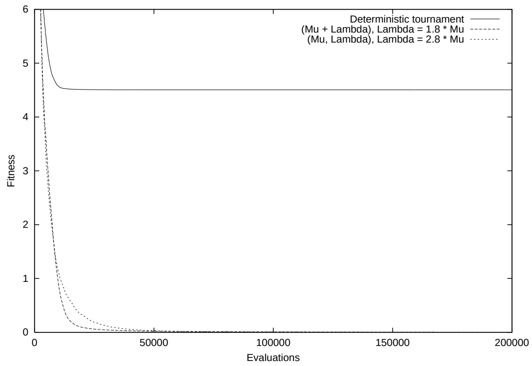

Figure 7: Ackley F1 20D.

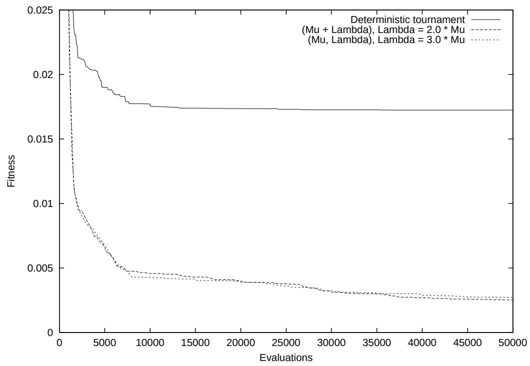

Figure 8: Schaffer F6

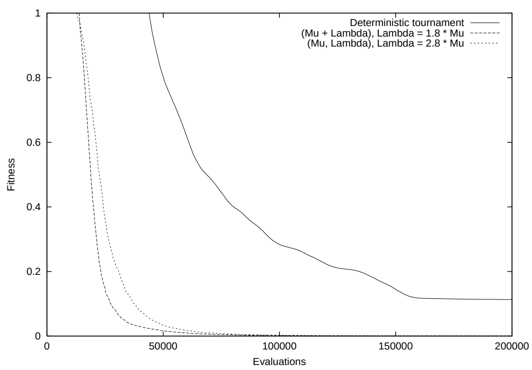

Figure 9: Griewank F1 20D

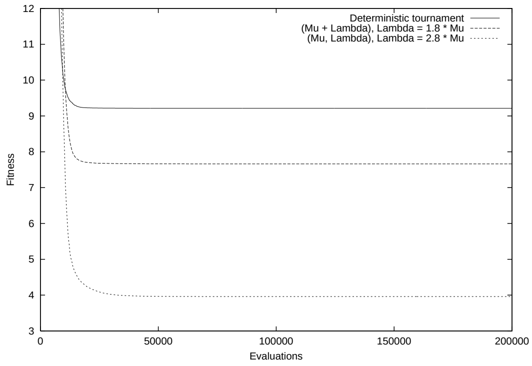

Figure 10: Rastrigin F1 20D

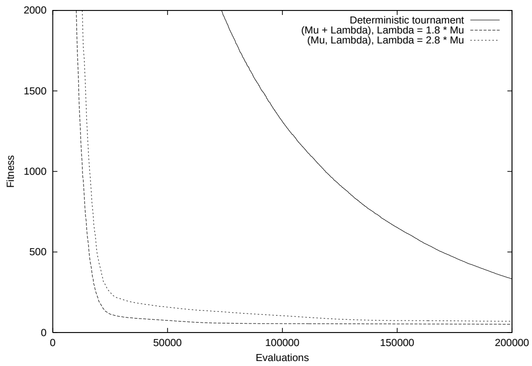

Figure 11: Rosenbrock F1 20D

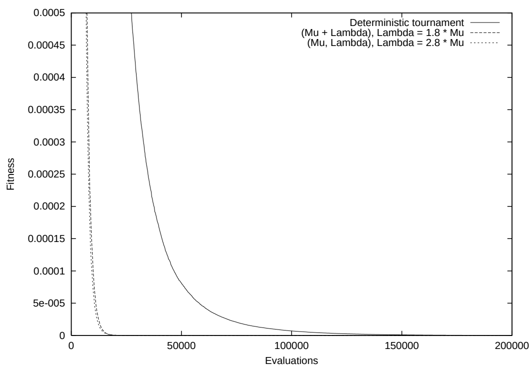

Figure 12: De Jong F4 30D

5.2 Tables

The numerical results are listed here. SD is the standard deviation.

Selection type	Avg. Best Fit	SD
Tournament	4.5041	1.2326
$(\lambda + \mu)$	0.0026	0.0005
(λ, μ)	0.0019	0.0004

Table 1: Ackley F1 20D

Selection type	Avg. Best Fit	SD
Tournament	0.0172	0.0127
$(\lambda + \mu)$	0.0025	0.0041
(λ, μ)	0.0027	0.0036

Table 2: Schaffer F6

Selection type	Avg. Best Fit	SD
Tournament	0.1131	0.2778
$(\lambda + \mu)$	0.0003	0.0001
(λ, μ)	0.0008	0.0024

Table 3: Griewank F1 20D

Selection type	Avg. Best Fit	SD
Tournament	9.2133	3.0539
$(\lambda + \mu)$	7.6612	4.5406
(λ, μ)	3.9600	2.4329

Table 4: Rastrigin F1 20D

Selection type	Avg. Best Fit	SD
Tournament	332.4587	1264.1685
$(\lambda + \mu)$	51.2055	78.6179
(λ, μ)	69.9949	107.6469

Table 5: Rosenbrock F1 20D

Selection type	Avg. Best Fit	SD
Tournament	9.08E-8	6.42E-7
$(\lambda + \mu)$	3.23E-15	1.84E-15
(λ, μ)	6.84E-15	5.34E-15

Table 6: De Jong F4 30D

Selection type	Avg. Best Fit	SD
Tournament	2.2623	2.47E-7
$(\lambda + \mu)$	2.2623	3.27E-7
(λ, μ)	2.2623	4.73E-7

Table 7: Ursem F8 20D

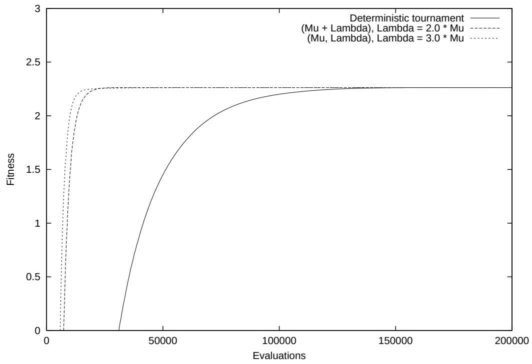

Figure 13: Ursem F8 20D

6 Conclusion

As could be expected from the above, the clear loser of the test is tournament selection – both stochastic and deterministic. When it comes to the other two types of selection, it is more difficult to say which is the better one. They both generate good results that are very close to one another.

The better results may very well be due to a more diverse population, but this is just speculation. A more close examination would perhaps give more insight. It would also be interesting to tune p_m, p_c and the other parameters that were left fixed.

Co-Evolution of Pacman and Ghost controllers using Evolutionary Algorithms

Thomas Clemen Pedersen and Simon Clemen Pedersen

Abstract— **The notion of Co-evolution refers to a simultaneous complementary evolution of two distinct populations sharing an intertwined fitness landscape.**
The objective of this paper is to investigate competitive co-evolution in which there is an inverse fitness interaction between predator and prey populations, as hypothesised to occur in nature. The fitness of individuals in the predator population is measured through competition with the fitness of the individual(s) in the prey population. Competitive co-evolution is applied and explored by using EAs in a game playing context for evolving controllers for both pacman- and ghost-players in the classical game of Pacman.

1 Introduction

Co-evolution is a common phenomenon in nature, where organisms evolve and better themselves through interactions with other organisms in either competition for resources, or through cooperation i.e. symbiosis.
Competitive biological systems are often divided into predator/prey or host/parasite [3] systems. The main difference being the objective of the involved individuals.
This paper's focus is on the predator/prey systems in which the population is divided into two disjoint sets of individuals. The predators are defined by their objective to better themselves at hunting and catching the prey. The objective of the prey is defined by eloping the predators and perhaps pursuing some secondary agenda. These objectives can easily translate into many game playing strategies. This paper describes our exploration of competitive co-evolution applied on the game of Pacman.
Evolutionary game theory originated as an application of the mathematical theory of games to the described biological context. It arose from the realisation that frequency-dependent fitness introduces a strategic aspect to evolution.
Pacman presents the notion of both predators and prey, where the

> *"predator/prey relationships also involve coevolution, with an evolutionary advance in the predator, for instance, triggering an evolutionary response in the prey."*

We introduce a way of co-evolving strategies for both ghosts and pacman players in a competitive environment, utilising the all vs all approach, where each individual is tested against all others.

2 The Game

In order to evaluate Pac playing strategies, we need to define the rules of the game. These are loosely based on the classic Pacman game. We enforce our own interpretation upon it in order to make the programming relatively simple, and we only use a subset of what hard core 80's arcade gamers would denote Pacman. In some of the experiments we subtly tweak these rules (see section 6). This is the layout of what we denote the **base system**:

The Board We use the classic starting board with $23 \times 19 = 437$ square tiles, of which 234 are walls, 181 are occupiable spaces containing cheese, 4 are spaces containing drugpills and 18 tiles are unoccupied. The ghosts start in a centrecage, and there is a throughgate transporting between positions $(10, 0)$ and $(10, 18)$

Mission statements The pac has to eat all the cheese to complete it's mission. The ghosts can prevent this by killing the pac three times. There is 1 pac, and 4 ghosts.

Movement A turn based system is used. Actors (ghosts and pacman) are allowed to move into any adjacent square not containing a wall. Diagonal movement is not allowed. To emulate a faster pacman, it gets an extra move for every 3 turns. Pacman is allowed to stand still, ghosts are not.

Eating If the pac moves into a square with a cheese or a drug pill, it consumes it. Drug pills place the pac in drug mode for the next 40 turns.

Life and Death in the Maze If a pacman and a ghost meet in a square the pac dies, except if it is in drug mode in which case the ghost dies. Pacman has 3 lives to complete its mission of eating all the cheese. When a ghost dies, a 'ghost of the ghost'[13] slowly moves into the centre cage and there respawn.

Time limit The Pac needs to complete it's task within a given time limit. Computationally this also makes sense, since we do not want our experiments to run indefinitely. We have set the time limit in turns to 3 × board width × board height. If the Pac can visit every tile 3 times (and more considering most of them are walls anyway) it should be able to eat all the cheese.

3 Playing Theory

When you play a game such as Pacman, you need to have a strategy. For instance, it would seem wise to move your Pac towards the cheese, but keep it away from the ghosts. The ghosts on the other hand might choose to be aggressive or defensive, or use various forms of cooperation.
All our deliberations on this subject gave birth to the notion of a strategy represented as a genome with normalised signed pseudo real values (i.e. they belong to the range $[-1.0; 1.0]$ in long doubles). The genes in the array represent an adjustable handle we can use to change behaviour of our Pacman or Ghost.

[13]'Ghosts of ghosts' in the classic game is represented by a pair of floating eyes

3.1 Pac Strategies

The following genes were selected for the Pac:

GREED The Pac needs to eat cheese in order to win. This gene represents the Pac's desire to move toward the cheese.

COURAGE Running away from the ghosts might seem prudent, but sometimes it is also a good idea to keep them closer in case you are able to snag a drug pill and start killing ghosts. This value represents a Pac's desire to move towards a ghost. Of course, a negative value will trigger the Pac to move away from the ghosts.

DRUGHUNGER Eating drugs is good for you. This gene regulates the Pac's desire to move toward drug pills.

AGGRESSION When the Pac is on drugs it might choose to spend the time eating cheese without interference, or it might prioritise killing ghosts. This gene regulates the drugged Pac's desire to move towards ghosts.

3.2 Ghost strategies

The following genes were selected for the Ghosts:

SPREAD Fanning out over the board and thereby covering more ground might be sensible. This gene regulates a Ghost's desire to move away from the other ghosts.

BACKTRACK Staying in one area playing 'area defence' might be helpful. This gene controls whether a ghost moves towards or away from previous positions.

COURAGE Running from the Pac when it is drugged is a good idea, but you wouldn't want to run to far, because the pac will go back to normal after a while. This gene regulates the Ghost's desire to move towards a drugged pac.

AGGRESSION Obviously the ghosts need to kill the Pac, so an aggression gene might seem redundant. However, it might be prudent to prioritise the other genes over aggression for an optimal result.

DEFENCE An alternative playing style for ghosts could include a low aggression and a tendency to stay in areas where there is cheese. The Pac will at some point converge on the cheese if it wants to win. This gene controls the Ghost's desire to move towards cheese.

4 Applying Co-Evolution.

Co-evolution is often utilised when dealing with closely interacting species, where a change in the fitness of one species directly reflects on the fitness of others. Jan Paredis [3] introduces the concept of an inverse fitness interaction when working with competitive co-evolving systems.

Success on one side is felt by the other side as failure to which must be responded in order to maintain one's chances of survival. There is a strong evolutionary pressure for prey to defend themselves better (e.g., by running faster, improved eyesight, etc.) in response to which predators will develop better attacking strategies (such as stronger claws, faster diving, etc.).

This behaviour is often described as resulting in an arms race, driven by the will to survive. The nature of the competitive co-evolving system makes the algorithm search for an optimal strategy which cannot be beaten by any other. Thereby making the evolution converge towards this. The article presents the ideas of Angelina and Pollack, 1993, [3] regarding competitive fitness functions. These functions calculate the fitness of an individual through competition with other individuals. The individuals are game playing strategies, and the strategies which wins the game are awarded. The idea is that the systems competitive fitness will trigger a gradual improvement in the quality of the strategies [3] Thereby creating more robust strategies for predators and preys. The article presents a way to classify applications using competitive fitness into different categories based upon the pair-wise competition patterns they use.

In our approach we consider two populations of **strategies**, one for the Pacman and one for the ghosts. The fitness of a strategy is evaluated by selecting it and letting it play against strategies in the other population. One of our initial design choices were to evaluate according to the categories presented by [3].

All vs. All Basically let all strategies in population one compete against all strategies in population two and compute their fitness.

All vs. Best Let the fitness be evaluated by playing the best individuals in the other population.

All vs. Random Let the fitness be evaluated by playing a random subset of individuals from the other population.

The latter two of these have advantages in systems where the fitness computation is heavy, and time is limited - however they might have unforeseen side effects. We have exclusively used All vs. All evaluation in this project.

The competitive element reveals itself in the division of ghost strategies into the predator population, and the prey population consisting solely of pacman strategies.
When the ghost evolves a better strategy, the pacman is forced to counter-evolve to survive, thereby producing better strategies for finishing the game.
The two strategies can be viewed as each others opposites. The ghost attempting to catch the pacman and thereby preventing it from eating cheese, and the pacman attempting to avoid the ghosts, and collecting all the cheese.

5 Implementation

At top level, we use an EA to keep track of populations, genes, selection, crossover, mutation and evaluation. Our evaluations are based on the notion of static strategies throughout a game. When a pair of strategies are to play each other, we feed them to a **State Machine** that resolve the outcome. Output is optionally viewed via a **GUI**.

5.1 The EA

The EA in this project is an extension of an earlier project on simple EAs. It is called **MyEA**, and was produced in project 1 of the course in Topics of Evolutionary Algorithms E02. See the following URLs for more information:

www.evalife.dk/ToEC2002/
www.daimi.au.dk/˜clemen/TEA/

We have extended MyEA with Co-evolutionary concepts, and also made hooks for elitism in selection.
The MyEA implementation uses a fixed population size, two-sized tournament selection, Gaussian distributed mutation and arithmetic cross-over. The population is initialised at random, as described in [4].

We use population sizes between 20 and 50 individuals.

It was our initial idea to play around with different mutation, crossover and selection implementations, but the time scope for the projects only allowed us to do experiments with the selection type. The other experiments we had in mind, are appended to our list of topics for further examination.

5.2 The State Machine

Conceptually the machine can be interpreted as an arcade game that you can play with two opposing strategies. First the board is initialised, the players are placed at their respective start positions and the eatable entities are laid out. Each turn the machine evaluates each actors next move based on the fed strategies and the current state of the board (hence State Machine). The actors involved in one game run are not adaptive in the sense that they modify the strategy based on how the opponent plays. All modifications of strategies take place in the EA part of the system, and are based on $popsize_1 \times popsize_2$ completed runs.

Building blocks

The state machine contains a board consisting of tiles with content either `air, cheese, wall, drug, cage`. Add to this four ghost actors and one pac actor.

Move evaluations

Each time the machine needs to compute the next move of an actor, the end-tiles of the move choices `noop, left, up, right, down` are evaluated and ascribed a value based on a linear combination as shown in the following schemes.

Pac Gene	Value modifier
GREED	Distance to cheese
COURAGE	Distance to nearest ghost
DRUGHUNGER	Distance to drug
AGGRESSION	Distance to nearest ghost

Naturally, the AGGRESSION gene is only enabled during drugged mode in which COURAGE is disabled.

Ghost Gene	Value modifier
SPREAD	Distance to nearest ghost
BACKTRACK	Dist to previous positions
COURAGE	Distance to pacman
AGGRESSION	Distance to pacman
DEFENCE	Distance to nearest ghost

Here AGGRESSION and COURAGE are disjointly used depending on pac being drugged. The BACKTRACK gene also indicates that we implement an actor memory of past positions in order to regulate whether we want the actor (ghost) to stay in the same area. Note that ghosts are not allowed to `noop`.

A measure of nondeterminism is introduced in the movement evaluation, as the state machine randomly picks between moves of equal value.

Stale mates

A run ends in a stalemate when the time limit is exceeded without a conclusive victory. Generally we would like to avoid stale mates for two reasons: They are of little interest to either side, and they represent the worst case time for evaluation.
We deal with two types : gene induced stale mates and traps. The gene induced ones are unavoidable in early generations since we use

randomly initialised genes. Consider the obvious case where the pac COURAGE is -1.0, GREED is low and the ghost AGGRESSION is -1.0. The pac will prioritise running away from ghosts, but the ghosts will also stay away from the pac due to low aggression. The situation will quickly end in a pacman against the wall in one end of the board and the ghosts against the wall in the opposite end. Fortunately, the low fitness of these types of individuals ensure that they die out relatively quickly.
Trap stale mates are a bit more tricky. They occur when the state of the board lock the position of the actors in a grid. Perfectly fine strategies that have the potential of producing good results can end in a trap. Consider the case where the aggression of the ghosts is high, and they are also determined to spread out in the hunt for the pac. The pac is prudently scared of the ghosts, and tries to stay away from them. If we are unlucky we can end in the gridlock depicted in figure 1.

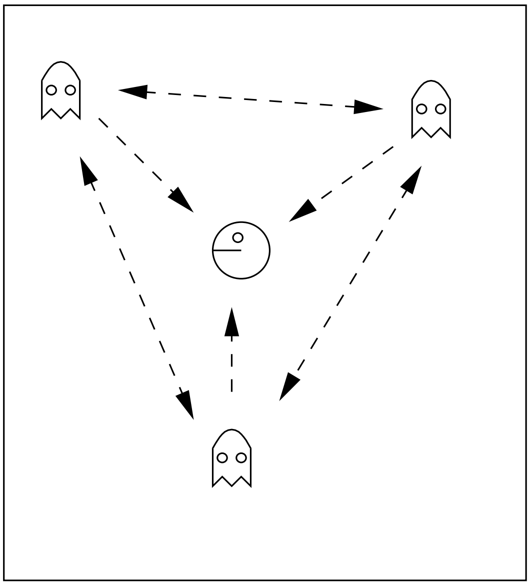

Figure 1: A locked scenario. The spread factor and relative positions prevent the ghosts from moving in on the Pac

In order to counteract traps, we use a pheromone trail idea inspired by [1]. We let the actors leave trails on the board, and if a specific actor's accumulated pheromone on a tile exceeds a preset threshold the machine forces a random move upon the actor. Pheromone dissipates at half the rate of deployment.

This mechanism causes a drastic cut in stalemate traps. Note that this also introduces more nondeterminism into the State Machine.

5.3 The GUI

An optional GUI can be created to display or interact with a run. Currently ASCII and OpenGL GUIs are available.
The state machine initialises the GUI and sends events to it after each turn.

6 Experiments and results

We decided on an suite containing 10 experiments. Each evaluation of two strategies take 1 second on average. This means that each of our experiments take 11 hours, and some of them take whole weeks. We have accumulated data derived from approximately one million games of Pac Man.
Unless otherwise indicated, the following experiments were performed with population sizes of 20, 20 generations and averaged over 5 runs.
The fitness function used for Pac strategies is

$$pfit = 10 \times c - s$$

Where c is cheese eaten and s is steps performed. Note that a `noop` counts as a step.
The ghost fitness is

$$gfit = l$$

Where l is cheese left on the board when the run is finished. Note that

$$l = 181 - c$$

expressing the inverse relationship between Pac strategies and Ghosts. With these functions in play, the Pac fit ranges from -1311 to 1629 and the ghost fit from 0 to 181. Note that we invert the sign of the fitness returned from the state machine in order to turn it into a minimisation problem understood by MyEA!

6.1 Expected results

We expect that the fitness of the Ghost strategies and the Pac strategies to both show a general tendency to converge on a value. However, we also expect to see mutually inverse behaviour, when the Pac fit increases the Ghost's decrease.

6.2 Base run

A data from a run of the system with the initial parameters will serve as a base for our result evaluation. We made a run with standard values and one with 100 generations and 30/30 popsize. The long result is seen in figure 2

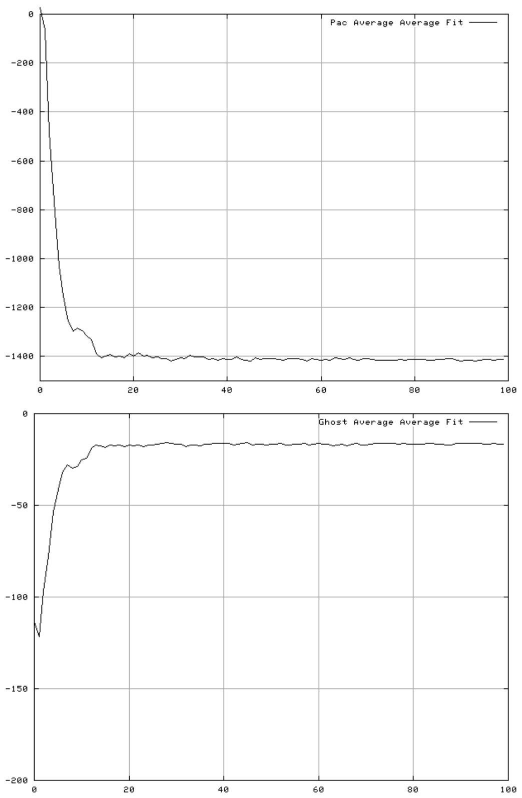

Figure 2: Average average fitness for Pac and ghost strategies on base run

Analysis

Interestingly it seems that initially the Pac is vastly inferior to the Ghosts, and then quickly gain ground ending up with an average fitness around 1400. Indeed this initial imbalance shows up in all our experiments! It can be explained from the choice of fitness function: Initially there are lots of strategies sure to end up in a gene-induced stalemate. The choice of $pfit$ heavily penalises the Pac strategies when a stalemate is achieved due to the step-dependent part. Ghosts are happy with a stalemate as long as cheese does not get eaten. Therefore they seemingly have an advantage, but the Pac population quickly evolves away from obvious stale-mates, and starts focusing on more greedy ways to play.

The final results are as follows:

Pac

Last Average Best Fit:	-1.441580E+03
Last Average Average Fit:	-1.411353E+03
Best Individual:	
Greed:	6.975330E-01
Courage:	-2.165817E-01
Drughunger:	6.533596E-01
Aggression:	2.534102E-02

The best pacman player is almost as drug greedy as he is cheese greedy. He runs away from the ghosts, but prioritises it lower than the hunger. Surprisingly the aggression level is low (but not negative) indicating that the worth of killing respawning ghosts is low.

Ghosts

Last Average Best Fit:	-2.106000E+01
Last Average Average Fit:	-1.676244E+01
Best Individual:	
Spread:	5.762271E-01
Backtrack:	2.476098E-01
Courage:	-2.546419E-01
Aggression:	8.353777E-01
Defence:	4.743499E-01

Interestingly, the best ghost strategy is for the pack to spread out. As expected the aggression value is high, and the courage is negative, but low. The defence gene is also comparatively high, and area defence is also a factor of some significance.

6.3 Low drug duration

In this experiment we set the drug duration to 20 steps instead of 40, effectively halving the time the pac can kill the ghosts.

The result is seen in figure 3

Analysis

The Pac settles on an average fitness value around 1000. Significantly less than the 1400 in the base run. This is as expected. Conversely, the ghosts settle around 56 as opposed to the base of 16. A noticeable improvement. The initial “Stale Slope” is evident here as well. The final results are as follows:

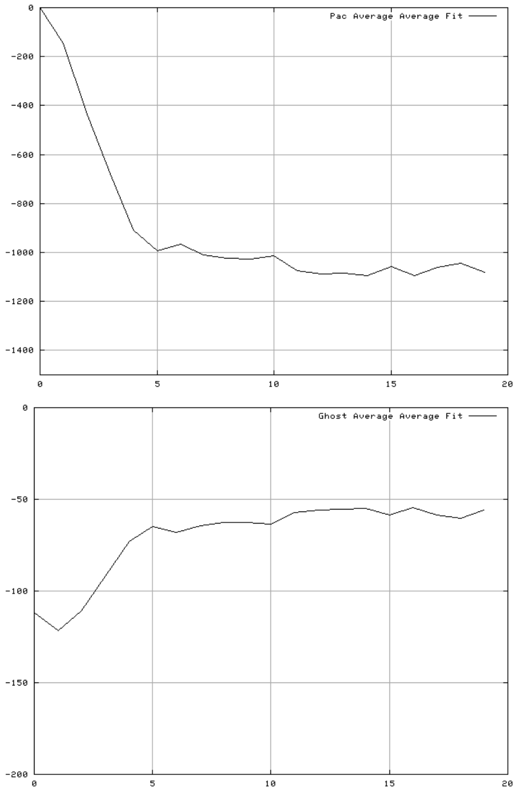

Figure 3: Average average fitness for Pac and ghost

Pac

Last Average Best Fit:	-1.205860E+03
Last Average Average Fit:	1.082631E+03
Best Individual:	
Greed:	4.876197E-01
Courage:	-3.707465E-01
Drughunger:	7.372013E-01
Aggression:	-3.327100E-01

In this scenario, the pac is better off seeking drugs at once, and being less aggressive, and less courages.

Ghosts

Last Average Best Fit:	-2.106000E+01
Last Average Average Fit:	-1.676244E+01
Best Individual:	
Spread:	1.879317E-01
Backtrack:	-3.156132E-01
Courage:	-1.156957E-01
Aggression:	7.745030E-01
Defence:	1.887251E-01

The ghosts go directly for the pac in a pack. Negative backtracking, low spread, low cheese defence.

6.4 Penalise stale mates

In this experiment we altered the fitness function to nullify both fits if there were stale mates involved. The experiment runs for 30 generations with 30/30 popsize. The result is seen in figure 4

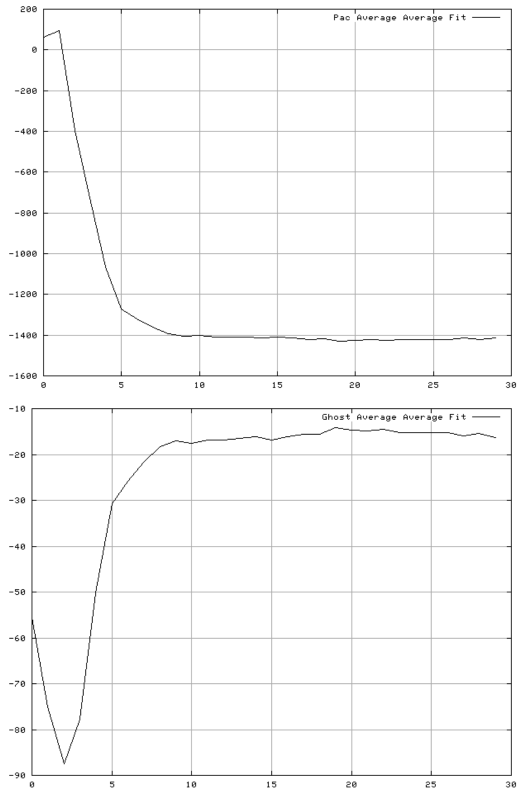

Figure 4: Average average fitness for Pac and ghost. Stale mates nullified

Analysis

This measure slightly improves the average fitness of both populations. Rewarding achieving an end goal has a beneficial effect on the evolution. Also, it seems that the Stale Slope is shortened, but this could also be due to the fact that the popsize is larger.

Pac

Last Average Best Fit:	-1.463220E+03
Last Average Average Fit:	-1.411564E+03
Best Individual:	
Greed:	4.758315E-01
Courage:	-1.575593E-01
Drughunger:	7.635541E-01
Aggression:	-6.702954E-02

In this scenario, the pac is actually better off seeking drugs than cheese. The aggression is slightly negative, meaning the pac uses the drug time to eat cheese.

Ghosts

Last Average Best Fit:	-2.142667E+01
Last Average Average Fit:	-1.622556E+01
Best Individual:	
Spread:	2.024142E-01
Backtrack:	6.126727E-01
Courage:	-1.071006E-01
Aggression:	6.981426E-01
Defence:	5.372010E-01

Note the negative courage and the unusually high backtrack.

6.5 Altering p_{mut} and p_{cross}

We have conducted four experiments with varying p_{mut} and p_{cross} for both populations. The initial values of 0.6 and 0.3 are as good as any of the others. The results show exactly the same behaviour as the base, Stale Slope and all. Graphs are available on request.

6.6 Two handicapping techniques

The first of these experiments lower the speed of which the ghost move toward the centre cage to regenerate. The second handicaps the Pac movement by making it slower.
The first result behaves as expected. The Pac settles on a slightly better value than the base run, and the ghosts on a lower one (15 vs. 16). The second one however, where the pac gets an extra turn every 4th step instead of every 3rd, is worth looking at:

Analysis

The Pac actually manages to end up with an average fitness convergence comparable with the base result! This means it actually finds a way to prevail even tho it moves slower!

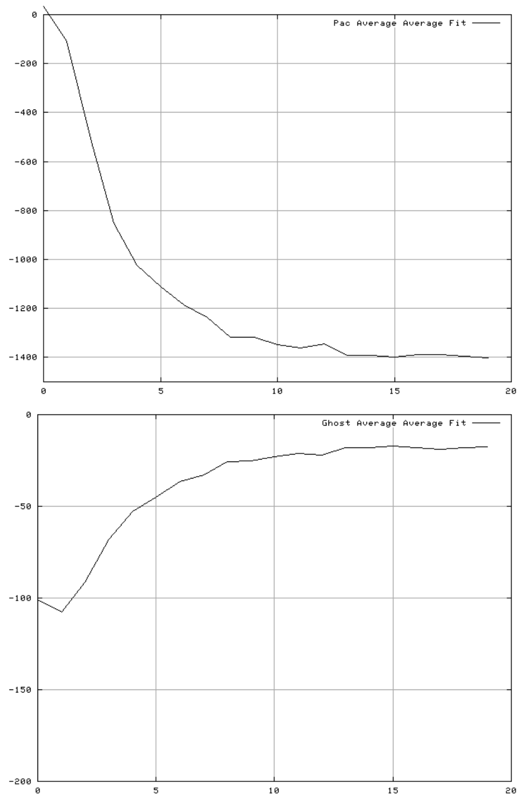

Figure 5: Average average fitness for Pac and ghost. Pacs handicapped

Pac

Last Average Best Fit:	-1.462500E+03
Last Average Average Fit:	-1.400404E+03
Best Individual:	
Greed:	4.417275E-01
Courage:	-2.671411E-01
Drughunger:	5.206147E-01
Aggression:	-1.455139E-01

Apparently lowering greed below drug hunger, and using the drugged time to eat cheese instead of kill ghosts can accomplish good results under these conditions.

Ghosts

Last Average Best Fit:	-2.376000E+01
Last Average Average Fit:	-1.757750E+01
Best Individual:	
Spread:	3.291351E-01
Backtrack:	4.376379E-01
Courage:	-5.081653E-01
Aggression:	4.546919E-01
Defence:	6.315349E-02

The ghosts counter the special pac strategy with high defence and backtrack values, and comparatively low aggression.

6.7 Elitist selection

In this final experiment we tried enforcing more elitist selection method in our EA. Every generation we save the one best individual, and make it impervious to selection, mutation and crossover.

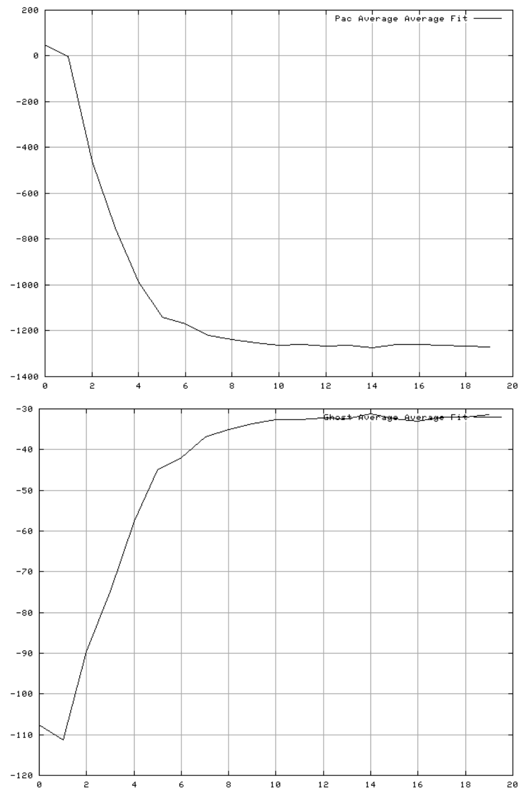

Figure 6: Average average fitness for Pac and ghost. Elitist selection.

Analysis

The average for Pac is significantly lower than the base run, but strangely enough, the best fit is just as high. This indicates that the last best fit, is better than the saved elitist individuals. Elitism does not benefit the Pacman.
Conversely the ghosts benefit, but again there is a larger difference between best fit and average fit than usual. Elitism does not work well with this system.

Pac

Last Average Best Fit:	-1.467740E+03
Last Average Average Fit:	-1.271144E+03
Best Individual:	
Greed:	6.557956E-01
Courage:	-2.249429E-01
Drughunger:	5.936066E-01
Aggression:	-1.779986E-01

Ghosts

Last Average Best Fit:	-4.144000E
Last Average Average Fit:	-3.141850E+01
Best Individual:	
Spread:	4.004247E-01
Backtrack:	-2.064016E-02
Courage:	-2.108645E-01
Aggression:	5.005739E-01
Defence:	1.293684E-01

7 Conclusion

The question to ask ourselves now is 'Have we evolved a better Pacman controller, and have we evolved a better Ghost controller?'
The results speak for themselves, the answer is definitely affirmative. However, we would have liked to have greater resolution on our experimental data. Larger population sizes, more generations and more runs to average over. The long base run took 5 days to complete, so some optimisations might be in order.
Nevertheless, the top individuals from the end experiments statistically perform well against all kinds of opposing strategies. The Co-evolutionary part of this project is well documented: Strategies respond to changes in environment and evolution in the opposing force. We would especially like to single out the aggressive change in behaviour when the fitness functions are changed to reward a non-stalemate end goal, and particularly the evidence that the Pacman finds a way to prevail even though it is handicapped with slower movement rates.

8 Further development

8.1 Choices of genes

We have chosen genomes based on our own ideas on how to play Pac Man successfully. Perhaps a wider or different selection of genes would result in more capable controllers.

8.2 Advanced game rules

In our implementation of the game, all actors are aware of each others position, and the content of the board. A more advanced game engine could include "first person" restrictions on the information contained in each actor, meaning that visibility would be restricted by walls.

8.3 EA related experimentation

We have based our implementation of the EA on the first project. It could be interesting to experiment with variations of selection, mutation and crossover models. This is however, out of the scope of this paper.

8.4 The competitive classification

As described in [3] there are various classification models to choose from when evaluating competitive fitness. One could try All vs. Best or All vs. Random models. We have restricted ourselves to using All vs. All, and thereby some solutions may have eluded us. Also, limiting the number of evaluations is a cardinal rule of Evolutionary Programming.

8.5 Specialisation of ghost strategies

And interesting advanced project could be changing our system so the State Machine is allotted a strategy for each ghost, instead of a "one for all strategy". This would introduce a cooperative dimension of Co-evolution into the ghost strategy population.

8.6 Code optimisations

There are several optimisations that could be done on the state machine in order to cut down evaluation time. Most trivially, the introducing a memory into the colouring algorithms would drastically reduce the need to search the board for distance to specific entities.

References

[1] V. Dorigo, M. Maniezzo and A. Colorni. *Ant System: Optimization by a Colony of Cooperating Agents*. Trans. on Systems, Man, and Cybernetics–Part B., 1996.

[2] Thiemo Krink and Rasmus K. Ursem. *Introduction to Evolutionary Computation*, chapter 1. (in prep.), 2003?

[3] Jan Paredis. *Coevolutionary Algorithms*, chapter C7.4. (), 1998.

Neural Network Training Convergence and Topology

Kim Pedersen

Abstract— **This work investigates the use of the Particle Swarm as a training algorithm for Neural Network controlled applications where a limited classification error is acceptable. Setting such an error limit shifts the focus from training accuracy to training speed and in that, on reducing the overall fitness evaluation time on the controller. The study explores a heuristic-based method inspired by Particle Swarms for finding better network topologies in terms of accuracy vs. training speed. A Particle Swarm extended with a "kick" mutation operator is used for evaluation of the topologies.**

It is concluded that the Particle Swarm is an effective training method for applications with such requirements. The heuristic-based topology search method however does not perform better than random search on the selected problems, and a further study should look in directions of less randomized methods.

1 Introduction

The Hydra Project, modular robotics

Neural Networks have been studied for use in areas such as classification, prediction and robot control. The present study originates from work in the Hydra project, which investigates modular robotics. Traditionally, robots may be controlled by Neural Networks for the purpose of doing stimuli-response tasks like avoiding obstacles, navigating towards light sources etc. The modular approach taken by the Hydra project involves coordination and social behavior between the modules, and the control mechanisms are therefore exprected to be much more complicated than simply using a Neural Network. The best way of controlling the modules has not yet been made clear, but one option is to have a Neural Network controller reading a combination of immediate stimuli such as the current neighbor connection configuration, along with state information about the current status of the common goal of the modules and the role of the current modules in the overall structure.

A current proposal for the morphology and structure of the Hydra robot modules is the 3-dimensional ATRON grid structure, in which each module is a sphere with one rotational plane and 4 connection points on each half sphere. Having this morphology the modules are able to move around on each other by series of neighbor connection, neighbor disconnection and rotational actions. A special feature of making a move is, that a module must ask one of its neighbors to rotate while the two modules are attached. The neighbor must then be able to determine which of its connection points that must be disconnected in order to perform the requested action. The behavior of a module navigating around on the structure is thus quite complex, and so it seems natural to evolve the behavior using evolutionary methods rather than directly programming the behavior.

A characteristic of the behavior is the regularity imposed by the grid structure. Once a module has moved to a new position it is in a new, discrete state. This regularity is different from the situation of an autonomous robot which is moving freely around, such one faces an infiniteness of situations to navigate in. The regularity of the ATRON grid structure means that if the module takes a wrong move it ends up in a new discrete state, from which it may be able to take a correct decision about the next move. Thus, the modules of the ATRON grid structure may be seen as an example of an application for which a certain error threshold is acceptable.

Characteristics of network topology

Feed Forward Neural Networks are known to be robust in the sense that different network topologies, i.e. different configurations with respect to number of nodes and connections, may be trained to accept the same set af data. When training a network, a vector of connection weights satisfying a minimal output error rate is searched for. With fewer connections in the network the search space is reduced. Reducing the network size does however reduce its classification abilities for the given problem [2], so there is a tradeoff between high convergence speed and low convergence value, i.e. low classification error.

The learning method that will be examined is the Particle Swarm. The Particle Swarm method has shown quick convergence in some cases and it is expected that this method would be suitable for quickly achieving a suboptimal solution with an error rate below some specified level.

Training experiments with a Genetic Algorithm will also be made. A general problem in using genetic algorithms for training and developing topologies of Neural Networks is the lack of obvious modularity. The ordinary arithmetic and n-point crossover operator mechanism require that the targeted problem posesses some modularity which makes it likely that the combination of parent individuals will produce offspring that are fitter. The current study includes experiments with a standard GA showing that simply calculating arithmetic averages of connection weights does not produce fitter offspring in general. This viewpoint is supported by [5].

Related work

Montana and Davis (1989) used a GA for evolving fixed weights in a fully connected NN [3]. The population size was 50, random initialization 1.0, 200 generations (i.e. 10000 evaluations). The crossover operator used is a special, node-based one, not the ordinary arithmetic and n-point operators. They conclude that the algorithm outperforms the classic Backpropagation method, but later work shows that the GA could not outperform better weight-adjustment methods such as quickprop.

Several different approaches to evolving Neural Network topology have been proposed and investigated over the time. The methods include pruning, GA-based evolvement of topology-generating grammars (parametric L-systems) and cell division methods. These methods, especially the grammar-based ones, are hard to justify intuitively as the topologies of parents and offspring usually are completely different.

A systematic approach which resembles the one we are investigating here, is the EPNet method by X.Yao [5]. Here, a population of randomly generated Neural Networks is maintained over a number of generations. The method uses no crossover operators, instead offspring is generated using up to four different mutation operators, which are Hybrid Training, hidden node deletion, connection deletion and connection / node addition. The parent to be used for offspring generation is selected using rank-based selection. Thus, the networks are evolved in parallel, in a competing fashion.

Apart from excluding the use of crossover operators the method differs from many others by evolving connection weights and network topology simultaneously. It is argued that the noise introduced by topology fitness evaluation makes it hard to determine the quality of a given topology. The noise arises from the fact that the true quality of a given topology requires intensive training, thereby increasing the topology fitness evaluation time dramatically.

The EPNet method is reported to have obtained good results on several problems, including the Cancer and Diabetes problems.

2 Experimental setup

The basis of the experimental setup is the Dendrite C++ framework by Thomsen,R. and Christensen, M.H [4]. The framework offers an open-source object-oriented implementation of feed-forward networks along with an implementation of the classic gradient-based Backpropagation training method.

The framework has been extended in order to be able to manipulate network topologies, being able to insert and remove connections and hidden nodes in a network.

Training methods

The targeted training method is a Particle Swarm implementation employing the basic velocity and position update methods as originally described by Kennedy [1]. Initially the basic velocity update formula was used, but experiments showed a tendency for the average par-

$v_i = \chi * (v_i + \phi_1 * (p_i - x_i) + \phi_2 * (p_g - x_i))$
$x_i = x_i + v_i$

Figure 1: The particle velocity and position update formulas.

```
randVal = randUniform(0,1);
pFactor = currentIteration /
                   totalIterations;
if (this != currentBest) {
  if (randVal > 0.9) {
    velocity *= 1 + 3*(1-pFactor);
  }
}
```

Figure 2: Pseudo code for the annealed kick operator.

ticle fitnesses to "oscillate". If the magnitude of the velocity update is too high, particles may "panic" and roam around in the neighborhood of the best performing particle without ever getting closer to it. Constriction Factor χ "slows down" the particles thereby reducing this effect and stabilizing the convergence of the swarm.

The basic particle position update operator has been enhanced with an annealed "kick" operator, shown in figure 2. The kick operator is basically the same as the mutation operator known from the standard GA. The idea is to introduce controlled fluctuations during the damped particle convergence caused by the constriction coefficient. During the convergence of the swarm towards a (sub)optimum, some particles are occasionally "kicked" away in order to find better optimas. If such a particle succeeds it becomes the best and the rest of the swarm will then follow it. If not, it will return converging towards the original leading particle. The kicking distance decreases linearly with the number of runs.

A Genetic Algorithm was also implemented and used in a number of runs for comparison with the performance of the Particle Swarm. The GA is real-encoded and implements tournament selection, elitism, standard arithmetic crossover and annealed mutation. The mutation operator targets individual genes.

```
Generate population with random networks
Evaluate each network
while (i < maxIterations) do
  for every network j in population do
    if not currentbest do
      adapt j
    fi
  od
  Evaluate each network
od
```

Figure 3: Topology training algorithm.

Using a GA with arithmetic crossover was expected to perform poorly on the problem of Neural Network training. Taking the averages of network weights between two parents generally produces offspring having very different weight values than the parents. This method of abruptly adjusting the connection weights differs significantly from e.g. gradient-based methods in which the weight values are adjusted gradually.

Topology evolving method

The method for automated topology search was inspired by the principles of the Particle Swarm. A swarm consisting of a number of randomly generated networks is generated and the performance of each network is evaluated. As with an ordinary Particle Swarm each network, except the currently best performing one, is repeatedly updated and evaluated again. The update is based on characteristics of the topology for the current best performing network in the swarm. The characteristics used is the number of connections and the number of nodes in the hidden layer. The topology update algorithm is shown in figure 3. The adapt function updates the topology of the Neural Network individuals according to the characteristics of the currently best performing individual. Different update policies (heuristics) may be employed here, two different ones are shown in figures 4 and 4. The idea behind Heuristic A is that smaller networks are preferred to larger ones. If the current best network has fewer hidden nodes or connections, a node or connection is removed from the currently processed network. If the

```
// Calculate the performance of the
// current network in relation
// to the best one.
// pIndex is a value in
// range [0;1], 1 is best.
pIndex = 1 - (ownTC.fitness -
                  bestTC.fitness);

if (bestTC.hiddenNodes <
              ownTC.hiddenNodes) {
  removeRandomHiddenNode();
}
else if (bestTC.connections <
              ownTC.connections) {
  removeRandomConnection();
}
else if (performIndex < 0.3) {
  addRandomHiddenNode();
}
else if (performIndex < 0.9) {
  addRandomConnection();
}
else {
  // Re-train network
}
```

Figure 4: Heuristic A pseudo code for Network Particle update.

```
if (bestTC.hiddenNodes <
ownTC.hiddenNodes) {
    removeRandomHiddenNode();
}
else if (bestTC.hiddenNodes >
ownTC.hiddenNodes) {
    addRandomHiddenNode();
}
else if (bestTC.connections <
        ownTC.connections) {
    removeRandomConnection();
}
else if (bestTC.connections >
        ownTC.connections) {
    addRandomConnection();
}
else {
 // Re-train network
}
```

Figure 5: Heuristic B pseudo code for Network Particle update.

best performing network is larger the performance index is used to determine whether a new hidden node or a new connection should be added to the currently processed network. The performance index is a measure of how well the currently processed network performs relative to the best performing network in the swarm. If the network performs very poorly a random hidden node is added, thereby extending the potential size of the network. If the performance is better but still not good, a new connection is added. Finally, if the network performs above a certain limit nothing is done. This approach may eliminate fitness evaluation noise arising from the fact that training error may classify slightly different networks arbitrarily.

Only one modification per network is made in each iteration. After updating all networks in the population all modified networks are evaluated by training with a Particle Swarm. Another topology update heuristic is shown in figure 5. The goal of this heuristic is to mimic the characteristics of the best performing network. During the process the best performing network might change, and so will the goal of the other networks in the population. Heuristic B thus resembles the classic Particle Swarm velocity update method more than Heuristic A does.

The topology search algorithm follows the principle of evolving Neural Networks incrementally like the method of pruning and the EPNet method [5]. This is contrary the the methods of evolving network grammars (L-systems), in which the offspring normally has a completely different topology than the parents in terms of the number of nodes and connections. The pruning and EPNet methods are more thorough, and expected to find better results eventually. However, with the proposed Network Particle Swarm method using the fast-converging Particle Swarm as training method, it is hoped that the road to good sub-optimal solutions can be shortcut.

Fitness function

Several error measures for Neural Network performance are used in the literature. The present one used is based on the rate of incorrect classifications in the dataset used.

$$error = \frac{incorrectclassifications}{totalsamples}$$

Two datasets are used. During the training of the network the evaluation function uses a training data set. When the search stops and a satisfactory solution has been found a different set of test data is used to give a final quality measure of the solution.

2.1 Experiments

Datasets

Since the Hydra project is not coordinated with the ToEC course, no mature ATRON module specification and simulator has been available for the current experiments. Instead, a classic theoretic problem and two well-known real-world problems have been used for the experiments:

- `XOR` : The classic XOR problem, which has only two inputs and two outputs, but is not linearly seperable, which means that at least one hidden layer is required.
- `Cancer` : A real-world classification problem concerning diagnosis of breast cancer. A tumor is classified as benign or malignant based on cell descriptions gathered by microscopic examination. The problem has 9 inputs, 2 outputs and the dataset has 699 examples. The data was originally from Dr.William H. Wolberg at the University of Wisconsin Hospitals, Madison.
- `Diabetes` : Another real-world classification problem diagnosing diabetes of Pima indians. Based on personal data and medical examinations it is to be classified whether a Pima indian is diabetes positive or not. The problem has 8 inputs, 2 outputs and 768 examples.

The current versions of the Cancer and Diabetes datasets were obtained from the Proben1 benchmarking set [2].

Since the topology search method uses the Particle Swarm as a basis, the characteristics of this method was investigated separately. The experiments investigate the significance of the various method parameters using the problems mentioned above. A number of different runs using the standard GA was also made.

Experiments with the topology search method were made using the knowledge about Particle Swarm characteristics found in the previous experiments. Experiments were mainly done on the Cancer problem.

3 Results

Training with a Particle Swarm

Several experiments with the three problems were made initially to get a feel of the different parameters. The swarm size used was mainly 20, however using a swarm size of 30 improved the performance of the XOR problem. Experiments with the phi factors showed that using the values $\phi_1 = 2.0$and $\phi_2 = 2.0$ as suggested by Kennedy [1], is actually not a bad choice.

Using the basic velocity update formula caused the particles in the swarm to "oscillate", so the constriction coefficient χ was required as a damping factor. Initial experiments used a value of $\chi = 0.6$ which stopped the oscillation and resulted in smooth convergence of the particles towards the final suboptimum. Further experiments using $\chi = 0.75$ according to the Constriction Coefficient formula proposed by Clerc [1] showed good results by converging towards lower values, however with a longer convergence duration.

The Particle Swarm converges pretty quickly so much can be obtained in only 50 update iterations on the swarm. However for obtaining results within a reasonable tolerance of 5% to 10% about 200 iterations is required. Further iterations does not improve the result much, so while the algorithm converges pretty quickly, it also finds and stays in suboptimal solutions, typically within a 5% to 10% classification error rate.

Figure 6 shows how convergence towards 23% error rate within only 20 iterations is obtained on a selected topology. The Particle

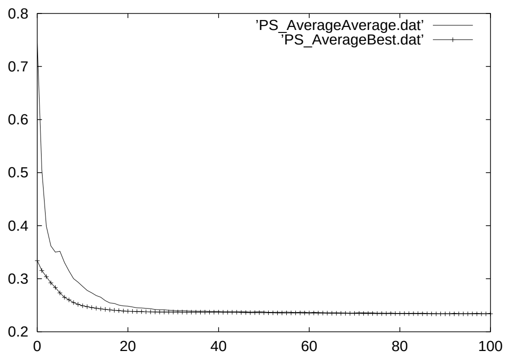

Figure 6: Using constriction factor.

Swarm converges towards a pretty high error rate of 20-25%. To improve this, the effect of the annealed kick-operator was investigated. Figure 7 shows the result of applying the kick operator with a 10% probability to the same conditions as was used in 6. The mutation introduced by the kick operator generally slows down convergence, but clearly improves the performance, bringing the average best convergence down to a 5% training classification error. The 5% mark is reached already at 160 swarm updates. The effect of network topology on training convergence speed is demonstrated by figure 8, which shows a Cancer problem run on a fully connected network (9-7-4-

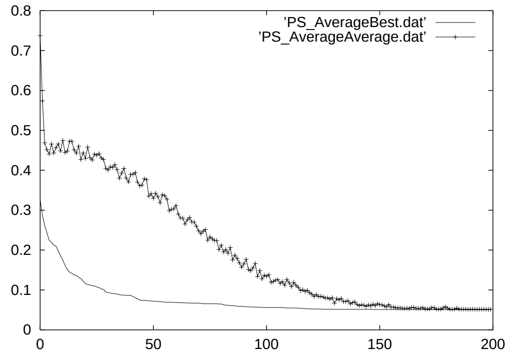

Figure 7: Adding the kick operator.

2). The average best values converge slower than with the selected network used in 7, but at a value of 7.1%. Although it is known that larger networks generally may be trained to perform better in terms of accuracy [2], the results showed that the Particle Swarm performs worse on such topologies compared to the selected ones. A longer run on the XOR problem

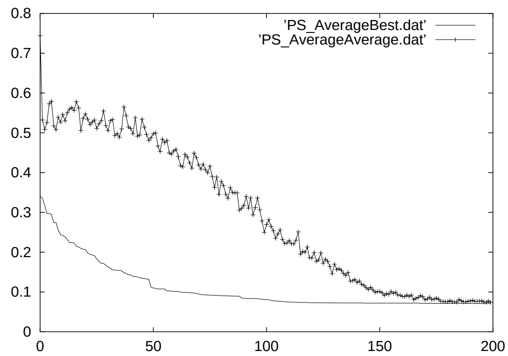

Figure 8: Training a fully connected network.

can be seen in figure 9, showing 500 swarm updates. The number of particles is 30 and the connection initialization limit is 10.0. The Average Best values converge at 2% classification error rate at about 320 updates, passing the 5% mark already at 190-195 updates. The average average values make an almost linear descend following the mutation annealing from just below 50% classification error among the initial, random population, towards the converging value around 500 updates.

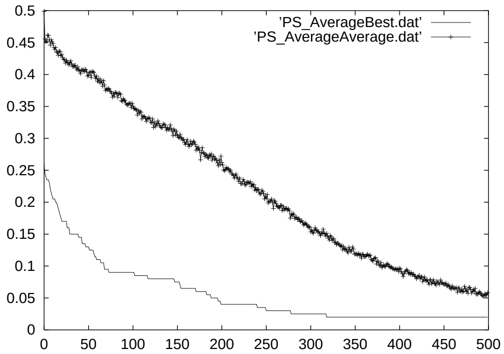

Figure 9: XOR problem.

Training with a Genetic Alorithm

To demonstrate the performance of a standard GA using arithmetic crossover for determining offspring weights, a number of experiments were run. The algorithm was expected to per-

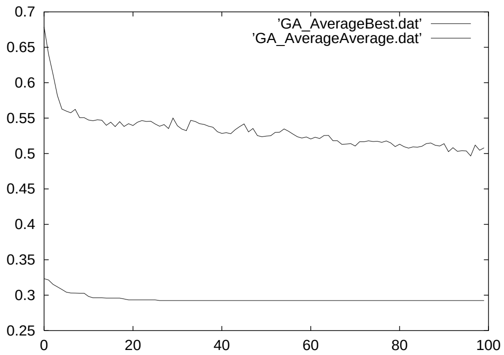

Figure 10: Training using a Standard Genetic Algorithm.

form poorly, which is indeed confirmed by the experiments, as seen in 10. The average average fitness value converges towards 50%, indicating that recombination does not generate better offspring. Instead, the population is constantly scattered around in the search space.

Searching for topologies

Figure 11 shows a run for the Cancer problem using Heuristic A. A Neural Network population of 20, 50 generations, each training Particle Swarm having 20 particles running over 50 iterations, and constriction factor, annealed kick and random initialization of connections as found in the previous experiments.

The average average classification error settles quickly at around 9.5%. This corresponds to the results found earlier on Particle Swarm training. It is however disappointing that the error never shows any tendency of converging towards the average best values. Another observation to be made is that in every generation there are networks performing extremely well, at around 3% error on average over the runs. Given the high average performance at 10% this indicates that the variation of the population is pretty high. Close inspection of the results showed that some networks perform at between 15% and 20% error rate, so the algorithm does not help finding the low-error rate networks any better than random search would.

Different performIndex threshold values for updating were tried, but the trend was the same. Figure 12 shows a run using heuris-

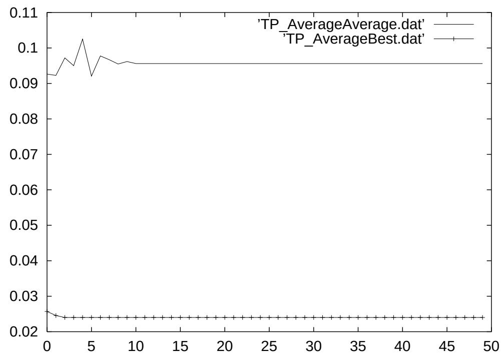

Figure 11: Using Heuristic A, Cancer problem.

tic B. The algorithm uses a population of 10 randomly generated networks, runs over 50 iterations, for every fitness evaluation a Particle Swarm with 20 particles running over 200 swarm updates is used. The results show the same picture as for Heuristic A, i.e. no convergence of the average average values towards the averege best values. The larger number of swarm updates in each fitness evaluation means that the average average level lies between 5% and 10%. The average best networks perform at just above 2%.

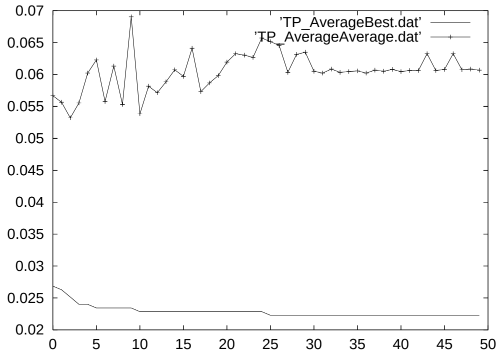

Figure 12: Using Heuristic B, Cancer problem

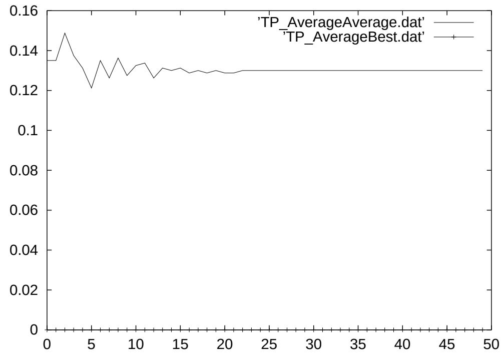

Figure 13: Using Heuristic B, XOR problem.

A run on the XOR problem using heuristic B is shown in fig.13. again, a population of 20 individuals are evolved over 50 generations, each fitness evaluation using a Particle Swarm with size 20 and 200 position updates. The random networks generated in this case were potentially rather large in relation to the problem size, up to 3 hidden layers were used.

The results show the same behavior as with the cancer problem in the previous experiments, i.e. the average average value settles close to the initual value, and the average best stays at a much lower level over the generations. In this case in fact, the average best value is zero in all generations! thus, for a problem of this size it is possible to find quite good networks by training a randomly generated population of 20 networks. The good results from this particular experiment also supports observations concluding that larger networks potentially perform better than smaller networks in terms of classification error.

4 Conclusions and further work

The overall conclusion from the experiments is that the Particle Swarm is a quite effective training method for Neural Networks in applications where a certain classification error threshold is acceptable. Introducing the annealed kick operator clearly improves the performance in terms of classification error, but slows down the convergence. Further work on the Particle Swarm used for Neural network training may be done in specific application areas such as modular robotics or other controller applications. The investigations in Particle Swarm mutation operators in combination with optimization of the velocity update parameters, could be a specific subject for further study.

It is however doubtful whether the Particle Swarm is worth pursuing as a fitness evaluation function for network topology evolution. The results of the experiments with the swarm-inspired topology search method shows that random search is almost just as good as using the two proposed search heuristics. In fact, from a population of 20 randomly generated networks, reasonably good results could always be found over a number of runs. Applying the search heuristics did not improve the average quality of the results however.

Network topology in relation to the given problem is still known to be signinficant, which was also shown by experiments on fully connected networks. Especially larger problems which real-world problems tend to be, benefit from optimizing network topologies, finding the essential connections. Decreasing the number of weights reduces the search space, thereby improving the convergence of a given training method. The issue however is to find the important connections in the network and leave less important ones out. The heuristic-based method used in the current study takes a randomized approach, which does not take the importance of the individual connections into account. The network updates are random, not calculated. While this principle may work for some kinds of problems it certainly does not work with Neural Networks, which do not possess the kind of modularity required. Another example of this is the non-modularity of the connection weights, as demonstrated by using a standard arithmetic GA for training. Further studies into network topology optimization should leave the randomized approach and instead pursue kinds of methods like EPNet ([5], [6]), which evolves topology and weights simultaneously and prunes the networks based on the importance of connections. Fitness evaluation noise is thus reduced because the population is not constantly retrained from scratch, and network changes are qualified, not randomized.

References

[1] J.Kennedy. Small worlds and mega-minds: Effects of neighborhood topology on particle swarm performance. *IEEE Transactions*, 1999.

[2] L.Prechelt. Proben1 - a set of neural network benchmark problems and benchmarking rules. 1994.

[3] M. Mitchell. *An Introduction to Genetic Algorihms*. MIT Press, Cambridge, Massachusetts, 1998.

[4] R.Thomsen and M.H.Christensen. *"Dendrite" - A C++ Neural Network Framework - A Quick Introduction*, 2002.

[5] X.Yao and Y.Liu. A new evolutionary system for evolving artificial neural networks. *IEEE Transactions on Neural Networks*.

[6] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 1999.

Evolving Palestrinian counterpoint with an EA

Søren Tjagvad Madsen

Abstract—**This paper presents a novel approach to make computer-generated Palestrina-style counterpoint. It uses an EA and a knowledge base as fitness function.**

The idea is tested with an implementation of a subset of the species counterpoint rules as defined by Knud Jeppesen. That involves melodic rules and harmonic rules. Experiments have been done with 1st, 2nd and 4th species, which results in acceptable "well-working" counterpoints.

The paper also suggests a way use the knowledge base to evolve music beyond the species counterpoint.

1 Introduction

Some approaches have been made on automatic generation of counterpoint. Farbood [1] describes a probabilistic way of doing first species counterpoint. The article also describes prior works on the subject. None of them uses an EA as the search algorithm.

Knud Jeppesen, a danish professor in music, has done an excellent book on Palestrina-style counterpoint, based on older studies of the subject. The book is describing the subject melody in great detail. Jeppesen has studied the works by Palestrina, and found a way to describe the music - which has resulted in the so called species counterpoint.

From the music he has derived a quite large set of "rules" or tendencies which the music follows more or less. Some of the rules are more strict than others, and some are just describing pleasant elements, which the style tries to incorporate.

2 Two part species counterpoint

The two part species counterpoint are all concerned about adding a voice to a given cantus firmus (fixed melody). The species counterpoint is an exercise in making larger musical pieces. The rules are fundamental for the larger scale music as well. There are 5 species for 2 part music. The cantus firmus (the given melody) is in all species in whole notes.

The tasks of the species are (in short) defined like this:

1. Add a counterpoint in whole notes. Consonance everywhere.
2. Add a counterpoint in half notes. Dissonances can occur on weak beats.
3. Add a counterpoint in quarter notes.
4. Add a counterpoint in half notes. Dissonances can occur on strong beats.
5. Add a counterpoint in free rhythm, according to the rules in the previous species.

Besides the species rules, which are mainly concerning harmony, there are some overall melodic rules which are also to be followed.

As indicated in the 5th species, the rules are building blocks for making larger music. They say something about the correlation between a pair voices. The rules apply in different settings, and together they deal with all cases.

3 The evolutionary algorithm

The evolutionary algorithm is a search algorithm. I will use it to search for the best counterpoint among many possibly which are correct, according to the musical style.

The idea is to maintain a population of counterpoints for the given cantus firmus. Each counterpoint is evaluated according to the implemented rules given by Jeppesen [2]. So I start with some random music and then evaluates it with the set af rules which apply. Only the best individuals are kept for the next epoch. They are recombined with each other and also mutated a bit before the next evaluation. The idea is to evolve a counterpoint which adheres to all the rules. This counterpoint should then hopefully be worth listening to!

3.1 Implementation details

I bring an overview of the algorithm:

- Population of counterpoints. The counterpoints are initialised with notes in the mode corresponding to the cantus firmus and with rhythmic values according to the wanted species.
- Evaluation:
  - Melodic contour and quality.
  - Species-rules according to the given cantus firmus.
- Tournamentselection. The best is kept in a safe place!
- Crossover: single- and doublepoint.
- Mutation: move a note up/down an interval (second, third or fourth) (in the correct mode)

To be more explicit: two counterpoints can be crossed in one or two points like the the crossover in traditional binary encoded genes (see [4]). For example a new (crossed) melody can consist of the start of the first parent and the end of the other parent.

3.2 The fitness function

The hard part of this algorithm (where to put the most effort) is to code the fitness evaluation.

I try to split the evaluation rules in two categories: Those concerning melodic contour and those concerning harmony. The melodic rules for the counterpoint are not correlated with the cantus firmus whereas the species rules are dealing with the two melodies sounding together.

I haven't implemented all rules in [2], but I began with the counterpoint for 2 part music. To give an impression of the rules, I will now introduce some.

Some fundamental rules

The counterpoint must not be too far away from the cantus firmus (optimally not more than an octave plus a third (decim)). Not all jumps are legal for the counterpoint for example the tritone and the seventh. In some modes, where the seventh in the scale is small (as in dorian and mixolydian), it should be altered in the cadence (leading tone). (For example c in d-dorian becomes c# in the cadence). The leading tone should be introduced stepwise or in a descending third – never by jumping up to it.

First species rules

Only consonant intervals are allowed (third, fifth, sixth, octave, decim). Beginning and ending: only true consonance (fifth, octave). Unison only allowed in first and last measure. Parallel and hidden parallel fifths or octaves (where both voices move in the same direction into the interval) is not allowed. The cantus firmus and the counterpoint are not allowed to move in parallel thirds or sixths for too long time (not more than 4 whole notes), since the counterpoint then loses it's individuality. If both voices jump in the same direction, none of them must jump more than a fourth (except an octave).

The most beautiful thing in this species is countermovement. It is not a requirement, but one should try to do so whenever possible.

Melodic rules

The melodic rules are not so important when the music moves slow as in first species (whole notes). But if possible, we try to keep them. I have implemented a few melodic rules. The melody should not shift direction all the time. If it shifts direction 4 times in a row it is an error. Larger steps should come before smaller when the melodic curve is going up, and smaller before the larger when the curve is going down. (There are some allowed special cases.) The top note (climax) should be unique, and should not be in the beginning of the phrase or at the very end. It is a good idea to have the climax about 1/3 before the end of the phrase.

Second species rules

Dissonance is allowed on weak beats, but only if they are introduced stepwise. Consonance is allowed everywhere. Unison only in first and last measure. Parallels between the strong beats should be avoided.

Fourth species rules

Dissonance (the intervals fourth (4) and seventh (7)) is allowed on strong beat, if it is prepared from the previous measure as a consonance (3, 5, 6, 8 or 10) and if it is resolved on the next weak beat one step down to a nontrue consonance (3, 6 or 10).

If there is no dissonance on the strong beat, the rules from 2nd species are used. Since 2nd and 4th species both are about putting two half notes to each whole note of the cantus firmus, they are naturally connected.

4 Experimental results

The set of rules I implemented concerning 1st species is quite complete. First species counterpoint is not a hard problem, so the computer did not take long to find a solution. And for each new rule added, there was no problem in fulfilling it.

All errors discovered was punished with the value 1.0, so it was easy to count the errors. The soft rules like the countermovement rule was on the other hand given 0.1 in punishment, everytime there was a non-countermovement. Figure 1 shows an example of an evolved counterpoint with fitness 0.6 in this setting. All harmonic and melodic rules have been kept, but there are 6 non-countermovements. I have put the sounding intervals between the notes.

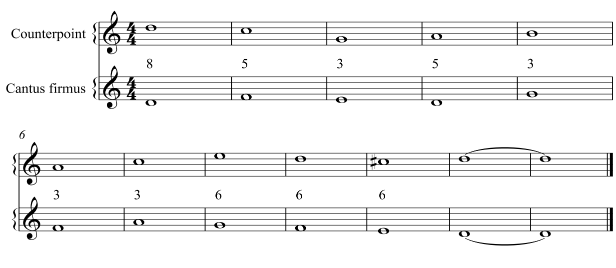

Figure 1: First species counterpoint

With less restriction on the cadence (see below), the algorithm found a counterpoint with only 2 non-countermovements.

The next experiment was about making music in half notes: the 2nd and 4th species. Now there is the possibility of making dissonances, if they are treated properly. So I run the algorithm, but only consonant intervals had been found which is just as correct, but a little boring, and certainly not the point of doing the species. It was of course easier for the algorithm to fulfil the simplest rules (make consonances everywhere) in stead of making dissonances which depends on the notes before and after it. So I had to trick the algorithm into making the fun stuff. I introduced a small reward in the evaluation, everytime the algorithm found a correct treated dissonance. I spent some time adjusting the size of the reward. It seemed that it should be smaller than the standard error punishment, so a correct dissonance can not compensate for some other broken fundamental rules (then it is better to keep it correct).

A similar problem is the making of the cadence. We like to have the leading tone introduced in the next to last note (the # altered note in both figures). It also requires special treatment in introducing, so again a reward was necessairy.

I ended with the reward value -0.1 for every correct treated dissonance and leading tone. With this setup the program was able to produce the counterpoint shown in figure 2:

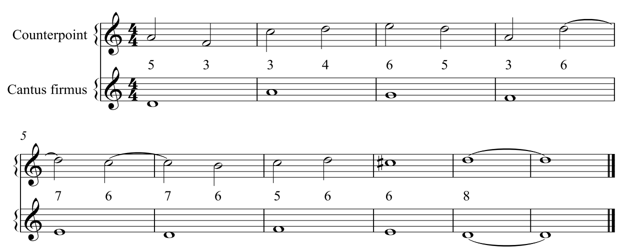

Figure 2: Second and fourth species counterpoint

There are features from both 2nd and 4th species. Remember that the intervals: 2, 4 and 7 are dissonant. 5 and 8 are true consonances and 3, 6 and 10 are just consonances. The fourth in measure 2 is introduced stepwise on the week beat and between 2 consonances (2nd species). The sevenths in measure 5 and

6 are prepared from a consomance in the preceding measure and are resolved to a non-true consonance (4th species).

It is starting to get a little more difficult to fulfil all the restrictions. Jeppesen points out that the melodic rule of the order of large and small steps loses it's importance when dealing with 4th species, since you locallly hear the syncopated dissonance a lot more than the melodic line. The given example however fulfills all the introduced rules. And still the running time is within a minute. For the experiments I used a population of 1500 for 60 epochs.

I must admit that I have only tried the algorithm on three different cantus firmus and in one mode (d-dorian), but I very much expect the algorithm to behave similarly on other melodies.

In my implementation I still need to take care about some special events, which are allowed in the music. But these are mainly minor changes or updates to the program. One of them is concerning the treatment of upbeat. More modes (than dorian) can easily be implemented. But more fundamental: I still miss something to take care of the tritone (augmented fourth / diminished fifth – most dissonant and therefore forbidden) interval which for the moment do not recieve special treatment (which it should). I must read the chapter on flat alterations to avoid the tritone interval.

4.1 The musical quality of the counterpoints

The music evolved is clearly acceptable – it adheres to the rules. I still need some melodic rules to implement. I can get an advantage from implementing rules dealing with sequences (repeats of musical fragments) and consequtive jumps in triads. Both are considered to deteriorate the counterpoint.

The melodic lines are sometimes a bit without direction, which they should indeed not be. A couple of preference rules could be added, to deal with that kind of problems.

5 Perspectives and future ideas

The species rules are, as we have seen, building blocks for evaluation of music with mixed rhythm values (like real melodies). So with some rules for quarter notes and for combinations of notes with different rhythm values it is possible to make 5th species counterpoint.

In 5th species it is necessary to introduce rhythmic operators for making mutations in the counterpoint. So far I have only used pitch mutation since the rhythm was fixed. They should not be totally random, but rather suggest som standard rhythmic licks from the style! For example mutations for making one whole note into one dotted half note and a quarter note. Pitch mutation should continue as before.

The next step is then to make free two part music. By then, it is possible to make different rhythm values in bots parts. We do not have the cantus firmus here, but still our species rules can be used – one of the voices is simply locally considered as the cantus firmus.[14]

My idea is to make a unit in the program, which looks at one measure at a time, and finds out which set of rules should be used for the evaluation, and then evaluates it.

The search might be somewhat harder when none of the voices are fixed, and both of them can be altered or mutated. The search space is then growing consederably, but then again there are a lot more correct solutions.

The next step in Jeppesen is 2 part imitation-style. We need to introduce a notion of imitation. In the simple style the imitations should be the excact reproduction of the theme, but later one can experiment with variation / mutation of the theme.

And then three part music and four etc. There are also rules for how to put lyrics on the music.

Another experiment which could be nice to do, is to evolve a cantus firmus. So far I use the ones in Jeppesens book, but with the melodic rules and a little more, it should be possible to evolve a wellformed melody which could be

[14] So far I have not talked about what happens when the counterpoint is below the cantus firmus, but the rules are exactly the same: same dissonance treatment, melodic lines etc.

used as cantus firmus in a species-piece.

All through the experiment I have been very discrete in the calculation of the evaluation. EA's usually perform very well on a more fuzzy evaluation function, but I haven't found any reasonably (musical) way of grading music which is almost correct (more than already described by the punishment and reward approach). Since some of the rules are "either-or" I don't think it is approproate to make those fuzzy, but I could experiment more with grading the more soft rules, when the algorithm meets larger challenges and not all wishes can be met.

6 Conclusions

This project shows how to use an EA to search for counterpoints. Farbood [1] describes a very slow, but quite complete system for making 5th species counterpoint for up to 6 voices made by Schottstaedt [4]. It seems that an EA would be able to perform a bit faster than the recursive method of that system.

I haven't had a second persons opinion of the quality of the counterpoints. The quality of the correct counterpoints do vary, but I think that the two examples brougt here are quite good. Some rules yet need to be implemented, so I have hopes. The posibility of extending the system are quite clear to me. The real challenge is to adjust the musical parameters with the grading of the fitness function.

References

[1] Mary Farbood Bernd Schoner. Analysis and synthesis of palestrina-style counterpoint using markov chains. *Proceedings of International Computer Music Conference. Havana, Cuba*, 2001.

[2] Knud Jeppesen. *Kontrapunkt, Vokalpolyfoni.* Wilhelm Hansen, Copenhagen, 4th ed edition, 1968.

[3] Thiemo Krink and Rasmus K. Ursem. *Introduction to Evolutionary Computation*, chapter 1. (in prep.), 2003?

[4] William Schottstaedt. Automatic counterpoint. *In: Current directions in Computer Music Research*, 1989.

Basic PSO versus Multi Swarm PSO

André Højgaard Jensen and Sune Kristensen

Abstract—
Particle Swarm Optimisers (PSO's) have some advantages compared to algorithms like Evolutionary Algorithms (EAs) eg faster convergence. However PSO's have a tendency to get caught in local optima. The goal of this paper is to investigate if it is possible to avoid this problem by using more than one swarm and apply criteria for when to move swarms stagnated in local optima to a new location in the search space or spread them.

1 Introduction

The PSO algorithm was originally introduced in [3]. In [5] experiments with multi swarm PSO's are performed. However we would like to bring their investigations further. Their investigation was to see whether a PSO with more than one swarm performs better than a basic PSO with only one swarm. Their strategy was to split the swarms of 20 particles into up to six different swarms having a few particles per swarm. In our eyes this approach did not seem optimal because, quite a few of articles points out that a swarm should contain around 20 particles to perform optimally. On top of that no actions are taken to prevent two or more swarms to clash together and become one. This was the inspiration to the following experiments with multi swarm PSO's. As mentioned above the basic PSO have one big weakness, it has a tendency to get stuck in a local optimum. By applying several swarms it is the assumption that the following advantages can be obtained:

a) By comparing the best fitness values in several swarms, and if one of the swarms has stagnated it is possible to determine, if this swarm has stagnated in a local optimum. Then the swam can be moved to a new area of the search space. This is further discussed in the section 'Model'

b) By having several swarms and keeping them away form each other, you can preserve higher diversity than with one swarm. The reason for this is that it is possible for one swarm to fine tune while another swarm are performing a global search.

So one of our main challenges when working with more than one swarm occur when two or more swarms are trying to fine tune on the same local optimum. Here we have to keep them separate to have the effect of more swarms and still not keep them so much separate that at global or local optimum can hide in between the swarms. So how close is that? Of course that is problem dependent.

2 Model

The basic Particle Swarm Optimiser consists of a group of particles witch moves around in the search space. Each particle has a velocity vector $\vec{v}$, a position vector $\vec{x}$, a best position vector $\vec{p}$ which is the best position seen by this particle so far, and a reference to the best position discovered by the whole swarm. The position of a particle at time $t+1$ is given by

$$\vec{x}(t+1) = \vec{x}(t) + \vec{v}(t+1) \tag{4}$$

Each particle are influenced by two forces each time a new velocity vector are calculated, that is the best point seen by this particle and the best point seen by all particles of the swarm. The new velocity $\vec{v}(t+1)$ are calculated by the following formula, which seems to be efficient for PSO's in general [7]:

$$\vec{v}(t+1) = w \cdot \vec{v}(t) + \phi_1 \cdot (\vec{p} - \vec{x}) + \phi_2 \cdot (\vec{n} - \vec{x}) \tag{5}$$

$\vec{n}$ is the optimum position for the best particle in the swarm so far. Notice here that when $\vec{x}$ is very close to $\vec{p}$ and $\vec{n}$ then the $\vec{v}(t+1)$ will only change very little. ϕ_1 and ϕ_2 is uniform randomly generated numbers from $[0..\phi]$, $\phi \in [0..2]$. The use of ϕ factors war introduces in [1].

The initia weight w is controlling the impact of the old velocity $\vec{v}(t)$ on the new velocity

$\vec{v}(t+1)$ for each swarm

$$w = (w_p - C)\frac{generations_{sw_i} - iteration_{sw_i}}{generations_{sw_i}} + C \quad (6)$$

w_p is particle weight, C is constant fixed at 0.4. $generations_{sw_i}$ is the total generations for swarm number i and $iteration_{sw_i}$ is number of generations done by swarm i. This implies that w will start out with a value of w_p and end with a value of C i.e that w is decreasing over time until hitting lower limit C.

The initia weight w in the velocity formula insures that the velocity are slowed down as the number of generations increases, and hereby going from global search to fine tuning.

A particle in the multi swarm PSO are represented in the same way as in the basic PSO and on top of that each swarm has an index to the best particle seen by all swarms. The swarm also contains a sample of the best fitness for this swarm 10 generations ago and a stagnation factor. These numbers are used to discover if the swarm has stagnated in a local optimum.

Pseudo code for the multi swarm PSO algorithm is illustrated in 1.

```
begin
  initialise
  for each swarm do
    while(not i=generations) do
    evaluate
    alter
    move
    i++
    od
  od
end
```

Figure 1: Pseudo code for the multi swarm PSO

`initialise`: initialisation of the particles and swarms.
`evaluate`: fitness is calculated for each of the particles. $\vec{n}$ is updated.
`alter`: here we handle the problems with stagnation and keeping the swarms away form each other. The velocity vector $\vec{v}$ is calculated here. The vector is limited by a maximum velocity on each coordinate. An alternative is discussed later.
`move`: The new optimal position vector $\vec{p}$ for each particle is calculated here.

The main difference between the multi swarm PSO and a basic PSO is that the inner loop is done for each swarms instead of just one swarm and that `alter` is responsible for controlling what happens in case of stagnation and when swarms are getting too close.

We have been working with two different PSO's based on the same multi swarm model. They differ in behaviour when a swarm is stagnated in a local optimum or two swarms are too close to each other.

In the first, which we call MusSPREPSO, the stagnated swarm are spread randomly over the whole search space in the same way as when initialising the swarms. For this swarm $iteration_{sw_i}$ will be set to zero and $generations_{sw_i}$ will be adjusted to generations left. This also happens if two swarms are too close and stagnated. Then the swarm with the worst fitness calculated from $\vec{p}$ are the one to be spread.

In the second, which we call MusMUPSO, a small mutation is made to the position vector for all particles in this swarm. The same randomly chosen index in the position vector for all swarms are mutated a little, which has the effect of moving the whole swarm. This happens using the same criteria for stagnation and 'are to close' as with MusSPREPSO.

But how can we determine weather two swarms are too close and weather a swarm is stagnated?
We say that two swarms are too close, if the positions of the best particle from each swarm are too close. Too close we define by "length of search space" $\times$ " some factor". The size of "factor" is problem dependent and are discussed in "experimental settings". Our stagnation criteria is that a swarm should improve its fitness more than a specific percentage over a number of generations (also discussed in experimental settings).

The neighbourhood type of the swarms are 'wheel' as suggested in [2].

3 Experimental settings

We have compared the performance of the two multi swarm PSO's against a basic PSO on four standard numerical minimisation benchmark problems. The first two are unimodal and the last two are multi modal.

Generalised **sphere** function:

$$f_1(\mathbf{x}) = \sum_{i=1}^{n} x_i^2$$
$$-100 \leq x_i \leq 100$$

Generalised **Rosenbrock** function:

$$f_2(\mathbf{x}) = \sum_{i=1}^{n-1} (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$$
$$-100 \leq x_i \leq 100$$

Generalised **Griewank** function:

$$f_3(\mathbf{x}) = \frac{1}{4000} \sum_{i=1}^{n} (x_i - 100)^2 - \prod_{i=1}^{n} cos(\frac{x_i - 100}{\sqrt{(i)}}) + 1$$
$$-600 \leq x_i \leq 600$$

Generalised **Rastrigin** function:

$$f_4(\mathbf{x}) = \sum_{i=1}^{n} (x_i^2 - 10cos(2\pi x_i) + 10)$$
$$-5.12 \leq x_i \leq 5.12$$

$\mathbf{x}$ is an n-dimensional real-valued vector and x_i is the ith element of that vector.

We tested the three PSO's on the four functions in 10, 20 and 30 dimensions. The multi swarm PSO's used two swarms when not anything else is mentioned. We used population size of 20 and 100000 *time steps* where *time steps = generations · number of swarms*. We used a *stagnation factor* of 0.99 which implies that a swarm stagnates if it have not improved more than 1% over 10 generations. We said that two swarms was too close when the distance between their best particles was less than 1/10000. All results are averages of 50 repeated runs. We used particle weight w_p of 0.7 which means that initia weight w was starting at 0.7 and ending at 0.4. We had a *maximum velocity* of 25. The multi swarm PSO's let a swarm stagnate while being too close to another swarm before spreading or moving it.

4 Experimental results

The main purpose of our tests was to investigate if multi swarm PSO's gives better results than basic PSO's. We make sure that the basic and the multi swarm PSO's gets an equal number of time steps.

On top of that it will be practical to know what the optimal number of swarms is. So we will investigate that as well.

Furthermore we would like to investigate which multi swarm PSO strategies that works best. We have developed two different kinds of multi swarm PSO's: MusSPREPSO, the random spreading algorithm and MusMUPSO the mutation algorithm.

The most important results of our testing is illustrated in the following table and figures. In the table average fitness and standard deviation are shown. Notice that in the figures the x-axes shows the number of time steps not the number of generations.

The goal of the first two tests which is shown in figure 2 and 3 is to see what number of swarms is optimal. Figure 2 shows that one swarm is optimal on the sphere problem, two swarms is second best etc. Figure 3 shows that two swarms is the optimal on the Rastrigin benchmark problem. Similar tests on Griewank and Rosenbrock gives the same impression. On the background of these tests we decided to use two swarms when testing the multi swarm PSO's.

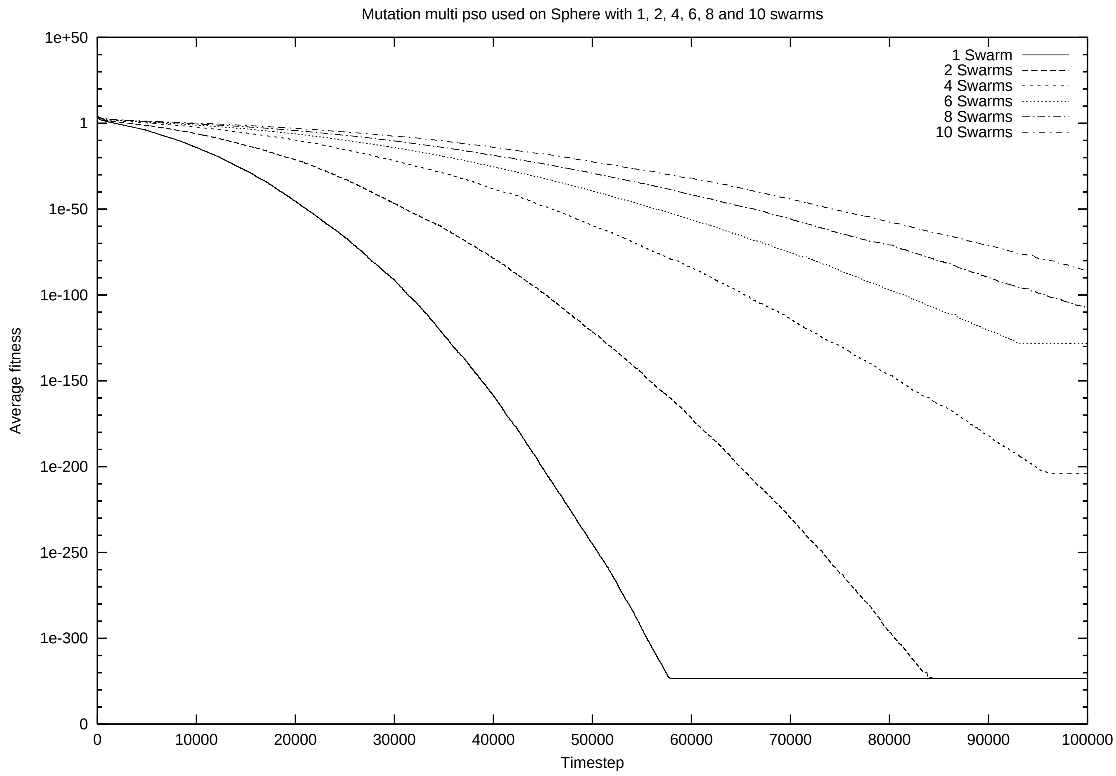

Figure 2: Graph comparison of different numbers of swarms on Sphere 30 dim

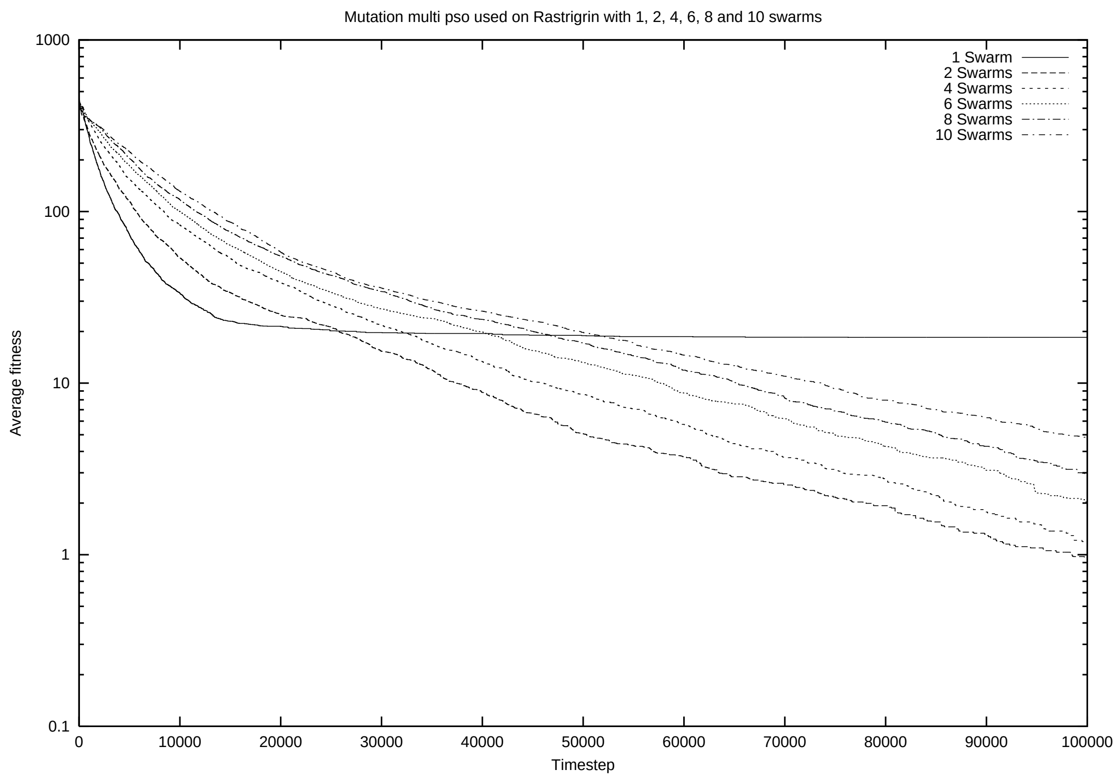

Figure 3: Graph comparison of different numbers of swarms on Rastrigin 30 dim

Dim	10	20	30
		Sphere	
bPSO	0 ± 0	0 ± 0	$4.94 \cdot 10^{-324} \pm 0$
MusSPREPSO	0 ± 0	$4.94 \cdot 10^{-324} \pm 0$	$4.94 \cdot 10^{-324} \pm 0$
MusMUPSO	0 ± 0	0 ± 0	$4.94 \cdot 10^{-324} \pm 0$
		Rosenbrock	
bPSO	0.78 ± 0.77	8.13 ± 8.06	17.79 ± 17.61
MusSPREPSO	0.67 ± 0.66	6.87 ± 6.81	24.58 ± 24.35
MusMUPSO	0.53 ± 0.51	4.33 ± 4.28	13.65 ± 13.52
		Griewank	
bPSO	0.058 ± 0.057	0.047 ± 0.046	0.014 ± 0.014
MusSPREPSO	0.051 ± 0.050	0.029 ± 0.029	0.012 ± 0.012
MusMUPSO	0.009 ± 0.009 test b	0.012 ± 0.011	0.008 ± 0.007
		Rastrigin	
bPSO	1.87 ± 1.85	7.10 ± 7.33	15.82 ± 15.66
MusSPREPSO	2.14 ± 2.12	9.31 ± 9.21	21.90 ± 21.68
MusMUPSO	0 ± 0	$3.17 \cdot 10^{-8} \pm 3.17 \cdot 10^{-8}$	$1.82 \cdot 10^{-11} \pm 1.81 \cdot 10^{-11}$

Table 1: Overview over the most important test runs.

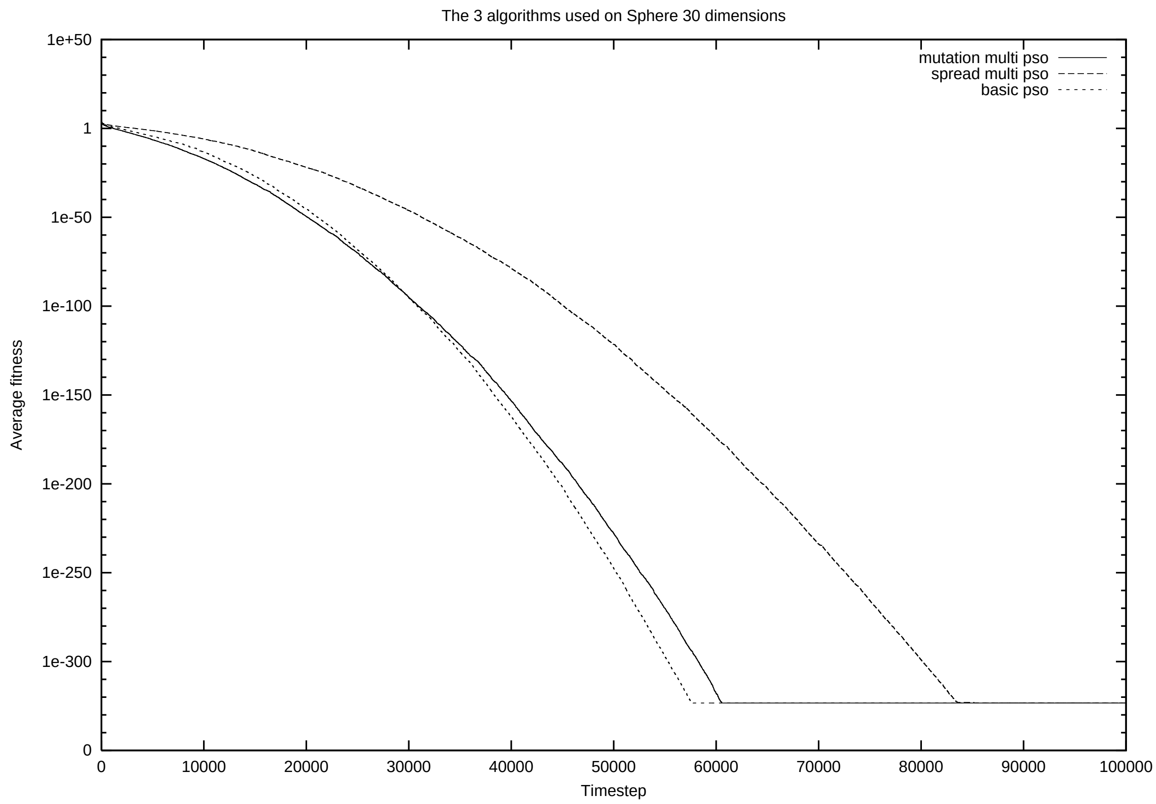

Figure 4: Graph comparison of 3 algorithms on Sphere 30 dim

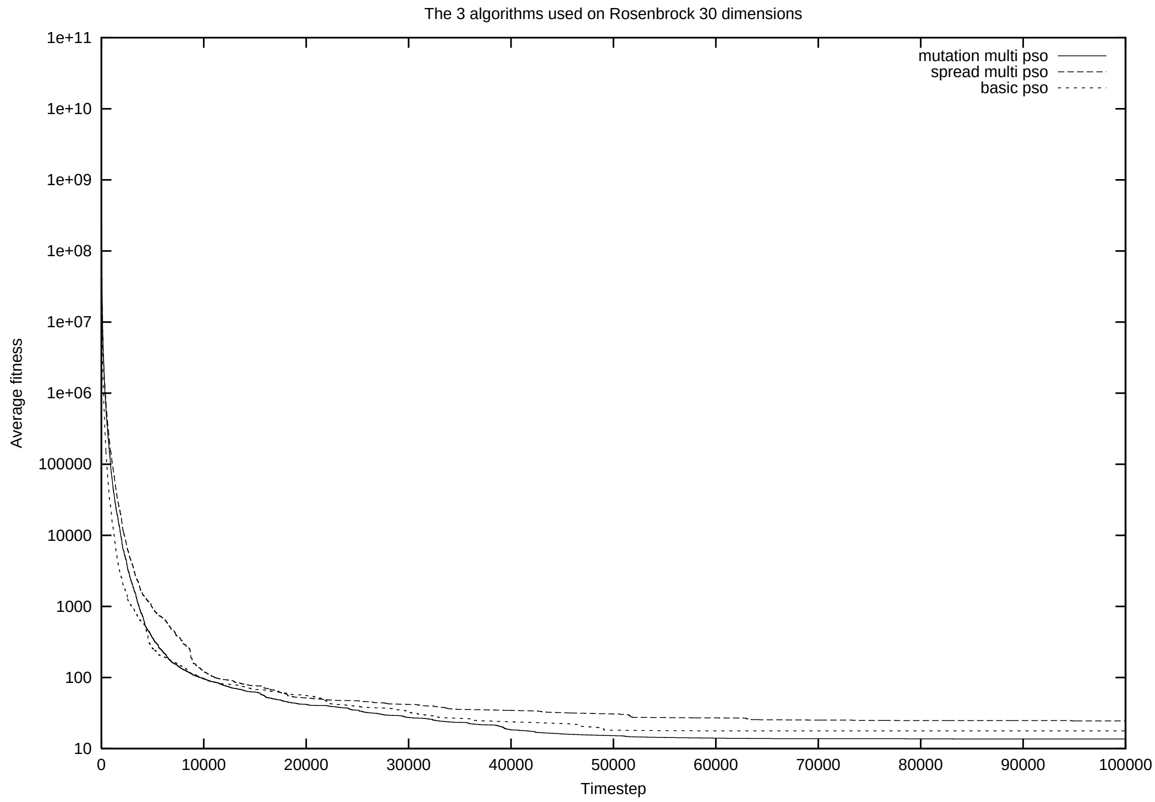

Figure 5: Graph comparison of 3 algorithms on Rosenbrock 30 dim

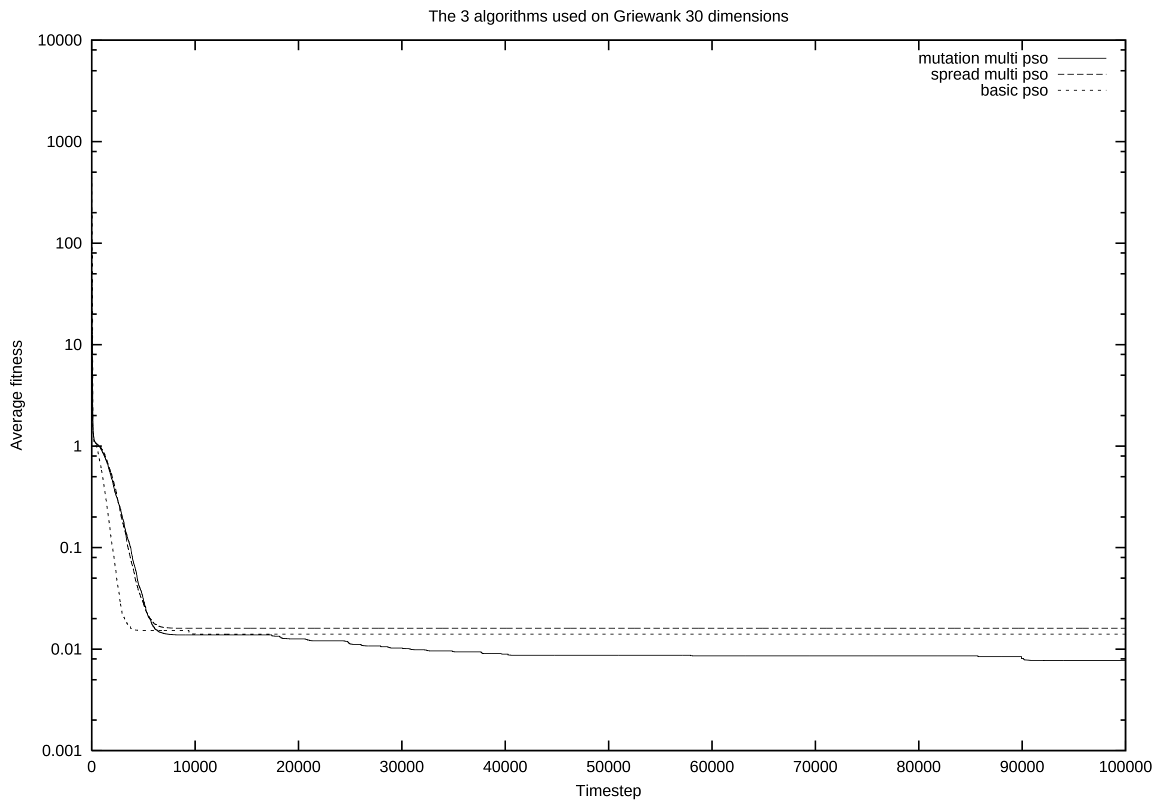

Figure 6: Graph comparison of 3 algorithms on Griewank 30 dim

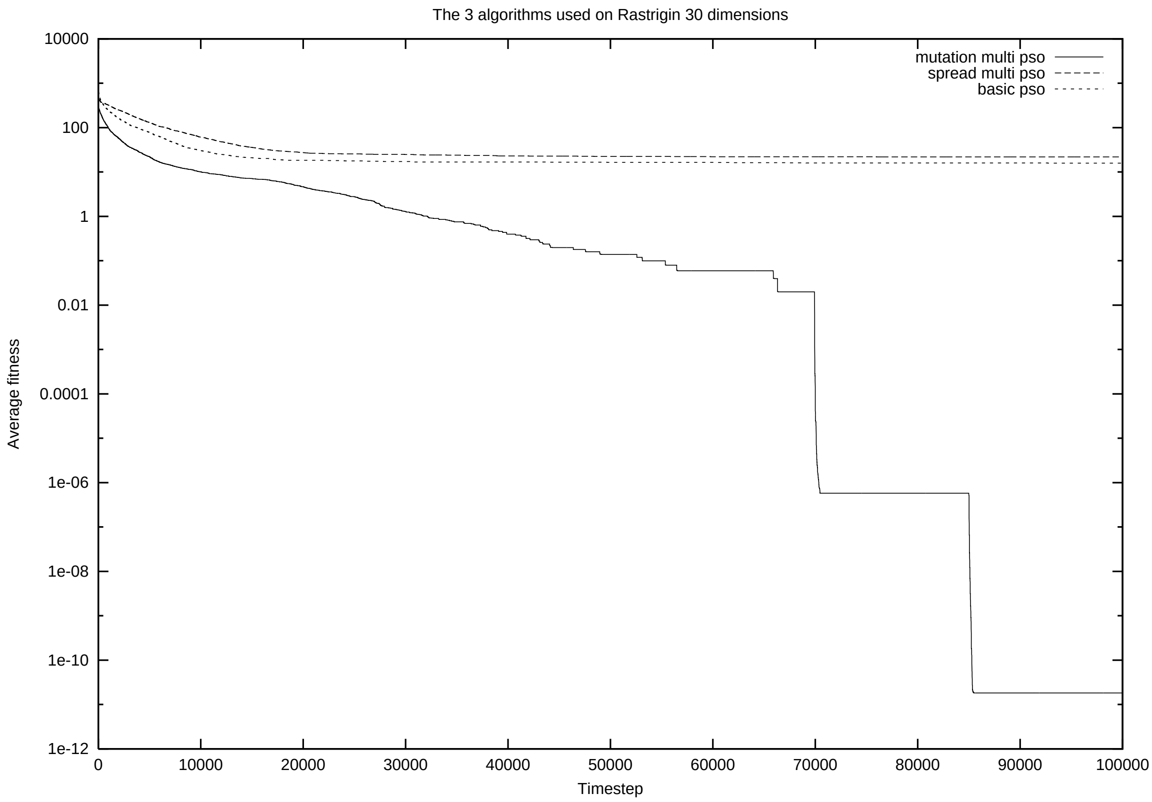

Figure 7: Graph comparison of 3 algorithms on Rastrigin 30 dim

Figure 4 shows the results from the three algorithms tested on the sphere 30 dimensions benchmark problem. All the algorithms has been running for 100.000 time steps. We see that the MusMUPSO converges a bit slower than the basic PSO and the MusSPREPSO is slower than all of them. However all the strategies finds the global optimum as table 1 shows.

All the PSO's seems to be converging equally fast in figure 5, which shows in the three algorithms above used on Rosenbrock in 30 dimensions. Again the MusMUPSO gets the best result and the basic PSO performs a little better than MusSPREPSO.

On figure 6 you see the three algorithms tested on Griewank in 30 dimensions. Notice that both MusSPREPSO and the basic PSO is stagnated before 10.000 time steps and that the MusMUPSO are still improving after more than 90.000 time steps - and is still improving. The basic PSO converges fastest but the MusMUPSO finds the best result and avoids stagnation. The exact result of the runs can be found in table 1.

At the Rastrigin 30 dimensional problem, the MusMUPSO shows its strength and outperforms the basic PSO and MusSPREPSO. This can be seen in figure 7. MusSPREPSO stagnates around 50.000 time steps and basic PSO after approximately 60.000 time steps. But the MusMUPSO takes some large steps and hits 0 if you let it run long enough. Actually many runs hits the minimum of zero within 100.000 time steps but that is hidden by the fact that figure 7 shows an average of 50 runs.

5 Tools

As a helpful tool in the development- and testing phases we had a viewer to visualise the behaviour of the swarms on two dimensional problems. This gave us the possibility to fine tune the behaviour to how we think it should be and notice if this behaviour was optimal in general or on specific problems. This for example gave us an idea about why a simple mutation (movement) of a swarm along one coordinate axes was better than mutation where the movement was not locked to coordinate axes. It seems that the mutation results in higher diversity meaning that pariticles is not clustering as much. The experience in [4] and [6] is that non-clustering leeds to better results. It also made it easy to judge if 'bad' swarms were moved away to new locations in search space often enough or perhaps to often and so on. Figure 8 shows a snapshot from the viewer of two swarms in the search space at initialisation. The snapshot in figure 9 shows that the swarms have already almost separated in to different swarms after only 8 generations. And the snapshot in figure 10 shows the fast convergence towards optimum after only 40 generations.

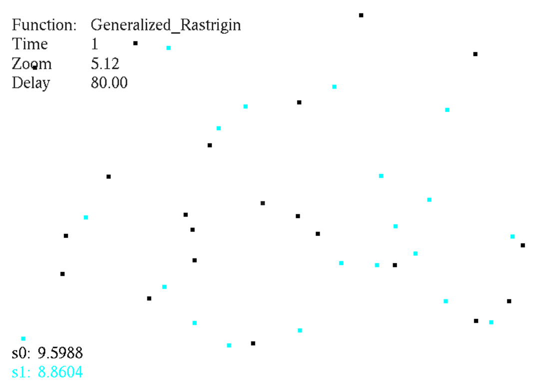

Figure 8: Snapshot of a run of the `visualise2D`-program at initialisation on 2-dim Rastrigin.

6 Discussion

Our results supports our assumption that multi swarm PSO's performs better than basic PSO's. However the results are not as good as expected in all cases. The improvement compared to the basic PSO on the problems shows a small one on Rosenbrock, Griewank and a big one on Rastrigin. On sphere MusMUPSO finds the global optimum slightly slower than the basic PSO.

MusSPREPSO, the spreading PSO, how well does it work? We expected that it would perform a little better than the basic PSO,

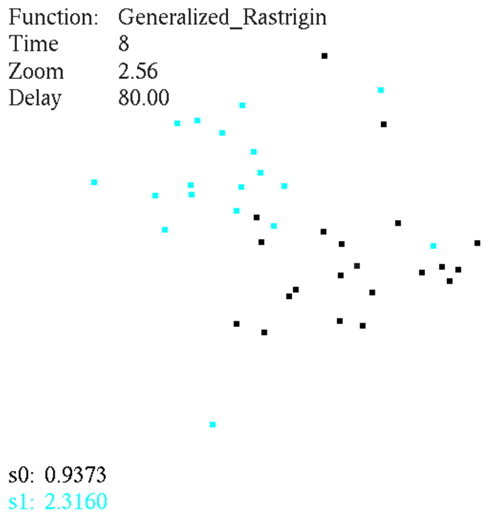

Figure 9: Snapshot of a run of the `visualise2D`-program at generation 8 on 2-dim Rastrigin.

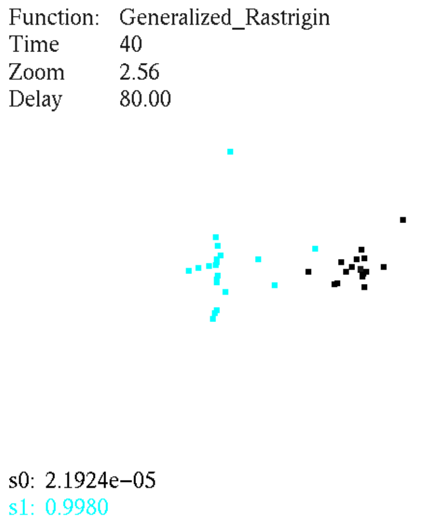

Figure 10: Snapshot of a run of the `visualise2D`-program at generation 40 on 2-dim Rastrigin.

but was aware that the random spreading was not a very good solution, because every time a swarm is spread it has to start searching from scratch again. This lead to MusMUPSO . . .

MusMUPSO: The idea behind MusMUPSO was that instead of starting all over by spreading the "bad" swarms over the whole search space, this PSO works from the best area found so far. It does this by making small movements away from this area along coordinate axes. It is a very simple but efficient kind of mutation. We have tested an other kind of mutation where movement in all directions are allowed. But its performance was disappointingly poor.

How many swarms are optimal for a multi swarm PSO? The tests indicates that 2 is optimal. And that seems reasonable as the more swarms the more work has to be done per generation slowing down the algorithm. Furthermore two swarms should be sufficient for avoiding getting stuck in local optima. We believe that by adjusting MusMUPSO's mutation distance among other parameters like initia weight w it will avoid stagnation.

Stagnation factor is important. If a swarm stagnates to quick, because the stagnation factor is low, then there is a risk that the swarm will be moved away even if it was fine tuning in a promising optimum. On the other hand, if the swarm is allowed to fine tune on local optima too long, because the stagnation factor is high, then valuable time can be wasted and the PSO slowed down.

The 'too-close-criteria' is very problem dependent. The distance should, as mentioned before, not be to large because then there is a risk that the best swarm stagnated at local optimum keeps the other swarms away from better optima within the 'too-close-distance'. If it's too small, then there is higher risk of having several swarms fine tuning on the same optimum and thereby loosing some of the effect of more swarms.

7 Conclusion

When using multi swarm PSO there does not seem to be any advantage to be using more than two swarms. At least this was the case when dealing with the four numerical problems we used in our tests.

On background of our tests it seems reasonable to conclude that a multi swarm PSO (MusMUPSO) performs better than the basic PSO in on all problems but the 'toy problem' sphere. The tests indicate that MusMUPSO have the advantage of avoiding get stuck in local optima in most cases.

8 Future work

8.1 Migration of particles

During our more unstructured testing we have discovered more particles in a swarm will help it to converge faster and deliver better results in complex problems (Rastrigin). Therefore we believe that it would be helpful to migrate particles between swarms. But whether the particles should be migrated from fine tuning swarms to global search swarms or the other way around is an open question. This could be a interesting issue for further investigations.

During our work we discovered 20 particles is not necessary the optimal number. Therefore it would be interesting to investigate how the number of particles affects the results.

8.2 Velocity formula

Restriction of velocity by clipping coordinates of $\vec{v}$ at maximum velocity has the effect of changing the direction and not only scaling. That means that a particle at high speed is zigzagging towards it goal ie $\vec{p}$ and $\vec{n}$ which also means that it risks not to hit the target. An alternative that should be tried is to limit the velocity vector by scaling its length or using constriction factor χ as suggested in [2].

Before we saw that w is linearly decreasing with the number of generations. It seems at little too simple to let velocity depend only on the number of generations. This could also be an interesting topic to investegate.

9 Code

The source for MusMUPSO can be found at:

http://www.daimi.au.dk/~andreh/pso/

References

[1] Maurice Clerc. The swarm and the queen: Towards a deterministic and adaptive particle swarm optimization. In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao, and Ali Zalzala, editors, *Proceedings of the Congress of Evolutionary Computation*, volume 3, pages 1951–1957, Mayflower Hotel, Washington D.C., USA, 6-9 July 1999. IEEE Press.

[2] James Kennedy. Small worlds and mega-minds: Effects of neighborhood topology on particle swarm performance. In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao, and Ali Zalzala, editors, *Proceedings of the Congress of Evolutionary Computation*, volume 3, pages 1931–1938, Mayflower Hotel, Washington D.C., USA, 6-9 July 1999. IEEE Press.

[3] James Kennedy and Russell C. Eberhart. Particle swarm optimization. In *Proceedings of the 1995 IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, Perth, Australia, IEEE Service Center, Piscataway, NJ, 1995.

[4] Thiemo Krink, Jakob S. Vesterstrøm, and Jacques Riget. Particle swarm optimisation with spatial particle extension. In David B. Fogel, Xin Yao, Garry Greenwood, Hitoshi Iba, Paul Marrow, and Mark Shackleton, editors, *Proceedings of the Fourth Congress on Evolutionary Computation (CEC-2002)*, volume 2, pages 1474–1479, 2002.

[5] Morten Løvbjerg, Thomas Kiel Rasmussen, and Thiemo Krink. Hybrid particle swarm optimiser with breeding and subpopulations. In *Proceedings of the third Genetic and Evolutionary Computation Conference (GECCO-2001)*, volume 1, pages 469–476, 2001.

[6] Jacques Riget and Jakob S. Vesterstrøm. A diversity-guided particle swarm optimizer – the arpso. Technical Report 2002-02, EVALife, Dept. of Computer Science, University of Aarhus, 2002.

[7] Yuhui Shi and Russel C. Eberhart. Parameter selection in particle swarm optimization. In V. W. Porto, N. Saravanan, D. Waagen, and A. E. Eiben, editors, *Evolutionary Programming VII*, pages 591–600, Berlin, 1998. Springer. Lecture Notes in Computer Science 1447.

Using Evolutionary Algorithms for Statistical Language Generation

Tor Arne Benjaminsen

Abstract— **This paper documents experiments with applying an EA to the task of randomly generating grammatically correct sentences from statistical language data. It contains a description of the method and a summary of the results. It also discusses the problems encountered when trying to evaluate the performance of language generation algorithms.**

1 Introduction

The task of generating grammatically correct sentences is related to and borrows several elements from the field of natural language processing (NLP), but is not by far as complex. The implementation used as the basis for this paper incorporates only simple word and collocation frequencies as a basis for fitness evaluation. More complex methods, eg. methods from NLP [5, 2], computational linguistics [3] or more advanced statistical methods [1] would improve the usefulness of the fitness function, but were deemed beyond the scope of this project.

1.1 Concept and usage

The usage of random sentence generation is admittedly fairly limited. The idea for this project was to implement an automata for the IRC chat network who would create random sentences for the amusement of IRC channel inhabitants.

While this can not be said to be neither a substantial nor an important usage of evolutionary algorithms, it provided more than a fair share of challenges and proved to be an exciting project.

2 Natural Languages

Before delving into the implementation, a short discourse on natural language theory is neccessary.

The field of natural language processing is huge, and it was quickly decided that a full NLP implementation would need to be delayed for future work. In addition, most of the already available NLP products are made for the English language, and are not very robust in the face of the informal, conversational and often times extremely noisy language form that is present on IRC.

However, many ideas and concepts were borrowed from NLP literature, and some of the terminology is used in this paper. For completeness, the most salient points will be covered in this section.

2.1 Language knowledge

According to Jurafsky and Martin [2], language knowledge consists of six categories. Of these *syntax* and *semantics* are the prominent ones when working with written language, where syntax governs the ordering of words, and semantics covers their meanings.

One might lead oneself to think that syntax would be enough to determine the meaning of a sentence. However, this will not be sufficient.

Because: For syntactic rules to work for you, you need to be able to discern the word classes of different words you encounter; wether they are verbs, nouns, adjectives, pronouns, determiners and so forth. To be able to discern word classes you need not only a dictionary to look up words, you also need a way to analyze the *morphology* of words to find the base word you need to look up (eg. to find the word class of the plural word "marks" you need to refer the word "mark" in a dictionary). Often words with similar spelling can have different meaning depending on the context they are set in (eg. "well" can be interpreted as an adjective, adverb or noun, depending on context). To infer what meaning was intended, you need to analyze not only the semantics, but also what Jurafsky and Martin refer to as *pragmatics* and *discourse.*

As should be evident, finding out which elements one needs to use to process language can be a slippery slope.

3 Statistical analysis

Because the linguistic techniques were not the main issue in this project, a simpler approach using only statistical analysis was chosen. While statistical analysis does not work as well as grammatical analysis, it is simpler to implement, and should hopefully be more robust in the face of different languages and noisy language environments.

3.1 Word counts

Much can be learned about a mass of text (or, in NLP lingo, a corpus) by looking at certain frequency measures. The frequency of the words themselves give away which words are the "work horses" of a language and which are the rare "luxury items". It has been shown [5] that the frequency of words follows a power law distribution. Eg. in Mark Twain's *Tom Sawyer*, roughly half of the words in a corpus will appear only one time, while the 100 most common words account for roughly half of all the words in the novel.

Another measure which gives away a lot is the frequency of collocations, or constellations of words, eg. "we are", "that is", "are you sure" etc. In addition to saying something about the frequency of words, these measures also give information about how words are usually ordered in a sentence.

Ordering of words is perhaps the most important measure. This can be used to reconstruct common word structures and bridge the gaps between different parts of the sentence.

Although statistics in itself is not enough to fully derive the workings of a language, it can be used with some success to generate understandable sentences, although with varying quality. Sentences can be rated by using the different counts to calculate a score. Words with high firstword frequency are preferred when selecting the first word, and similarly for the last word. Bonuses can be awarded for sentences using collocations, thinking that this will encourage re-use of previously viewed structures. Statistics on the orderings of words are used to (hopefully) maintain some level of grammatical correctness.

3.2 Details

For this project one and a half years worth of backlogs from an IRC channel the author inhabits were used as the corpus. This consisted of some 650.000 words, mostly in the authors own language, but also intermixed with English, typical IRC lingo (consisting mostly of abbrevations, emoticons and various expletives) and some other languages.

What's interesting about using the output from one specific channel as the training material is that the algorithm will produce sentences customized to that channel. Not only will it produce sentences in the same language, it will also use the same terminology, and will be inclined to make sentences concerning the same topics the channel often discusses. This is of course provided that the corpus is sufficiently large to make meaningful statistics.

The following statistics were generated:

- Barewords, ie. the total number of times a word has appeared in the text
- Firstwords, ie. the number of times a word has started a sentence
- Lastwords, ie. the number of times a word has ended a sentence
- Collocations
  - Two-word collocations
  - Three-word collocations
- Word ordering

Several other measures might be relevant, eg. next-to-last/next-to-first frequencies, relation to punctuation etc., but these were left out because of the limitations of the hardware this project was performed on. The implementation was made with the Perl programming language. Perl's native hashes were used for most of the data structures, and this consumed an enormous amount of memory.

4 Model

The EA model is pretty straightforward, and the algorithm used is a standard EA without extensions, although this might be implemented at a later stage, as more experience with how the algorithm performs is gained. This section gives an overview of the implementation details.

4.1 Individuals and genome

Sentences are taken to represent individuals. The words and their ordering represent an individuals genome. This means genomes have varying lengths. Because of this some care must be taken when writing the fitness function so that long sentences are not unfairly rewarded simply by having more words contributing to the score.

4.2 Fitness

The fitness function dictates the success of the algorithm. By adding more methods from computational linguistics to the fitness function more grammatical correctness can be acheived. This implementation uses a somewhat naïve approach of adding together the relative frequencies of the different elements in the sentence. This proved moderately successful, but leaves plenty of room for improvement.

The fitness of a sentence is calculated by adding together the relative frequencies of barewords, firstwords, lastwords and collocations. By relative frequencies of a word w is taken to mean the ratio between the count of w divided by the count of the most frequent word of the same type. The elements in the sum are weighted, making some types count more towards the total fitness. Firstwords, lastwords, two-word collocations and three-word collocations were given weights of 2, 2, 4 and 8, respectively. These values were selected by intuition and some amount of hand-tuning, but will probably stand further analysis.

The length of sentences is controlled by a fixed, hardcoded ideal length. A penalty equal to the difference between the ideal and the actual length is deducted from each sentence. A possible future extension would be to calculate the average and the standard deviation of sentence lengths from the analyzed text.

4.3 Search space

The search space is huge. It is probably infeasible to define a global optimum because of the high number of parameters that figure into linguistics. This means the algorithm is trying to maximize an open-ended fitness function, over a practically infinite search space. It may sound like an impossible task, but the goal as such such is not to optimize fully. One might even argue over wether or not there is such a thing as an optimal sentence. If there was, the algorithm would be providing the same sentence or a sentence similar to it on every successful run. The goal is rather to cover a sizable portion of the search space and select the best candidate. Given enough time, the algorithm will have produced a sentence of acceptable quality. The vastness of the searchspace ensures that the produced sentences are different from run to run.

4.4 Operators

Crossover

Crossover is performed similarly to n-point crossover for binary encoded genomes [4]. Two randomly selected sentences are compared. If they have one or more words in common, crossover is performed using the positions of the matching words as the crossover points. This helps ensure that any syntactic correctness from the parents is transferred to the offspring. The crossover operation is best illustrated with an example:

- “You **haven’t** seen the best part yet.”
- “We **haven’t** got a chance.”

These two sentences might yield the following offspring:

- “You **haven’t** got a chance.”
- “We **haven’t** seen the best part yet.”

In this example we see the equivalent of a 1-point crossover. If the sentences have more than one word in common, an n-point crossover can be performed, in which alternating substrings between the matching words are transferred to the offspring. Note that since the crossover positions may be different in the two sentences the substituted strings may be of different lengths. They may even be of zero length, which is still a completely valid crossover.

Mutation

Mutation is performed by simply exchanging a random word in the sentence. The mutation always tries to pick an equivalent word when replacing, meaning it will look at the previous word (if there is one) and pick one of the words that is known to have followed it. Eg. the offspring from the crossover example above might be mutated into:

- "You haven't got a **clue**."
- "We haven't seen the best **result** yet."

The mutation rate expresses the chance of mutation being performed on a sentence. It might also be interesting to parameterize the number of words per sentence to mutate, but none such experiments were performed in this project.

5 Evaluating performance

The results from this experiment are not easily quantifiable. This stems from the fact that language in itself is highly inquantifiable in nature. Two well known sentences illustrate this:

"Colorless green ideas sleep furiously."

This sentence, accredited to linguist Noam Chomsky, gives no meaning semantically, but is nevertheless syntactically correct. Another slightly different example:

"Time flies like an arrow. Fruit flies like a banana."

This sentence, as spoken by actor Groucho Marx, illustrates how sentences may be semantically ambiguous. Indeed, most sentences are ambiguous in one way or another, but the human brain is highly adept at finding the correct meaning from the surrounding context. The brain is also good at providing context where one is not present. However, for a language processing program working only with grammatical rules, it can be very difficult to resolve wether the speaker is making comment on the aerodynamics of fruit or the dietary preferences of fruit-eating insects.

In one-off sentences without context even the human brain can have difficulties, eg. if someone out of nowhere says "I made her duck.", a listener will have to select from the following possible interpretations [2]:

- I cooked waterfowl for her.
- I cooked the waterfowl belonging to her.
- I created the duck she owns.
- I caused her to quickly lower her head or body.
- I magically turned her into undifferentiated waterfowl.

Some of these interpretations can be more or less immediately discarded by a trained speaker/-listener, but to a language processing program working only by grammatical rules, each one is as probable an interpretation as the other.

This shows the difficulty of ranking sentences by quality. This weighed heavily when deciding on a simpler statistical approach for this project. The problem with the statistical approach is that the only measure of the "goodness" of a sentence is the "degree of previousness", ie. how much of the sentence has been observed in previous text. The syntax and semantics are not analyzed, making any syntactic or semantic correctness a result of a sort of "guided randomness".

So the approach is still basically to string together words - guided, albeit still randomly - and rely on the human readers to extrapolate enough data to put the sentence in a context in which it can be understood. The algorithm is lucky to create a sentence that is understandable or perhaps even amusing every now and then.

6 Results?

So the question of sucess becomes one of wether or not the EA produces more amusing or interpretable sentences than comparable, non-EA algorithms. The raw fitness value is not a good measure of the algorithms success in this respect, as you need the human brain to evaluate wether or not the sentence makes sense. The author's impression is that the algorithm performs better than other approaches.

An idea for performance evaluationmight be to use user feedback. The IRC bot implementation lends itself perfectly to this goal, providing access to immediate user feedback to a generated sentence. The typical IRC denizen will go "hehe" or "LOL" when someone says something funny, and although somewhat inprecise, this might be used as a measurement of how appreciated the bots utterances are. Another idea might be to implement a ranking system, where the users are prompted to give a score from one to ten when the bot speaks a sentence. This might provide more valuable data, which the bot can use for verifying its assumptions about correctness.

7 Conclusions

EAs, although quantitative in nature, have proved useful for language generation although this domain is qualitative in nature. It is the view of the author that the project has been successful, even if no quantitative measures can be applied to the performance. The lack of performance measures is most likely an inherent problem when working with language.

The critical point of the algorithm, the fitness function, still leaves many options unexplored due to complexity. The Future Work section points at some of these.

8 Future work

This section lists some ideas for future work.

- EA extensions
- Explore different NLP-techniques, eg. link parsers
- Utilize grammatical rules and dictionaries for the fitness function
- Statistically identifying dynamic word classes instead of grammatical word classes
- Co-evolution of weights for the fitness function
- Semantic analysis
- User feedback
- Refactor with regards to memory usage to allow expansion

References

[1] Eugene Charniak. Statistical techniques for natural language parsing, aug 1997.

[2] James H. Martin et al Daniel Jurafsky. *Speech and Language Processing*, chapter 1. Prentice-Hall, 2000.

[3] Roland Hausser. *Foundations of Computational Linguistics*, chapter 1. Springer-Verlag, 2nd edition, 2001.

[4] Thiemo Krink and Rasmus K. Ursem. *Introduction to Evolutionary Computation*, chapter 1. (in prep.), 2003?

[5] Christopher D. Manning and Hinrich Schtze. *Foundations of Statistical Natural Language Processing*, chapter 1. MIT Press, 1999.