EVALife

Research on Complex Systems

Topics of Evolutionary Computation 2001

– Collection of student reports –

Rasmus K. Ursem (Ed.)

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

Evolution of controllers by means of symbiosis

Mads O. Sørensen and Jacob F. Qvortrup

Abstract— **This study experimented with evolving controllers by assigning a new perspective to EA's - inspired by symbiosis. Different concepts of symbiosis were implemented and tested. This framework, added to the ordinary algorithms concerning evolving controllers, had a significant effect on performance, mainly because of the extended interaction between the individuals.**

It is the idea that there are tendencies in nature which prevent any species from either becoming too abundant or going extinct [1]

1 Introduction

The quote above describes one of the main subjects of this paper, that is, how do we protect individuals who might have attributes worth saving, but not are sufficiently awarded by the fitness function, and how do we keep the diversity in the gene pool?

We have tried a simple but jet quite effective procedure, to present the system with a greater diversity in the gene pool, by protecting weaker individuals from extinction. This method doesn't replace current ways of evolving individuals in an EA, but should rather be considered as an extension to the standard methods. The procedure is inspired by biology's concept of symbiosis. We have chosen symbiosis because it is a powerful construct, which in nature makes collaboration and interaction between two distinct species possible. On one side this introduces better individuals into the ecosystem, but also makes life possible for organisms, which might not had survived if it wasn't for a symbiotic relationship. We will in this paper argue, that the way we model symbiosis into the system, introduces greater diversity and protects part of the gene pool, which might be worth saving. We will test this symbiotic procedure on the problem domain, which is evolution of controllers for a simple stacking problem.

2 Symbiosis

In the following sections we will give a general introduction to how symbiosis works in nature, and aspects of symbiosis we have chosen to implement in our EA.

2.1 Nature

Introduction to symbiosis

Symbiosis is an essential construct in nature, it is a way for organisms to work together and compete. Symbiosis is a close ecological relationship between the individuals of two or more different species. In nature these symbiotic relationships happen every day - everywhere - and the examples are as numerous as there are distinct.

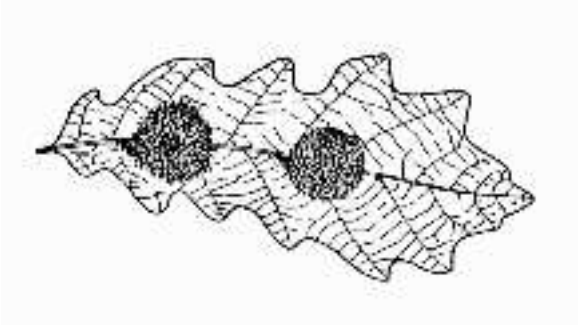
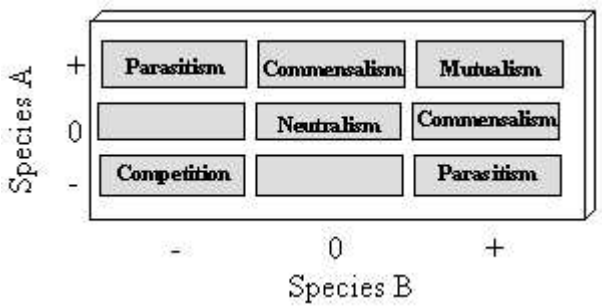


Figure 1: An example of symbiosis

Microorganisms live in the skin of many animals. These organisms live on e.g. dead skin cells supplied by the host, they further more get shelter from the environment. From this relationship the host also gets protection from harmful bacteria which could cause infections or skin diseases, since it is harder for the harmful bacteria to colonise the skin when other microorganisms live there. This relationship gives explicit advantages for the microorganisms, where the advantages for the host are somewhat more implicit. An example where both host and microorganisms get explicit benefit are galls, which are often seen as deformities on leafs, see figure 1. Plant galls are "ab-

normal" structures that develop in the cells, tissues, or organs of a plant only when it is colonised by certain organisms such as bacteria, fungi, nematodes, mites or insects. These organisms secrete enzymes, which simulates hypertophy (overgrowth) in the host plant. For the microorganisms the advantages are the same as the previous example: Protection and food (nitrate).
Not all relationships between different organisms are mutually beneficial. The leech for example lives on the host's blood, and can carry different kinds of diseases and blood infections. This relationship is only beneficial for the leech and might indeed be harmful to the host. In figure 2 the different concepts of symbiosis can be found.[1]



Mutualism – both species benefit

Commensalism – one species benefits, the other is unaffected

Parasitism – one species benefits, the other is harmed

Competition – neither species benefits

Neutralism – both species are unaffected

Figure 2: The concepts of symbiosis

Why use symbiosis in an EA?

The main problem in many evolutionary algorithms is to keep a certain amount of diversity in the system. The diversity of cause prevents the system from stagnating too soon. A specific problem in developing controllers is that good partial solutions to a problem may not be awarded in the fitness evaluation, which might cause the good partial solutions to disappear along with its gene mass from the system. The keywords here is keeping diversity, and saving weaker individuals, which might have a good solution of some partial problem. It is here that symbiosis comes in. By joining two individuals, we try to see if the combined effort the two individuals evaluates better than the two individuals appart. The new combined individual holds the entire gene mass of the two individuals, which joined in symbiosis. By doing this we investigate how different individuals might solve problems together, and as a side effect the weaker individuals has a better chance of surviving. This new relationship might either be based on Mutualism, Commensalism etc.

(F1 < Fnew) and (FNew < F2)	→	Parasitism
F1 = Fnew = F2	→	Neutralism
(F1 < Fnew) and (F2 < Fnew)	→	Mutualism
(Fnew < F1) and (Fnew < F2)	→	Competition
(F1 < Fnew) and (F2 = Fnew)	→	Commensalism

Figure 3: Relationship between model and nature.

The relationship of cause depends on the fitness evaluation after the symbiosis. If Individual 1 has the fitness F1 and Individual 2 has the fitness F2 and the joined individual Inew, which consist of a symbiosis between I1 and I2, has the fitness FNew then the symbiotic relationship between I1 and I2 would be as in figure 3.

All these relationships, except competition, tries to

- Protect parts of the gene mass from extinction, if the two individuals performs beneficially with one other individual (if they don't compete or if there are no severely parasitic draining).
- Find new solutions by combining the individuals, which might bring greater diversity into the system.

The model we present in the following section is not based on any definition of species. All the individuals can join together in symbiosis, even two with the exact same genes. The idea of defining rules for which symbiosis should be

[1] A ordinary way of dividing symbiosis. Se description in Boucher [2] or http://www.cals.ncsu.edu/course/ent591k/symbiosis.html

possible seems reasonable, but is not included in the following model.

2.2 Model

As described in the previous section, symbiosis is a complex mechanism. We wanted to take this mechanism and make it part of our model. We defined symbiosis in our model to be somewhat different than that in nature. When two individuals join together in symbiosis, they become part-individuals of a new third individual, which has the ability to use the two parts as functions, see Figure 4. In order to maintain a fixed population size, the part-individual with the highest fitness is copied and lives on in the population.
Only two individuals can join at a time, this

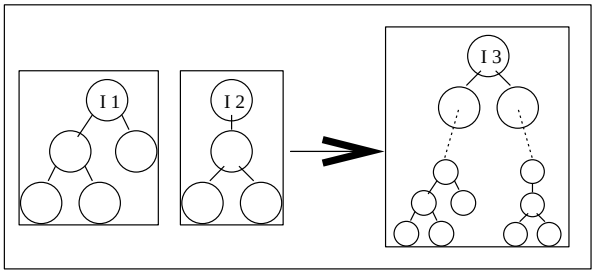


Figure 4: Two individuals enter symbiosis.

could look like a limitation in contrast to the endless posibilities in nature. However the individual created by symbiosis enjoy equal opportunity as other individuals in the symbiosis selection process. This makes it possible to join multiple individuals.

3 Experiments

3.1 The Block Stacking Problem

This problem, described in detail by Koza 1992 [3], was used to test the difference between GP with and without symbiosis. The goal is to find programs that can take any starting configuration of the blocks, and arrange them in the correct order. In Mitchell 1996 [4] and in our experiment, the correct order of the blocks was the word "universal".
A starting configuration is simply a description of the content of the stack and the table. Problems of this kind have been used extensively to develop and test planning methods in artificial intelligence.

The building blocks available to make the programs were a set of terminals and nonterminals. Nilsson [5] defined this set of building blocks in 1989. The terminals are called sensors, and the nonterminals are divided into two categories, control structures and actions.
Each sensor returns some information that can then be used by nonterminals to either change the starting configuration, or as expressions in the control structures.
The building blocks are:

- Sensors
  - CS - returns the value of the top block on the stack, if the stack is empty, NIL is returned.
  - TB - returns the value of the topmost block such that all blocks below it is positioned correct on the stack. If this block doesn't exist (which means the stack doesn't contain any correctly placed blocks), NIL is returned.
  - NN - returns the value of the block that is needed on top of the topmost correct block. If the goal has been reached, NIL is returned.
- Actions
  - MS (x) - move x to the stack if x is on the table and return x.
  - MT (x) - move x to the table if x is the top block on the stack and return x.
- Control Structures
  - DU (e1, e2) - means do e1 until e2 is true (all values except NIL is considered true)
  - NOT (e1) - if e1 is NIL return true, if e1 is the value of a block return NIL.
  - EQ (e1, e2) - returns true if the value of e1 equals the value of e2.

In order to use symbiosis we added a category

- Symbiosis

– CALL - if in symbiosis, call a individual as a function and return the value.

From these simple building blocks a random population was created, the fitness of an individual was the number of starting configurations the individual could solve.

3.2 Results

In our experiment we used 1-point crossover of the syntax trees, and the following mutations

- `mutateCollapse`, finds a random nonterminal and collapses it to a termnial
- `mutateNewNonTerminal`, finds a random nonterminal and replaces it with a new nonterminal and subtree.
- `mutateExpand`, finds a random terminal and expands it to a nonterminal with a subtree.
- `mutateNewTerminal`, finds a random terminal and replaces it with another terminal

The population size was 100, and we had 14 starting configurations.
This was then run for 100 generations, both with and without symbiosis.
As can be seen on page 6 in Figure 6 the average fitness raised with more than 100%, and the same for the best fitness, see figure 5 on page 5.

4 Conclusions

The concept of symbiosis, taken from biology, showed quite effectfull in our problem domain. Compared to the standard implementation it performed quite well, it found the optimal solution of the stacking problem after about 100 generations, while the standard EA used an average of 220 generations. This can be explained by the symbiotic algorithms ability to avoiding stagnation. These tests must be seen in the light that the symbiosis algorithm isn't very performance expensive.

There are some problems worth mentioning. The size of the programs generated grows very fast; as a consequence of this problem we have limited the number of commands of a single program to 200. Programs exceeding this limit are penalized by the fitness function. Another subject of concern is of course the search for a solution to the problem that is not just correct but also efficient, this hasn't been our main subject of concern in this paper.
By using a kind multiobjective optimization this problem could properly have been included in the algorithm.

In this paper we focused on a simpel problem domain to test the symbiotic algorithm approach. There is still work to be done in expanding and testing other more complex domains.

References

[1] D. H. Boucher. *The Biology of Mutualism: Ecology and Evolution*, chapter 1. Croom Helm Ltd., 1985.

[2] D. H. Boucher. *The Biology of Mutualism: Ecology and Evolution*, chapter 2. Croom Helm Ltd., 1985.

[3] J. R. Koza. *Genetic Programming: On the programming of computers by means of natural selection.* MIT Press, 1992.

[4] M. Mitchell. *An Introduction to Genetic Algorithms*, chapter 2. MIT Press, 1996.

[5] N. J. Nilsson. *Action Networks.* Proceedings from the Rochester Planning Workshop: From Formal Systems to Practical Systems, Tecnical Report 284, Computer Science Department, University of Rochester (N.Y.), 1989.

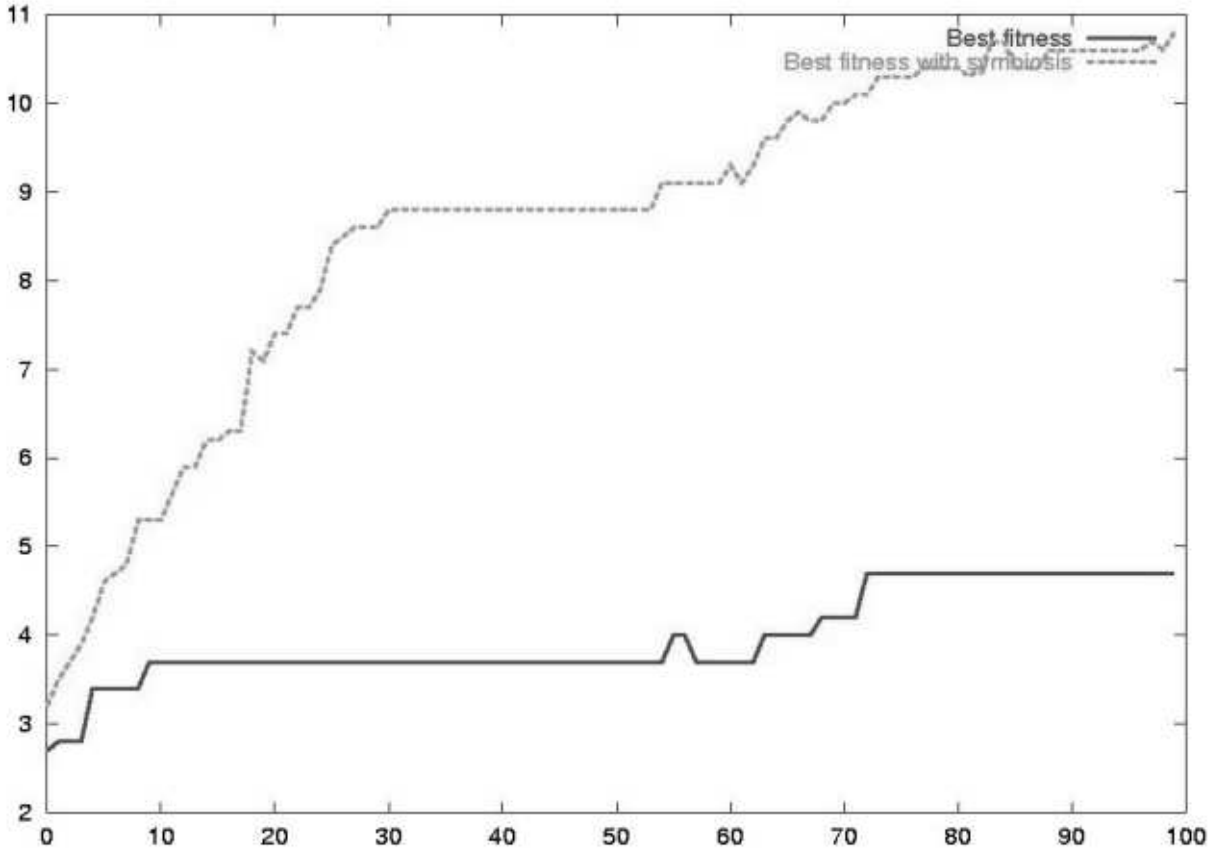


Figure 5: Comparison between the best fitness with and without symbiosis.

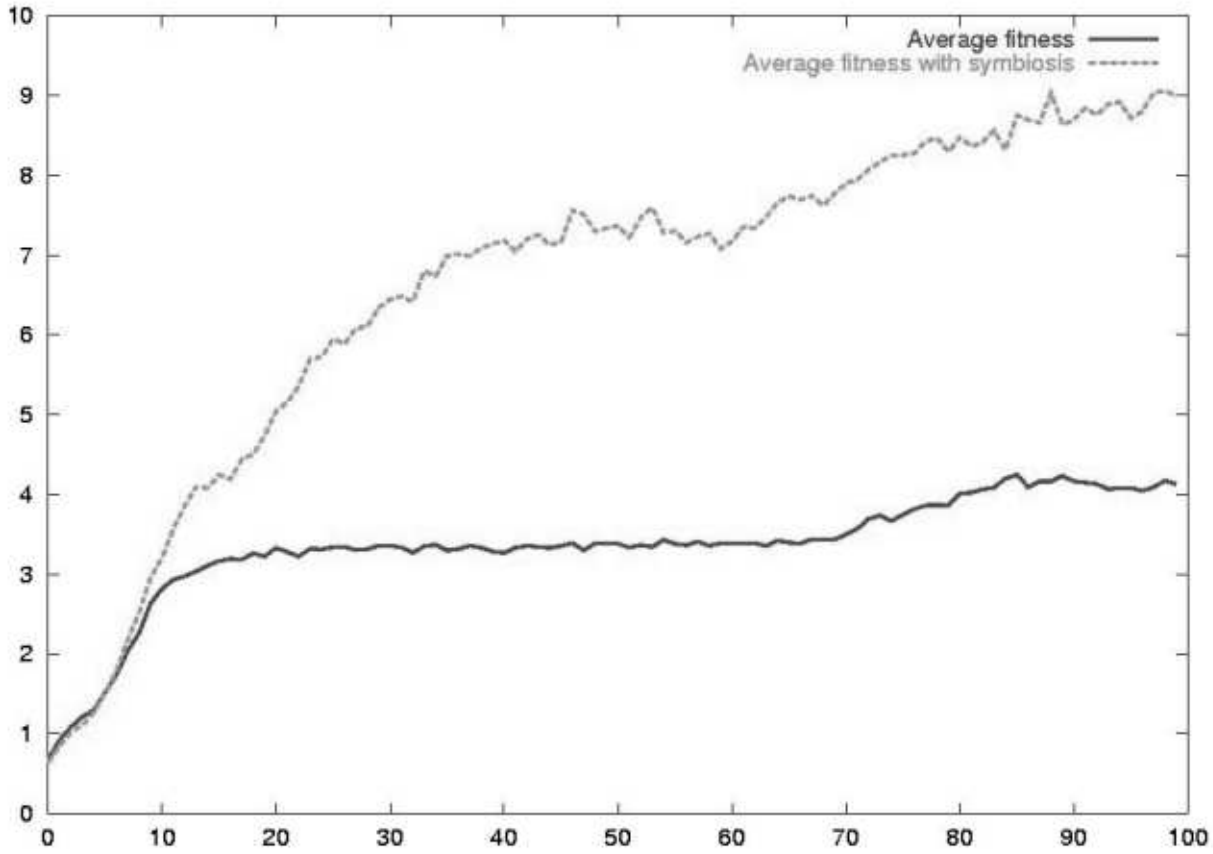


Figure 6: Comparison between the average fitness with and without symbiosis.

Feature extraction in chess

Henning Korsholm Rohde and Wouter Boomsma

Abstract— **This paper describes an evolutionary approach to the discovery of chess *features*, which in essence constitute the knowledge base of a chess engine. As features set a bound on the reachable level of play, our approach is in sharp contrast to the usual fixed hand-crafted features. Numerous problems arise and are addressed - most prominent being slow convergence and evaluation uncertainties. Although initially equipped with the material advantage feature, no useful features have yet emerged.**

1 Introduction

The game of chess has traditionally been regarded as a purely intellectual discipline requiring a considerable amount of intuition to master. This was generally undisputed until the human world champion, Garry Kasparov, was defeated by IBM's Deep Blue in may 1997. In a large extent Deep Blue's strength relied on pure computational power, but more importantly quite significant effort had gone into hand-tuning the machine's positional understanding prior to the match. This proved to be the difference between success and failure.

Although in the strict game-theoretical sense any legal position is either won, lost or drawn, practical play does not benefit from that fact. In evaluating a chess position both man and machine typically extract certain abstract *features*, such as material advantage, mobility, king safety etc, and weigh them up against each other. These *weighted features* do not follow from the rules of the game, but are *rules of thumb* emerged from experience or supplied by an external (human) expert. Humans in a large extent extract features by *pattern matching* to recognise *chunks* indicating, say, good pawn structures, whereas machines more often rely on measurable tactical and material features - thereby lacking the long-term positional perspective.

As these building blocks bound the level of play - in both directions, e.g. a pure material advantage evaluator will always accept a sacrifice and will never set a piece *en prise* - finding the right set of features is in essence the key to success. The caveat is that as chess is a subtle game, high-level play is expected to require a considerable number of features and finding these and the corresponding weights is still somewhat of a dark art. The traditional approach is to use domain knowledge for the feature selection and to find reasonable weights by training using a local search scheme (e.g. gradient descent).

We propose an unbounded approach in which features are represented in full generality and subject to an evolutionary process, thereby in principle allowing *novel* features to emerge. The main problem is that - although possible - they are highly unlikely to emerge from scratch within reasonable time limits. To avoid essentially random play, the notion of the relative values of the different pieces - the so-called material advantage feature - is included in the initial population. The goal of the project is, given this minimum of domain knowledge, to extract useful features, rather than to evolve a good chess player.

2 Evolutionary approach

An important aspect of the project is to try to substantiate the feasibility of automatic feature extraction in chess. With this in mind, the implementation in constructed around a simple evolutionary algorithm.

2.1 Overview

Each individual in the population[2] contains an evaluation functions for board positions in chess. These evaluation functions consist of two abstraction levels: *subfeatures* are intended to capture geometric attributes of the board, whereas *features* combine subfeatures nonlinearly to express complex properties. The feature extraction process is enhanced with the TDLeaf(λ) learning algorithm [1], which readjusts the weighted features after each game.

Initialization: Initially, a small percentage of the population is equipped with a material advantage feature. Since genomes with this feature are expected to do extremely well compared with the random initialized genomes, the material advantage awareness was expected to rapidly spread throughout the whole population. To trace the influence of different features we attached colors to subfeatures in the following manner:

red Stand-alone material advantage

yellow Material advantage

green Random initialization

black Contribution from geometric mutation

Each genome thus has a list of colors specifying the origin of its different subfeatures. Throughout the genetic process, these colors will undergo changes according to the mutations effecting the subfeature they denote.

Selection: Selecting the individuals for succeeding populations is most naturally done by tournament selection. This allows the qualities of an evaluation function to be determined by letting our underlying chess computer use it in play. This strategy naturally only gives a crude approximation of the best individuals in the population. Completing a full size tournament in every generation would solve this problem, but it is in practice infeasible.

Alternatives to the tournament selection approach would involve a definition of fitness for each evaluation function, which in turn would require some sort of fitness function on the evolved engines. However, such a function would encapsulate a great deal of chess knowledge, and thereby violate one of the principle ideas behind the project.

2.2 Feature representation

The features and the subfeatures are jointly represented in a 2-hidden-layer neural network (Figure 1), where the number of subfeatures and features corresponding to nodes in the hidden layers can be increased and decreased by the genetic operations. This representation is in theory capable of expressing any bounded continuous function [3], which is more than sufficient in the discrete game of chess.

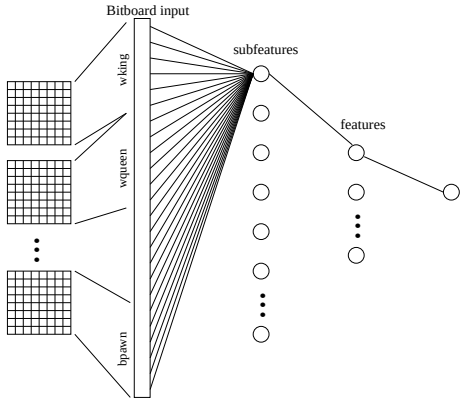


Figure 1: The evaluator

For compatability with the neural net the board must be flattened to a vector. We expand the board into a separate bitvector for every piece and color as most features in chess are concerned with specified pieces in certain situations. This allows human initialization of meaningful features. Furthermore, the discrete nature of the representation also makes it

[2]The experiment was conducted with a population size of 100. A distributed algorithm using 25 800MHz PIII-PCs completes a generation within 30 to 60 minutes.

possible to reuse large amounts of calculations during successive evaluations, making the performance in play unaffected by the large input size.

A more compact alternative of input representation uses one entry for each piece position, and the pieces are assigned values according to type and color. This has the problem that mutation and crossover operations are not necessarily stable with respect to the pieces their subfeatures target: even small mutations in the weights of the neural net may shift focus from one piece to another. This is similar to the *hamming cliff* problem with the standard binary representation.

In the feature level a differentiable nonlinear function, the sigmoid, is employed to accommodate the gradient descent based learning scheme. In the subfeature level it is not sensible to use learning due to the extensive amount of zeros in the input. This allows for the use of non-differentiable functions to further simplify hand-coded subfeature expressions.

2.3 Genetic operators

The crossover operator selects features from two genomes and combines them to form a child. Since the subfeature layers of the two selected genomes are generally not equal this often results in the copying of nearly all subfeatures nodes, thereby doubling the size of the subfeature layer. To avoid an exponential blowup in the size of genes several pruning strategies were implemented to remove nodes solely connected to the network with particularly small weights.

Due to the complex nature of the genome representation we designed mutation operations that operate on several levels of the genome, where they each have a specific goal. One is designed to decrease the size of the subfeature level and to allow the spreading of feature information throughout a single genome: it simply merges two subfeature nodes arithmetically.

Another operation is designed to try to overcome one of our greatest concerns: besides its vast size, we have little insight in the topological properties of our search space, and it may easily prove to be so sparsely populated by good solutions that the algorithm has problems finding even a small amount of them. This problem is additionally complicated by the fact that geometric properties of the board are not apparent to the evaluation function due to the flat representation of the board. We try to solve this problem by introducing mutations based on geometric patterns induced by the movement rules.

3 Results

The presented results are rather inconclusive, as we unfortunately were forced to stop the run after only 41 generations due to disk space exhaustion. Nevertheless, to evaluate the strength of the population each member battled against the initial red material advantage player - and it was not a pretty sight: 73 defeats, 25 draws and two victories; this was certainly not what was hoped for.

Upon examination the prospects that were able to draw, exhibited extremely disappointing behaviour. Many of them achieved a draw by *extraordinary* passive play - thereby creating a position without any nearby capture and exploiting the lucky fact that the red player considered a draw better than a quiet equal position. Absurdly, both players thus apparently cooperated towards ending the game by a three times repetition of a position.

The two winners naturally seemed the most promising and play similarly to the material advantage most of the time. Regretfully, the rest of the time they typically play miserably. After closer scrutiny they appear like slightly noisy material advantage players and no novel useful features seem to have appeared. It is of course possible that some of the losing genomes play similarly, but it is still seems unlikely that they should contain truly new features.

3.1 Origins of genes

As the total number of subfeatures in the population change dramatically during the course of evolution, a baseline - a run with random selection - was calculated allowing us to more directly observe the contribution of the tournament selection pressure to the gene flow. Thus, in effect coarsely measuring the usefulness of the colors - especially detecting whether novel genes (green/black) are spreading throughout the population.

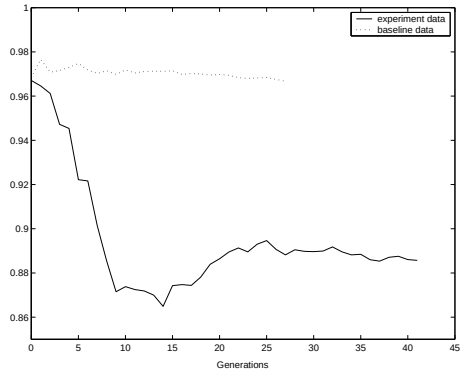


Figure 2: Color distribution - Green

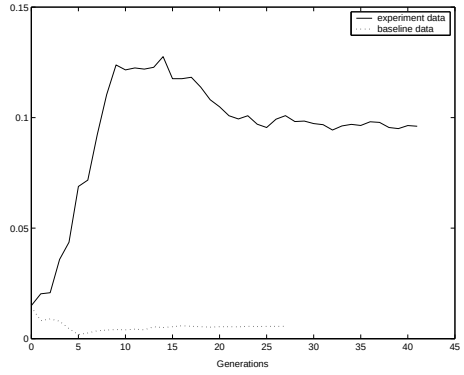


Figure 3: Color distribution - Red

The stability of the baseline distribution is remarkable (Figures 2–5). The mutation and crossover operations successfully spread gene properties through the population, but not to the advantage of any singular color. Only black has a slight increase due to the new material introduced with geometric mutations. Between each generation all individuals are trained and one may therefore expect a bias towards material advantage even without selection pressure - simply due to the decrease of weights

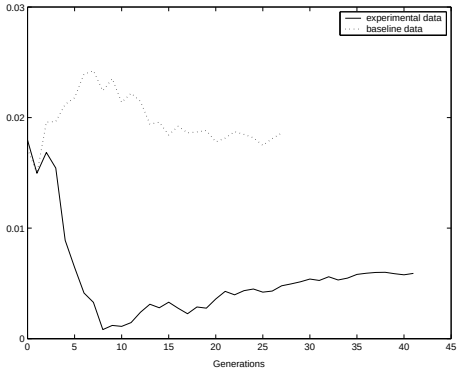


Figure 4: Color distribution - Yellow

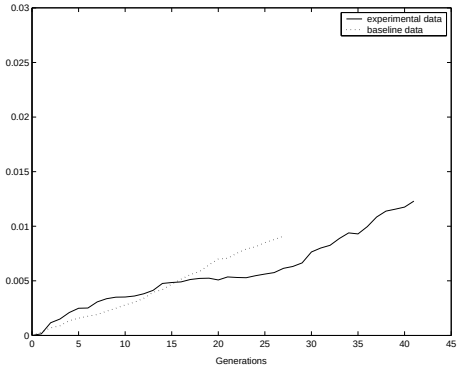


Figure 5: Color distribution - Black

on edges connected to useless nodes and the different pruning operations. The lack of expression of this effect might indicate that the training of the neural nets is not sufficient.

It is also interesting to observe the genome sizes under the reduced selection pressure. The subfeatures in the genomes literally explode in numbers - with an unfortunate impact on performance both of the evolutionary algorithm itself and on the quality of play as well. This indicates that too many subfeatures of poor quality manage to survive.

4 Discussion

As the main goal of the project is left unfulfilled, the focus naturally falls on unresolved issues. First, the sensitive subject of subfeature selection pressure is discussed in light of the presented results; second, related issues of more pragmatic nature are discussed.

4.1 Subfeature selection pressure

In order to evolve useful subfeatures a reasonable selection pressure on the subfeature level must be present; several steps were made to try to ensure this:

Real-time conditions: As the engines battle under real-time restrictions a too large neural net is prohibitive for deep search, which in effect encourages smaller and more compact engines, due to the fact that - given reasonable play - a increase in search depth implies a greatly increased playing strength [2]. This interplay is known as the *search-knowledge trade-off*. Due to certain optimizations a quite large window of genome sizes search to equal depths making the pressure somewhat coarse-grained; this is, however, slightly remedied by the shifting battle conditions caused by different work loads.

Pruning of unused subfeatures: If the output of a subfeature is weighted lower than a certain threshold (currently 0.001) by all features, it is considered to have such a small impact on the end result that it can be removed without consequences. Note that *rarely used* - but highly weighted - subfeatures are unaffected, which is desired since useful (human) subfeatures often only rarely come into play.

Inclusion of material advantage: The material advantage gene is - when taking growing genomes into consideration - as expected spreading rapidly (Figure 3), and it was also the initial hope that it would not only increase the selection pressure but also set a standard for the quality of play; as reported this was not the case.

Apparently, these steps were not enough to counter the subfeature blowup caused by the crossover operation. The material advantage gene thus completely drowned in green noise; speculatively, the material advantage selection pressure seems to have completely vanished at the stagnation after 20 generations (Figure 3).

4.2 Pragmatics

As demonstrated, in any real-world application it is the adaption to the domain and the pragmatics issues that precede results. The most peculiar, however, is a misprint in the central learning algorithm [1], which we first discovered when recalculating the results using Sutton's original article on Temporal Difference Learning [4].

Besides parameter readjustment several issues presented themselves:

Limited learning: The learning phase takes place after every game and readjusts the weights of the upper two layers to maximize the use of the subfeatures; but this does not seem good enough to even coarsely evaluate the subfeatures - thereby adding unwanted noise to the evaluation process. Unfortunately, it is expensive to, say, let engines battle more than once, but otherwise we risk relying on semi-random subfeature evolution.

Training: Closely tied with the learning issue above is the problem of newly modified genomes in which the misadjusted weight problem is even more outspoken. Due to reluctance to introduce well-selected training games, an alternative could be to use a fixed material advantage player as a sparring partner for weight adjustments. Again, this is expensive. Lastly, to train using the games played in the selection process could prove a useful compromise, which is relatively cheap, but we then run the risk of using literally obscure games.

Algorithmic modifications: To further advance the pressure from the material advantage gene - and since premature convergence is not really an issue yet - it could also prove useful to let children battle against the parents they replace. This would reject incompetents at the cost of exploration, and hopefully a stronger population would emerge.

Clearly, the greatest obstacle is limited time; this does not come as a surprise, and the difficulties consists of choosing where to allow slack and where to accept longer running times. On

that account, removing the real-time factor using (shallow) fixed depth search is not deemed good enough. Using a more sophisticated method of evolutionary computation, say spatial models, is also an option and will be considered after more experiments.

5 Conclusions

The difficult task of extracting features from the complex game of chess can not be expected to be solved after a few generations; therefore - with some modifications - we still believe in the feasibility of our initial approach. Regretfully, as always with non-vanilla applications we encountered heaps of problems of more practical nature, which invariantly have hindered the evolutionary process.

As suggested, in future work[3] the practical problems naturally need to be resolved, and much more aggressive actions to ensure subfeature selection pressure must be taken; these issues are closely tied together and when resolved should make very long evolutionary runs feasible. This is a prerequisite for useful results.

References

[1] Jonathan Baxter, Andrew Trigell, and Lex Weaver. KnightCap: A chess program that learns by combining TD(λ) with game-tree search. In *Proceedings of the International Conference for Machine Learning*, 1998.

[2] Hans Berliner, Gordon Goetsch, Murray S. Campbell, and Carl Ebeling. Measuring the performance potential of chess programs. *Artificial Intelligence*, 43, 1990.

[3] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[4] Richard S. Sutton. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3:9–44, 1988.

[3] Newest developments can be found at <http://www.daimi.au.dk/~wb/morlock/>

Applying EAs to Shape fitting

Jacob Mortensen and René Manggaard

Abstract— **Moving from visualising discrete line segments and instead visualise parametric curves in a medical application, that are close to the original line segments, have different implications.**
The result output from the implementation must be sufficiently close to the discrete line segments and at the same time provide smoothness. Furthermore, can some of the original vertices from the line segment be deleted, with only a small impact on the area deviance, this is a great advantage; the new solution with the parametric curves will be smaller. Specifically for this project, the human operator that validates the points for deviances from the raw MR pictures after the algorithm, will have to move fewer points with a parametric curve implementation, if some of them are bad placed. The result should still keep the close resemblance with the raw data.
In this paper we describe our implementation of an EA with the task to smoothen line segments and at the same time reducing the number of points in the new solution. The task is in cooperation with a ongoing research project dealing with visualising models of patients hearts.

1 Introduction

Our project is a small part of a ongoing research project, The Cardiac project, at CAVI, which is a cooperation between Aarhus University and Systematic Software Engineering.
Our goal has been to investigate and develop a method to shape figures resembling figures constructed from MR scannings using evolutionary computation. The goal from the research projects point of view is to reduce the dataset while smoothing and maintaining the shape of the scanned body part.
Humans beings with heart problems of various kinds, are at the current time scanned in a scanner type called MR scanner. The product of the scanning is around 50 pictures, that looks very similar to ordinary x-ray pictures. At the current time, this is what the surgeons use for diagnostics and planning of the operation. Needless to say, it can be very difficult to visualise a volumetric heart out of these raw MR pictures, and unfortunately, sometimes diagnoses proves to be wrong. This can be very dangerous if the error is first discovered under the operation.
The cardiac project provides the surgeons with a application, that runs on a standard pc, and can visualise a 3d model, constructed from the original raw data, obtained from the MR scannings. Pictures from a MR scanning of a heart can be seen in the appendixes.
There are many inherent problems with the raw data from the original MR pictures. The primary is the technical, since it is not possible to synchronise the pictures with the heart's subtraction phases. This means that the original data from the MR scanner is very noisy. Furthermore, finding the border between the blood and the tissue is very difficult, and calls for advanced algorithms.
The current algorithm used for finding the border produces a large set of points that lie on the border, as shown in the appendix. These points are connected with line segments to form the line segment border between the blood and the tissue.
At the current stage of the application, a human operator checks the line segments, to verify that they conform with the underlying raw MR picture. Since the set is quite large, this is a tedious job, should some of the points prove to be ill placed. For these ill placed points, the human operator usually has to move a somewhat large set of points, to make the line segments conform with the raw MR picture.
The cardiac team would like to make this easier, by displaying fewer points, that still are sufficiently close to the line segments. Sufficient here is somewhat waving since the raw data already are noisy. Furthermore, the cardiac team wished to smoothen the data, as this is closer to the original human tissue, and parametric curves have nice attributes besides the smoothness: they can be provided at any

detail level, it is only at matter of setting the granularity of the individual evaluation of a line segment.
We will in this paper describe our implementation of an Evolutionary Algorithm, which is used to provide a solution to this problem.
The next section gives a formal definition of our task, followed by a section on parametric curves. After that, follows a section on Bezier splines, the type of curves we chose to use in our current implementation.
The next sections are more directly connected to Evolutionary Algorithms. Section 5 will give a overview of the different EA mechanisms we chose for our implementation. Section 6 concerns improvements that we suggest, for getting better solutions, by improving different mechanisms in our implementation.
Finally, we have section 7 and 8 about our experiences and the conclusion on the implementation.

2 The Task

Our task is

- Given a set of ordered vertices representing the figure, find a smooth approximation, which resembles the same figure
- Reduce the number of vertices used by the parametric curves to a minimum, while keeping the above in mind.

It should be taken into consideration that the raw data is noisy, and therefore removal of points is more important than close fit to the curve.
The final decision on parameter tweaking depends on the Cardiac team, and should be done in cooperation with these. This is the human factor aspect of the task.

3 Parametric Curves

There are many different classes of curves, each with their specific characteristics. One of the main attributes of curves is the scalability in resolution; it is possible to evaluate the curve segment with higher granularity, and as such it is possible to easily implement zooming arbitrarily close to a curve, without it getting "jagged".
Another characteristic is that a parametric curve is a more compact and manipulable representation than the line segment representation.
We chose the cubic Bezier splines as a starting point, as it was easy to find material about them, and they are quite easy to understand and implement. Cubic parametric curves are more flexible and allows greater control, than do linear and quadratic curves, and that without the higher degrees unwanted wiggles and computational costly evaluation. A remark here is that our implementation of Bezier splines isn't G_0, but that it is possible to get C_1 in more restrictive cases. See [1] for a more thoroughly introduction.
Below is a short introduction to Bezier splines, that should be sufficient to read and understand the way we use parametric curves in this report.

4 B-Splines

Bezier splines (B-Splines) is specified by a number of control points, depending on the order of your spline. We will introduce the linear, quadratic and cubic B-splines which have 2, 3 and 4 control points respectively.

4.1 Linear B-Splines

This is the simplest form of B-splines, and is just a linear interpolation between the two control points.

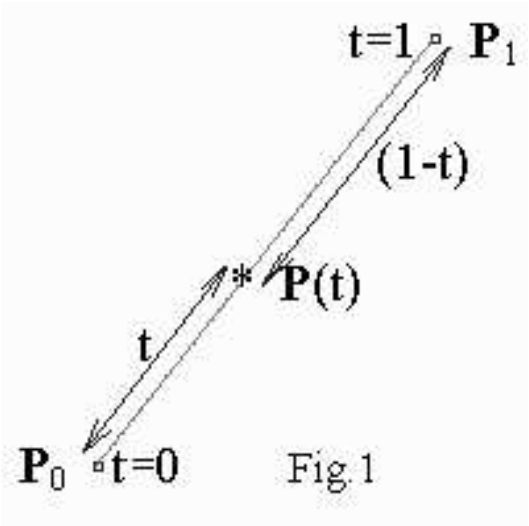


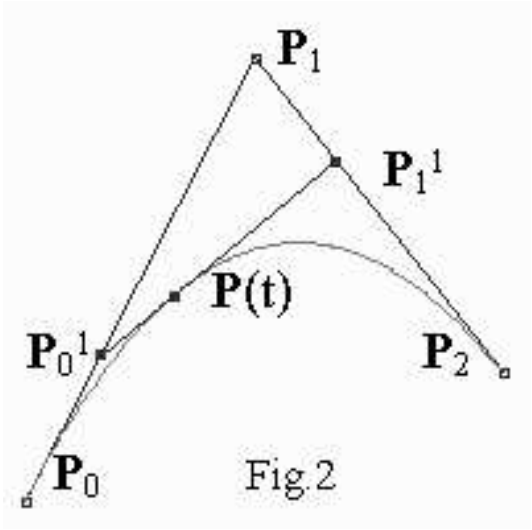
Fig. 1

$$P(t) = (1-t)P_0 + tP_1, \quad 0 \leq t \leq 1$$

This is the current stage of the cardiac teams own implementation, i.e. the line segments between the vertices on the border separating blood and tissue.

4.2 Quadratic B-Splines

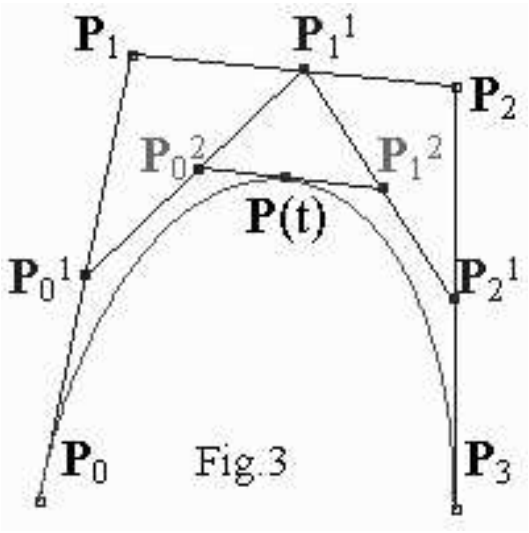
We interpolate between P_0 and P_1 and get P_0^1, and then interpolate between P_1 and P_2 and get P_1^1. Finally, we interpolate from P_0^1 to P_1^1 and get P(t), the point on the curve.



$$P(t) = (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2,$$
$$0 \leq t \leq 1$$

4.3 Cubic B-Splines

Again interpolation. We start by interpolating on the three line segments given by P_0, P_1, P_2 and P_3, this gives us $P_0^1(t)$, $P_1^1(t)$ and $P_2^1(t)$. This gives us the basics for $P_0^2(t)$ and $P_1^2(t)$, which is constructed by interpolation between the three points $P_0^1(t)$, $P_1^1(t)$ and $P_2^1(t)$. Finally our curve is given by interpolation between $P_0^2(t)$ and $P_1^2(t)$.



$$P(t) = (1-t)^3P_0 + 3t(1-t)^2P_1 +$$
$$3t^2(1-t)P_2 + t^3P_3,$$
$$0 \leq t \leq 1$$

5 Applying EAs to the task

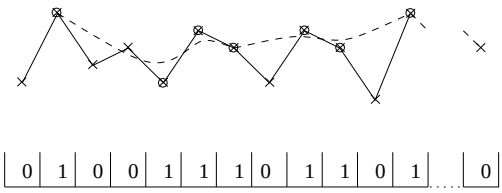
The above mentioned sections gives us some tools to shaping the figure, but how do we achieve our goal, reduction of the size of the dataset and smoothing of figure, by using an EA? We have to come up with the basics of the EA constructions; genome layout, fitness function, selection, mating and mutation. Our considerations about these topics are outlined in the following subsections.

5.1 Genome Layout

From the start it was obvious, that the genome should at least have some relation to the raw list of vertices, i.e. the line segment data. Apart from that, we had some considerations about bringing other things into the genome, one thing could be the order of the splines. After reading material about splines it was obvious, that this could become rather complex compared to the expected result, and we decided to focus only on the raw data instead.
Another consideration, which was conceived later as our knowledge about parametric curves became better, was to bring weights into the genome, so that individual control points could influence the path of the curve. This would allow a tighter fit of the curve to the line segments - another solution to allow the curves

to obtain a better fit, was to let the individuals keep their own copies of the line segments, allowing the individuals to move the vertices, and in that way try to evolve a tighter fit. This is important with our current implementation, as the Bezier splines usually don't pass through the control points. Even with a curve type that passes through the control points this mechanism could be good, since it allows greater freedom for the individuals, in the evolution race to achieve the highest fitness.

We have chosen a rather simple genome layout, since the bit-vector seems like a good solution in this case. It is then possible to expand with the weighting/copy of the control points in a later stage, so this is not a part of our genome at the current stage of the implementation. Before this comes into consideration, we might try to implement another curve type into our EA, a type that goes through the control points, as this is a rather simple step.
We used a bit-vector of length equal to the number of vertices from the raw data. Our interpretation of the bit-vector is as follows: If bit i is $TRUE$ then we use the data point at that index in the raw data array in this individual (solution), otherwise the point isn't used. So the bit-vector is a pointer into which points in the raw data, this individual uses to achieve a set of curves, that fits the line segments.
From the bit-vector (and thereby the data points), we construct a number of splines given the above interpretation of the bit-vector.

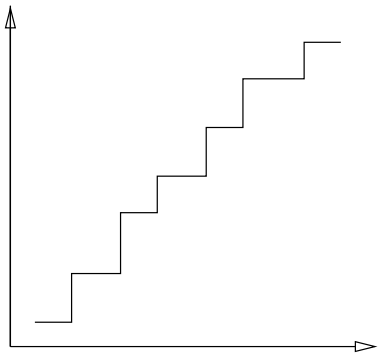


5.2 Fitness Function

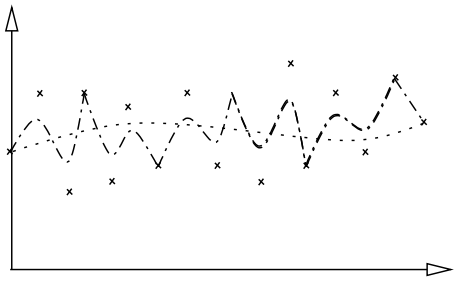
The purpose of the fitness function is to lead the solution set to the optimum, which in our case translates to:

- Maintaining the original shape of the figure
- Reducing the dataset

This is rather difficult, since the raw data is rather noisy and different raw data sets don't exhibit the same properties; some parts of the raw data is very smooth, and have a tendency to show "stair case behaviour", see the figure below.



The optimal solution here should be a curve that takes the "middle-way" and doesn't try to interpolate between the data points, as this will give a undesired "wave" effect, see drawing below.



If these stairway cases becomes just a bit larger, this could be some property of the underlying tissue, so here we would actually like to have the EA emerge with a wave like solution. Is is important to find useful weights of the fitness function, that takes this into consideration - our human intuition is hard to program into the evaluation.
Furthermore, we here have two different goals: maintaining the original shape of the curve has

a tendency to call for use of more points. Actually, as we have defined our fitness function, the best fitness would be linear interpolation between the raw data points, as the area deviance is calculated at the difference between the curve segment and the data line segment. So we have to force the EA to use at least one curve in the solution. But generally, the implementation has a better chance of getting a small area deviance, when it uses more points. Against this is the goal of reducing the number of control points, which has a tendency to give higher deviances in area. As stated before, this is an important goal of two reasons:

- The data is very noisy, so the original line segments probably isn't the best solution at all
- Fewer points makes the validation of the raw data easier, as moving a control point at a curve segment is equivalent to moving at large set of points in the equivalent part of the line segment edition.

Finding the correct weighting between this two opposing goal takes some tweaking, to get a curve solution that is sufficiently satisfiable.

Therefore our solution is to:

- Punish area deviance
- Reward solutions using fewer points
- Last but not least: A fair weight between the above

Our current implementation of the EA therefore uses the fitness function:

$$f(t) = factor\frac{PointsInSolution}{ReferencePoints} + AreaDeviance$$

5.3 Evolution Mechanisms

In this investigation we have used rather simple evolution mechanisms. The mechanisms are some of the ordinary when using bit-vectors as genomes. Short descriptions are listed below.

Selection

Our selection process are based on tournament selection as described in [2] and [3].
Two randomly chosen individual compete for survival. The winner takes the losers place in the new generation. In each evolution step, we hold as many tournaments as there are individuals in the population.

Mating

Mating in our EA is done by 1-point cross-over as described in [2] and [3].
Two randomly chosen individuals produce an offspring with a random cut-point. The offspring is then inserted randomly into the next generation, thereby discarding the previous content/individual of the slot.

Mutation

Genetic mutation is done by simple bit-flipping at a random point in the bit-vector.

6 Improving the EA

In the development of the current implementation we have come up with some improvements in search for better solutions.

6.1 Evolution Mechanisms

Our current implementation is rather simple, and not guiding the solutions in any special directions. We haven't used any special knowledge about the problem area, and therefore it should be possible to invent some better techniques.

Mating

We have thought about implementing n-point cross-over, combining genetic material from different parts of the parents. We are currently not sure that this will have a large impact, compared to 1 point-cross over, which seems to function very satisfiable.
Instead we have thought of some different mating mechanisms:

- Producing better off-spring by combining locally good solutions from the parents. To do this, we will have to somehow introduce fitness of part of individuals, perhaps by splitting a single individual up into multiple individuals, thereby effectively creating subpopulations, that could mate with equivalent subpopulations in other individuals, perhaps by some migration mechanism.
- Poly-parenting, by letting it be possible for more than two parents to supply genetic material to the new individual. We imagine this could be done by letting local fitness play a part (as above), by evaluating and combining the best local fitness' of the parents into the new child - this could also be done with the normal two parenting process, be evaluating the local areas of the parents, and then by some probability let the fittest parent win by copying its genetic material into the new child. A kind of tournament selection mechanism will then be introduced into the mating process.

Mutation

We have some different ideas that might make mutation behave better than the current implementation.

- Instead of making bit-flip randomly, we can move existing true bits in the bit-vector to the left or right of their current position, thereby exploring solutions situated in the local vicinity. We have high expectations of this, since the impressions we get from our visual feedback, lead us to believe that this will make the algorithm converge faster without loosing too much diversity. The argument here would be, that we only move about in the local area, and as such explore i this way. If we to this behaviour add the usual mutation mechanism, we still have a good chance of keeping the diversity in the population.
- Furthermore we have thought about letting local area differences influence the mutation rate in the neighbourhood. This way, in areas where there are large area deviances, we will search more thoroughly, and perhaps add points to make the deviance go down (ie in areas with large area difference from the raw data). On the other hand, if we have areas where the is a low deviance, we can try a solution without some of the points, again improving the fitness of the individual.

7 Experiences

From the project we have had different experiences, more or less frustrating.
On the positive side, our graphical feedback of how the population slowly converged towards the reference graph, was an enormous help. It made the behaviour of the population more easy to understand, than it would be just looking at raw data. Furthermore the goal of the project was oriented at giving the human operator a visual feedback of the curves fitting the reference line segments.
On the other side, working with EA's can be very frustrating, since the control is placed under the evolution mechanism, and it can be very difficult to understand why a certain behaviour emerge. When we introduced another idea, the algorithm usually came up with another behaviour, that proved to have unintended side effects. So the adapting ability, which definitely usually is the nice thing about this technique, can sometimes be rather annoying. It definitely makes problem solution a lot harder, because the program is harder to reason about, than ordinary programs. Or is it just because we are used to normal algorithmic solutions?
We have had some problems getting the algorithm to work for different reference models, i.e. different geometric forms in the raw data calls for different optimisations for the specific problem instance. The problem is fitting the algorithm to all problems. We have already described the related problems in section 5.2, about finding a satisfiable fitness function.
Also, it can be rather difficult to reason about new idea's impact. Will Poly-parenting for example be worth trying? How will it affect convergence? What about the diversity in the population? Will subpopulation in the indi-

viduals have any effect? and so on. This takes more experience than we have at the current time - we have had a lot of "huh!" experiences during the implementation process, most of them luckily good natured. We were rather surprised of the robustness of just the very simple implementation of an EA.

8 Future work

8.1 Weights or moving control points

The next implementation area, that we think would yield a far better result, is very probably the idea of moving the points/weight the control points, to make the individuals capable of fitting the reference line segment curve tighter.

8.2 Proposed improvements

Some of the ideas that we have presented in section 6, improving the EA, could be the focus of implementation into the program. There certainly should be work enough for some time. A aspect of this is developing a better design, that allows fitting new improvements quickly into the implementation. This would allow more experimentation, which is necessary, because it is difficult to foresee all pros and cons in advance of implementation of new methods.

8.3 Using other curve types

The implementation of the current curve type is not C_2, and the curves are very jagged in the points where they end and begin. Some of the other types of curves are C_2, and will of course fit better to the smooth surface a heart has. This is only a minor change in our implementation, after a bit of redesign. The reason why we didn't start out with a C_2 curve was, besides the already stated reasons, that we wanted to focus on the framework of the EA, and get it up and running before working more with splines, as this is the focus of the course.

8.4 Using another technique

It could be very interesting to implement another technique, for example Hill Climbing, to see how this would solve the problem, compared to the EA. This could also give insight, which could be used to improve the EA.

8.5 Design of good reference models

When designing the EA it was hard to direct it to good behaviour, because good behaviour is a fuzzy term. The solutions to this could be to develop some reference models, where we have some kind of intuition about the wanted behaviour. As reference model the clearest example coming to our mind is the stair case model, where our immediate feed-back could give us good tools in the tweaking of the parameters for the EA. The results could also give good input and feed-back from The Cardiac Team members in some kind of show case, since they are the final judge on the project result.

9 Conclusion

We quite pleased with the emergent behaviour of our simple implementation. The algorithm is actually capable of finding decent solutions, even though we have just used some of the most simple and fundamental techniques from EAs. The question is if it is possible to improve the solution under the current framework, without implementing some of the mechanisms we have talked about in the above, or if that would just introduce other more obscure problems. We think that one of the major strengths of EA is the simpleness of the algorithm compared to the solution capabilities they have.
As stated above, we have some problems dealing with different graph types, display different attributes from the problem domain, where some parts of the heart is smooth, and other parts are more curly. The big problem here is to find a sufficient good fitness function, that takes these different inherent problems into consideration. It is more difficult to reason about the suggested improvements will give better solutions to this problem.
We are not quite sure that EA's are a good

solution to precisely this problem, but so far it gives a sufficient good solution, that gives us confidence in a sufficiently good solution to the task, but this will take a more work.
We are surprised of the ease and simpleness of implementing an EA to our given problem. The toughest task hasn't been the implementation, but getting the right ideas; what to choose for genomes, which parameters that are important and such. It is our experience that insight into the problem area is very important, as this gives better possibilities for getting the right idea at the right time.

References

[1] James D. Foley, Andries van Damm, Steven K. Feiner and John F. Hughes.
Computer Graphics: Principles and Practice, second edition
Addison-Wesley Publishing Company 1992

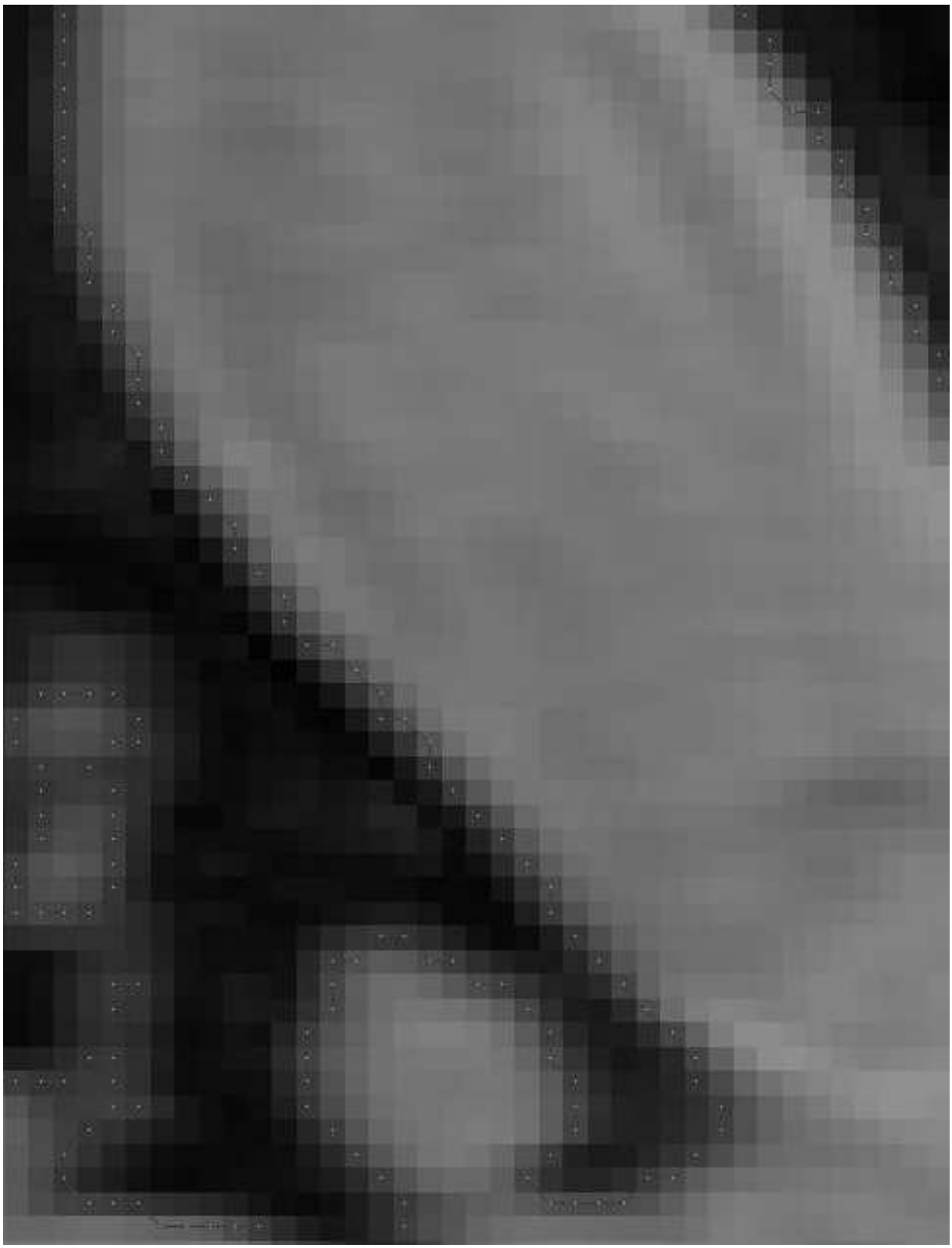
[2] Melanie Mitchell.
An Introduction to Genetic Algorithms
MIT paperback edition, 1998

[3] Note handed out in the course: Introduction to Evolutionary algorithms

10 Raw data picture from the MR scanner - 1: Overview

The grey colour is blood, the black is tissue. The dots are the vertices used in the line segment border.

11 Raw data picture from the MR scanner - 2: Closeup



3D model acquisition from 2D images

Rory Andrew Wright Middleton, Jesper Mosegaard, and René Dalsgaard Larsen

Abstract— **The utilisation of 3D models, play an increasingly important role in science and mainly in the context of understanding and experimenting with real complex objects and structures in a virtual environment. To do this it is often necessary first to obtain the 3D model resembling the actual object from a set of data, in some cases images of the object in question, and then construct the 3D model by hand. This process of hand modelling is often highly time consuming, and has motivated the idea of automising this process partly or entirely using a standard evolutionary algorithm. This report explains the idea, implementation, and testing of an EA for obtaining a 3D model, under a specific rendering, from one or more 2D images depicting a given object.**

1 Introduction

Several approaches to the problem of 3D model acquisition have been explored. Some are limited by the topology of the objects from which the 3D model is to be obtained, and others are problem specific and only work on some specific type of objects.

One method [1] operates by finding sample points e.g. corners of a building in a series of images depicting the same object but from slightly different angles. Then by matching up sample points from the different images corresponding to the same region of the object the actual shape of the object can be calculated, if camera lense and position is known in advance. The sample points are obtained by analysing the color changes in the picture and by applying heuristics to interpret these changes in terms of corners.

The results obtained with this method are good, but as the method is dependent on the obtained sample points the method does not work on smooth surfaces such as faces where color changes are infrequent, and also topological complex objects might get this method into trouble.

The method described in [2] is somewhat more problem specific. The goal of this approach is to obtain 3D face models by analysing 2 pictures of a human head, one from the front and one from the side. This method also operates with the notion of sample points, and these are obtained by analysing the two pictures and by recognizing edges and face features such as mouth, eyes nose. Then, by matching up these points with corresponding points in a "standard" 3D face model, this standard model can be modified to resemble the head of person from the pictures.

An other and widely used method regarding acquisition of 3D face models, is the use of a projected light grid [3]. By projecting a grid of light onto the given object before making the 2D image to analyse, the method can obtain knowledge about the object shape. By using the projected grid in the color analysis of the 2D image it is possible to construct the corresponding 3D model. The results with this approach are good, but one disadvantage is that hardware is needed to project the light grid.

Our idea is to develop a general method of 3D model acquisition from 2D images, which will work on a series of images depicturing any object, and we want to do this without assuming anything about object topology, the camera settings, and without the use of special hardware. We have chosen to use a standard EA for the purpose, because we can't hope for good results using neither exact methods nor sample point techniques because of our very general formulation of the problem. What we could hope for is that the evolution can come up with some optimal strategy or near optimal at least, relating to each problem instance.

We maintain a population of 3D models each consisting of a number of triangles, and by evaluating the fitness of each individual we determine if the individual is good enough to survive. As we will get into, it is somewhat hard to define operators such as mutation and crossover, and the notion of fitness evaluation is not straightforward either. In the following

we will describe our considerations regarding implementation and some inherent difficulties of the problem. We start by explaining our implementation and data structures. We will go on describing two attempts of solving the problem and the difficulties we experienced regarding these approaches. Finally we discuss what could be done to overcome some of the difficulties and what could be done to improve the performance of the EA.

2 EA and chromosome design

Basically a chromosome consists of two lists: One of points in 3D space and one of triangles. Each triangle has a reference to three points and to its neighbours. The triangles are all connected in such a way that they form a 3D surface. Each triangle also has a anticlockwise ordering of its points, which we use among other things for finding the normal of the triangle (used for shading).

2.1 Recombination

Recombination deals with how we get from one generation to the next. Of course, first we compute the fitness based on the technique given below. The chromosomes in the next generation are a combination of the chromosomes found using tournament and new chromosomes created by crossover. We have a variable in our program that tells us how many percent of the next generation we want to be created by crossover. We find the best individuals of the old generation by tournaments with 2 individuals. We also use elitism by always keeping the best two chromosomes of the old generation. When we do crossover, we have a tournament between four individuals and the best two get to perform a crossover. When we've created the new generation we run through all of the chromosomes and use the various mutation operations described below on the chromosomes.

2.2 Fitness evaluation

In order to evaluate the fitness of a chromosome, we render its triangles onto the screen, and load all of the image data into a buffer. Now we have an array corresponding to the original image, and an array corresponding to the rendered image. We compare these two in order to get a fitness. Instead of comparing the images pixel by pixel, we take lots of random sample points and use these to calculate the fitness. The fitness evaluation is divided into two parts, first we look at the whole picture and see how close the resemblance is, and this number is normalized to a number between 0 and 50 (50 being good). Then we run through all of the triangles and see how well they resemble the original image. This also gives each triangle a fitness. We take the average triangle fitness, which is a number between 0 and 50, and add it to the chromosomes overall fitness, giving us a number between 0 and 100, where 100 means that every sample point has exactly the same RGB value as the original. Of course, if we had multiple source images we could render the chromosome from all the different angles and get a more realistic fitness. As things are now, a chromosome can easily get a really high fitness just by being a big gray blob, and it's easy to imagine that we can get a chromosome with fitness 99 which may resemble the given picture from a certain angle, but that the chromosome has achieved this resemblance by doing all sorts of weird tricks thereby turning itself into a very corrupt 3d-model. Of course, this would also be an obvious place to do some sort of multi-objective optimization. For instance, our objectives are also to create an object with few polygons, so we could penalize a chromosome if it consisted of too many triangles.

2.3 Mutation

We have a variety of mutation functions that are all applied to triangles. When we mutate a chromosome, we run through all of the triangles and use these mutation functions with some given probabilities. These various operations have been chosen in such a manner that it is possible to create a wide range of 3D-shapes by applying them. We have functions for adding and removing triangles, for adding and removing complexity, and functions for moving points around.

Add Triangle

This mutation function can only be applied to a triangle if one or more of its neighbours are non-existent, which means that it's possible to grow an extra triangle out of one of the triangle's sides. The new triangle will have exactly the same shape as the original triangle, since it will be a mirror image.

Delete Triangle

"delete triangle" can only be applied to a triangle if one or more of its neighbours are non-existent, otherwise we'll end up with a shape with a hole in the middle, which we won't be able to fill out again.

Split Half

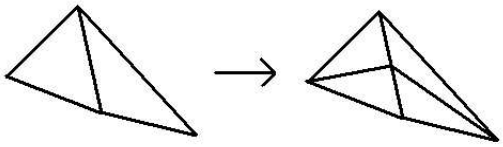


Figure 1: The split half operation

The "split half" operator basically splits an edge in half, thereby changing one triangle into two smaller ones (if the edge is only incident to one triangle), or two triangles into four smaller ones.

Contract

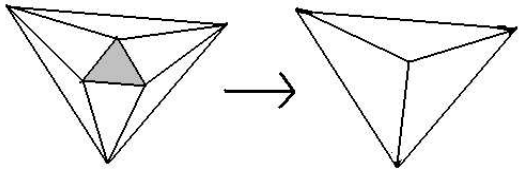


Figure 2: The contract function performed on the grey triangle.

Since "split half" adds complexity, we figured we'd need one to remove complexity as well. What contract does to a triangle is eliminate it by pulling all of its three points together into the center point of the triangle. This operation has a big effect on the surrounding triangles as well, and quite often messes up the surrounding triangles by causing them to flip and overlap.

Move Point

This simple mutate operator moves a point a little bit randomly. We tried to take some precautions with this operator to make sure that the created triangles didn't become too messed up. First of all, we didn't want triangles to become to long and thin, and we didn't want triangles to flip either (to "flip" is when point A in a triangle crosses the line defined by B and C). This can be prevented by looking at how far away a point A is from the line defined by B and C. We don't want it to be too close, but we don't want it to be too far away either. So if a "move point" operation would result in an undesirable triangle defined in this way, we wouldn't perform this operation. Unfortunately this constraint resulted in some points that became "locked" meaning that they never got moved. Another thing we tried to do in this operation was to see that if a triangle had any undesirable properties before we moved it, we tried to move the points in such a way that the triangle became healthy again. Unfortunately this wasn't much help either, since when we tried to "cure" a triangle, we usually ended up messing up its neighbours.

2.4 Crossover

It is pretty hard to define a crossover function that does something that makes sense without ending up with a corrupt chromosome. The crossover function we've come up with works as follows: It takes two chromosomes c_1 and c_2, and starts of by copying chromosome c_1, thereby creating c_3. Then for each point p_i in chromosome c_3 we find the nearest point p_j in c_2 and move p_i towards p_j. After all of this, c_3 will be the child of c_1 and c_2. This function can be modified to take the fitness of triangles into account. In this variation, we only move a point if the destination point is fitter.

2.5 Problems with mutation

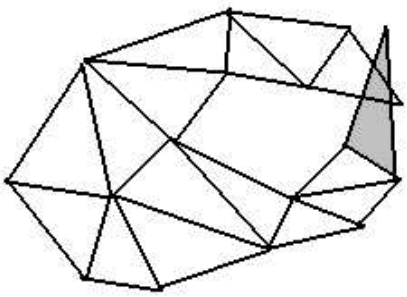


Figure 3: Example of "illegal" add triangle operation in the 2D case.

The main problem with these mutation and crossover is that we have a reasonable idea about how we want a 3D shape to look like, but it's pretty hard to tell the algorithm about this. For instance, the attempts we made to get the move operator to make nice triangles ended up giving us some new problems. Simple operators like add Triangle can also very easily end up making overlapping triangles, because each triangle only knows its immediate neighbours and it's much too timeconsuming to check the entire triangle structure to check that every move is legal.

3 First approach: Finding 2D shapes

To break up the implementation into smaller steps we decided first to implement an EA recognizing 2D shapes.

3.1 Outline

In this approach we initialised the population with individuals containing only one triangle positioned in the middle of the frame. The idea was to let this triangle evolve into the 2D shape using lots of mutation, add triangle and crossover, touching only the x and y coordinates of the points. Input is one black and white image depicting a solid white 2D shape.

3.2 Problems and their solutions

First of all we do not want to "ruin" individual with mutation and crossover. By "ruin" we mean we do not want triangles to flip. Furthermore we don't want small and thin triangles, because these are more likely to get flipped in a mutation operation. Flipped triangles are not so much of a problem when trying to approximate 2D shapes, but one of the reasons that they are unwanted in this context is that when a triangle flips, we most often get three triangles on top of each other. This is however not beneficial when we as a secondary goal want to minimize the number of triangles used to construct the model, and as we come back to later this problem is even bigger in 3D.

To avoid these flips and thin triangles we check the shape of the triangle, after having calculated the mutated version of it but before applying the mutation to the triangle of the individual. We calculate a ratio for each point p defined as

```
(the length of the side subtending to p)
/(distance from p to its subtending side).
```

These ratios have to be above a minimum threshold after the mutation and only if this is the case we apply the mutation, else we leave the triangle untouched.

One other major issue is that our implementation of the data structure only allows a local view of the individual, that is a triangle and it's neighbours. But this is a problem because mutate affects not only one triangle and its immediate neighbours, but all triangles sharing a point with the mutated triangle. So by mutating one triangle we might flip another without knowing. This problem however can be solved in great extent by increasing the minimum ratio threshold mentioned above. Some experimenting was needed though to find the optimal threshold value because a value too big didn't prevent the flipping and a value too small had the effect of locking a great deal of the points. At the moment we do nothing to prevent triangles from being too small. The crossover operator described earlier gives rise to the same problems as the mutation operator, and is harder to restrict and control. So in this approach we used it with a very low probability to avoid ruining our population.

Because we start with one triangle in each individual, we want this to grow, and we allow this by use of the add triangle operator described above. With a certain probability we decide for a given triangle edge if it has no neighbours at that edge whether to add a neighbour or not.

We choose the triangle among all triangles in the individual with a certain probability, which means that the more triangles in the individual the less chance for a specific triangle to be chosen. This gives rise to a problem when dealing with non convex figures. In most cases the EA ends up in a situation where the only beneficial thing to do is to add a triangle at a certain edge of the individual. But as we've already created a huge part of the figure, (in most cases a convex part) we've also got lots of triangles, and thereby low probability of adding a triangle the right place. At this stage we did not do anything about this problem because we would rather wait until we knew how big a problem this would turn out to be in the 3D case.

As mentioned above we have a fitness for each triangle in an individual. We have tried to use this in addition to the actual probabilities to give the EA a hint of what to do with the different triangles. We did this by calculating the fitness for each triangle as follows:

```
fitness = (old fitness/2 + reward)
```

Where reward is -1 if the triangle is outside the figure according to a random chosen point and 1 if its inside the figure. Is this way the fitness will be between -2 and 2 and we used this fitness as an extra parameter for choosing to add or remove. If the fitness was close to -2 the probability of deleting the triangle was high, if the fitness was close to 2 the triangle had have been a good fit for several generations and there by we try to add a new triangle at its edge to see if this would be beneficial.

This method was not a great success because of two things. Firstly one sample point per triangle is not enough. If say a third of the triangle is outside the figure the triangle will most likely have a pretty good fitness, even though a third is way too much for the result to be nice. Also the convergence towards 2 and -2 respectively was too fast. So we decided to have 10 samplepoints per triangle to be able to calculate a better and more informative fitness. We calculate this for each generation and when used it is overwritten by the new fitness.

3.3 Results

We had pretty good results with this approach. On testcases such as squares, and discs the EA found near optimal triangulations of the shapes. On non convex shapes the result was not as good, because of the probability problem described above.

4 Second approach: Finding 3D models

In the second model we have tried to move into 3D space. This means that we now use the 2D source images to construct a 3D model. In this model each point now has an extra dimension, and therefore the pure dimensionality of the problem, and thereby the size of the search space, is bigger. The overall number of triangles needed to solve a problem is increased, as the extra dimension is more demanding on the flexibility of the model. As we shall see the initial chromosomes in the population is somewhat larger than in the 2D case.

4.1 From 2D shape recognition to 3D model recognition

The first thing we did to begin recognizing a 3D model instead of 2D shapes was to just use the 2D shape recognition algorithm with shading of the triangles. The fitness now depends on the difference between the pixels in the source image and the pixels in the rendered image. Some of the problems in 2D got even worse in 3D, and some new problems arose.

Originally we thought that the move to 3D would help the algorithm, because the extra information in shading would restrict the way the algorithm could use the triangles. We thought that the increase in dimensionality and the use of shading would give the algorithm another way of recognizing the overall shape. This would therefore help the algorithm in getting rid of the "ugly" triangle configurations,

as these configurations would not make useful shadings.

Unfortunately this was not the case. The algorithm seemed to create a population that would quickly get stuck in a local extreme because of an "illegal" configuration of triangle.

The flipping and overlapping of triangles were also a problem in 3D. Now we were even faced with the difficult task of defining what a flip and overlap was. This problem will be called "illegal" triangle configurations.

To start solving the problem in 3D we started out restricting the algorithm. This was done by not letting the algorithm itself grow a shape, but instead laying out a grid of triangles. For experimental purposes we sometimes restricted the algorithm to a mutation in the axis of the camera, in order to speed up the algorithm for certain source images. The detection and correction of flipping and overlapping has not been implemented yet.

4.2 Data-structure

The datastructures used to record the chromosomes are almost equal to the 2D datastructures. Only now we record the third dimension as well.

4.3 Recombination and mutation

By restricting the model to use a grid, we had a controlled environment in which to make experiments. Therefore we did not use any operators that could destroy the grid. This means that add-edge, remove-edge and split-half were not used in the later experiments.

At this point we did discover an interesting feature of probabilities. It turned out that the high probability of moving points connected to many triangles was a problem for the EA. This resulted in some spiky landscapes. This problem was remedied by creating a new way of selecting points so they all got the same probability. It was still a problem that our fitness evaluation didn't take the area of the triangles into account. This will also result in a skew of the probabilities of mutation.

The result of the first set-up was not optimal, the symptom being very slow convergence. It seemed that the EA was not able to remember the good configurations of triangles. We interpreted this in two ways. First of all, an area, which in our eyes had developed a good configuration, could suddenly vanish. Secondly, mutating points to make good triangles could make others worse, and the probabilistic selection of points could skew the landscape very much. Both the area of the triangles, the other triangles sharing this point and their area, would decide the outcome in respect with the fitness calculation of this triangle.

We tried to solve these problems in a heuristic way, by giving the EA some help. To help the EA recognize "good" configurations of triangles we made the recombination of two parents select which of the pair of points from the two parents to give the child, taking into account the fitness of the point. The fitness of the point is the average of the fitnesses of the triangles. Unfortunately this did not seem to solve much. This could have something to do with the fact that a "good point" cannot be easily transferred from one chromosome to another.

The problem with the destruction of good triangles is not so easily solved. A triangle having the right angle, measured with the shading, might not be in the right position along the axis of the camera in the simple set-up for the second model. The way we tried to regulate this, was to let the triangle move all points if it had a good fitness, and move only one point of it had bad fitness - With a certain definition of bad and good. Locally this will help the algorithm, but globally the triangles we are trying to preserve could easily be destroyed by other triangles sharing a point with the triangle.

5 General analysis of problems

As can be seen from the two different approaches, we have had several problems with the convergence of the algorithm.

The majority of our problems clearly stems from the huge search space and the topology of the search space.

5.1 The dimensionality of the search space

The coordinates of the points in the shape or model span the search space. A model with only one triangle already has 3 dimensions of float's. If we have n faces the number of dimensions will be between ((n+4)/2)*3 and (n+2)*3.

N	4	10	50
Min	12	21	81
Max	18	36	156

Table 1: Some examples with different amounts of triangles (N). Min and Max are the corresponing minimum and maximum dimensionalities of the problem

In the calculation of the minimum number of dimensions we have presumed that the graph is a connected planar graph, which is never the case, so this is overly optimistic. On the other hand the maximum number of dimensions is overly pessimistic because it presumes that every face is only connected to one other face. This would rarely give a good configuration of triangles when recognizing a model in 3d.

5.2 The topology of the search space

In testing our algorithm we have experienced several problems that could be traced back to the topology of the search space.

One of the most serious problems about the search space's topology is, that an arbitrary configuration of the triangles in a chromosome may have several features that do not help it in converging to a model that could resemble a source image under the OpenGL rendering.

In 2D we can clearly define a flip of triangles or intersecting triangles, which are not wanted. But in 3D this will have to be generalized to not wanting self- intersecting shapes, even though one could imagine these to be helpful in some situations. Because the restriction is now on the whole shape, the time to check and correct this situation is probably much too big. Furthermore there is no easy way to "correct" a configuration of triangles, as explained earlier in the 2D case.

Another feature of the search space has got to do with the way the triangles are interconnected. Each triangle is dependent upon its neighbours, and therefore a single triangle can contribute with a good fitness without it actually contributing to a good configuration of triangles. This also explains the way rather lousy chromosomes get a good fitness.

Without proof we also state that the search space is rather bumpy, with large local extrema which have a very flat configurations of triangles, very bumpy configurations of triangles or "illegal" configurations. The flat configurations can be escaped, but the bumpy configurations and "illegal" configurations are very hard to escape. This has alot to do with the interconnectedness of the triangles.

5.3 OpenGL rendering and fitness evaluation

Our implementation uses openGL flat and gouraud shading. This simple setup gives us several problems. First of all, standard openGL flat and gouraud shading is very primitive, the pictures rendered can be very ambiguous in respect to the 3D model. Some ambiguity might me solved by giving several source images from different angles, but this has not been implemented yet. Another problem with this ambiguity is that triangles that are not in a good configuration might still get a good fitness.

An example of this is the picture below of a pyramid seen from above. The program tried to create this pyramid shape by making a bumpy landscape for dark colors and flat landscapes for bright colors. It is extremely hard for the algorithm to move the center point all the way to the top.

5.4 So what can we do?

First of all we can restrict the dimensionality of the problem by using heuristics about the mutation and recombination of chromosomes.

Secondly we can calculate the fitness in some other way to better reflect a good individual in our eyes.

Third, we can mentally divide the chromosome into subindividuals, which are the triangles. The triangles work together in a sort

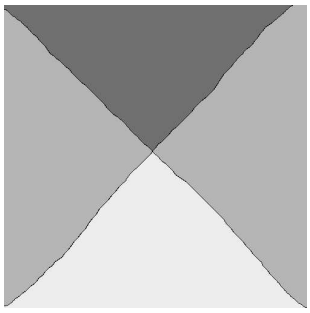


Figure 4: Pyramid input image

of symbiosis. This way of looking at the problem also clearly shows us the way each triangle only has a local view of the world. By realizing this, we can use the fitness of triangle to create heuristics (or evolve) for the chromosome fitness, recombination and mutation. This has been implemented but not used enough, because it is difficult to create good heuristics.

Fourth, more source images and a more realistic rendering might solve some of the ambiguity concerning the 3d models.

A fifth idea is to add several kinds of memory to the triangles and points. One idea is for a triangle to remember a number of optimal relations between it's three points. Another idea is to add some basic swarm intelligence, that each point remembers the direction it is moving in if it's fitness increases, and keeps moving that way.

6 Conclusion

During our long work process with this project we have come to the conclusion that this problem is probably too hard for a standard EA. For better results with this problem, one would have to try some of the alternative heuristic techniques as presented in the previous section.

7 Location of Code

The code can be found in the directory:

/users/mosegard/kurser/toep/opg2/

In this directory there are two directories, 2D and 3D, that contain the two different approaches. They are compiled using make and run with the command cgp1. They only work on linux machines. Zip versions 2D.zip and 3D.zip can be found in the opg2 directory. The input image is "test2.rgb" for both of the programs.

The tex files for this report can be found in the directory:

/users/u981334/ea/report/group_04/

The ps file main.ps can be found in the directory:

/users/u981334/ea/report/

8 Program customization

8.1 2D Program

To change mutation probabilities for the population, change lines 8-10 in the file Population.cpp. To change mutation probabilities for the chromosome change lines 420-423 in the file Chromosome.cpp.

8.2 3D Program

To change whether or not gouraud shading is wanted in the 3d version, change the booleans in lines 23-25 in the file Mafi.cpp. To change mutation probabilities for the population, change lines 19-21 in the file Population.cpp. To change mutation probabilities for the chromosome change lines 571-576 in the file Chromosome.cpp.

References

[1] Niels Husted Kjr & Kaare Begh. *Image-Based Rendering.* October 2001.

[2] Thomas Contaxakis. *3D Facial Models.* www.CS.McGill.CA/ cruel/.

[3] IMAGE, VIDEO Group for MultiMedia Communications, and Applications. *Video Cloning.* www.eurecom.fr/ garciae/3D_modeling/.

Investigation of Different Fitness Functions for the Dynamic Job-shop Problem

Jon Fogh and Peter T. Nielsen

Abstract— **This paper further investigates the idea of using penalty function in combination with fitness functions in evolutionary algorithms, to make flexible schedules for the dynamic job-shop problem. Two new penalty functions is suggested, tested and compared to all ready known, well working penalty functions. One of the new penalty functions looks promising but need further investigation and testing, while the other is showed to be no good.**

1 Introduction

Some of the hardest problems to optimise and at the same time some of the most encountered problems in the real world are dynamic problems. A dynamic problem is a problem in an environment that changes over time. When trying to optimise a static problem, that is a problem that is in an environment that does not changes over time, you are looking for the perfect optima. In a dynamic problem you will also be looking for a perfect optima but in a dynamic problem the perfect solution may be very different from the static problem, since in a dynamic problem you have to take into account, that the problem might changes the second after you have found your optima. So when you search for an optima for a dynamic problem, you have to try to predict what's going to happen in the future. You try to find the optima solution which not only is a good solution to the problem as it looks like at the present time, but also a solution which is able to handle changes in the environment in a reasonable way.

In this paper the dynamic stochastic job shop scheduling problem is investigated in connection with different fitness functions in an evolutionary algorithm. The job shop problem as we define it, is a problem describing a floor of an industrial factory with m machines and n jobs each consisting of a variable number of operations. The operations in a job, each have to run on a different machine and the operation i have to run before operation j for $i < j$. For $m > 2$ the problem is NP-hard [5].

The dynamic part of the job shop problem consists of jobs arriving at different times. The dynamic job shop problem can be seen as consisting of several static problems. Every time a new job arrives you have a new static problem that have to be scheduled. This way you end up with having a series of slightly different static problems. The problem is described in more detail in section 4 on the experimental set-up.

The dynamic job shop problems can be minimised for different performance measurements like when the last job is finished, the mean flow time of the jobs or the tardiness for the jobs. Tardiness is the difference between the duration of the job and the time it actual takes to manufacture the job. For instance if a job consist of 5 operation each with duration d_i, and in the schedule the first operation is scheduled to start at t_a and the last operation is finished at t_b. Then the tardiness is $(t_b - t_a) - \sum_{i=1}^{5} d_i$. We are in this paper looking to minimise tardiness.

Our approach is inspired by [4] where the authors experiment with the use of penalty functions to get anticipation in the dynamic job shop problem. A penalty function is a way to favourite the individuals in the population that have wanted properties. For instance the penalty function in the [4] gives a penalty to the individuals in the population representing a schedule with early idle time on the machines. This way you will after a number of generation have a population that have few individuals representing schedules with early idle time. A schedule with little early idle time has the idle time latter on in the schedule and therefore it might be easier to reschedule when a new job arrives.

We will in the following investigate differ-

ent penalty functions and the effect they have on the tardiness for a job shop problem. We are trying different penalty function and combination of these. The ideas to the penalty functions all came from our intuition on what would be the fastest way to get jobs made in a factory and on the same time be able quickly to reschedule. For instance one of the penalty functions gives higher penalty to schedules with small operations in the beginning. This is from the idea that it is easier to reschedule the jobs when a new job arrives, when there are only small operations left from the old jobs. Another idea is to try to get as many jobs as possible finished as quickly as possible, so when a new job arrives there are the smallest number of old jobs left to reschedule. The fitness functions is described in detail in section 3

Besides the penalty function the algorithm is close to a standard evolutionary algorithm with selection, crossover and mutation. The algorithm and operators is along side the representation of the schedules described in section 2. The representation used for the schedule problem is a sequence of integers also known as the permutation representation. The integer represents a job and is in the gene once for each operation in the job. Therefore to get the schedule represented by an individual in the population a schedule builder is needed. The schedule builder we are using is a non-delay schedule builder and the reason for this is also described in section 2

The structure of the paper is as follows: Section 2 describes in detail the algorithm, operators, representation and the choice of schedule builder used. Then in section 3 the various fitness functions and the ideas behind them are explained. In section 4 the results we have found and how we got them is described. Finally the paper ends with a conclusion on the achieved results and why they are as they are.

2 Algorithm

In the following the algorithm is described in detail. As all ready mentioned the algorithm is close to a standard evolutionary algorithm. The only difference is that the fitness function is more advance in the sense that it uses a penalty function. The fitness functions and the penalty functions is described in 3.

The main loop of the algorithm is therefore as known from a standard EA and can be seen in figure 1

```
population.init(popSize);
while(t<generations) {
  population.evaluate();
  population.selection();
  population.crossover(crossoverRate);
  population.mutation(mutationRate);
  population.evaluate();
  t++;
}
return population.getBestIndividual();
```

Figure 1: Main loop in the used algorithm.

As all ready described the dynamic job shop problem can be considered as a series of static job shop problem. Every time a new job arrives a new static problem is formed. The static problems are closely related and therefore the individuals found in the old problem can give you a leap start in finding the best individuals for the new problem. Therefore 50% of the old population are used in the new population. The rest of the population is, as the start population, randomly created. The population size is kept constant on 100 individuals.

The number of generation in each static problem is, as in [4], calculated on basis of the number of operations in the problem. More precise the number of generations is half the number of operations in the problem. This way a problem with many jobs, and therefore many operations, gets more time to find the best solution.

2.1 Representation

The representation is know from other problems such as the travelling salesman problem, as described in [5]. The genes in this representation is a sequence of integers where each integer, in the dynamic job shop problem, represent an operation in the job having the same number as the integer. So in a chromosome there might be five $j's$, if job number j has five operations. The first j then represent the first operation in job j and so on. The size of the

chromosome is then

$\sum_{j}^{\#jobs} \sum_{i}^{\#operation} operation_{ji}$

The disadvantage with using this representation is that you need a schedule builder which is described in section 2.3 and the advantage is that since the schedule builder is used every individual represent a feasible schedule. The representation of the schedule as permutation of the job numbers also helps in designing the operators used in the algorithm because the usual operators used in connection with permutation representation can be used.

2.2 Operators

The paper by Branke and Mattfeld [4] was the inspiration for this paper and the results found in this paper is later compared to the results found in [4]. To get a fair comparison we have chosen to implement the same operators as Branke and Mattfeld. We have also tried others operators but they are not tested.

Crossover

The crossover operation is one often used on permutation representations and is called PPX - Precedence Preservative Crossover. The PPX crossover uses a mask filled with 1 and 2 to choose between parent 1 and parent 2 to form the offspring. In figure 2 is an example of a PPX crossover with genes of size 6 representing a problem with three jobs each with 2 operations. A closer investigation of the PPX crossover operator in connection with the permutation representation can be found in [2]

```
parent 1   3 2 1 2 3 1
parent 2   2 1 3 2 1 3
mask       1 2 2 1 2 1
----------------------
Offspring  3 2 1 2 1 3
```

Figure 2: An exempel of the PPX crossover.

Mutation

Again the mutation operator used in this paper is chosen only because it is used in [4]. The operator used here is the move mutation operator and it is very simple. It picks a random gene in the chromosome and moves it into a new position. An example can be found in figure 3

```
Before mutation 3 2 2 1 3 1
After mutation  3 2 3 2 1 1
```

Figure 3: An example of the move mutation moving gene number 5.

Selection

The selection used in the algorithm is tournament selection of size 2 with an elite of one, where the individual can be in a tournament several times.

2.3 Schedule builder

There are two types of schedule builders, Non-delay and active. A non-delay schedule builder simple starts an operation as soon as possible. This is a local greedy way to avoid early idle time for one machine. The objective is to avoid early idle time on all machines and this is where the Branke-Mattfeld fitness function comes in to the picture, because it gives a penalty to individuals with early idle time.

An active schedule builder on the other hand checks to see if there is an urgent operation, which can be processed in the near future. This means that an active schedule builder can make schedules where a machine waits for an urgent operation instead of process the first possible operation. This of curse gives some idle time in the schedule.

Therefore it sounds like the Branke-Mattfeld fitness function in combination with an active schedule builder would be the perfect choice for the dynamic job shop problem. Our representation does not allow for operation to have priorities and we chose not to implement, what you could call a semi-active schedule builder, where the last operation in a job always is an urgent one.

The fact that we have chosen the permutation representation and not to implement any form for active schedule builder will not give

us the best possible result with the Branke-Mattfeld fitness function and our own fitness functions. The reason for choosing a non-delay schedule builder is that it is simpler then an active schedule builder to implement and the focus in this paper is on different fitness functions. In [4] the results have proven to get better with an active schedule builder and the same would probably count for the fitness functions in this paper.

3 Fitness Functions

Since we are decomposing the Job Shop Problem into deterministic sub-problems, which are solved successively, the problem's dynamics are ignored. We have in our implementation decided to transfer good individuals from the previously sub problem to the current sub problem, since it has been shown in [3] [6] that this approach has been proven successful. The idea in this approach is that the algorithm, tries to learn from the past and thereby transferring some information on promising areas of the search space. In this paper, we propose different penalty functions to integrate a form of anticipation into the algorithm, by giving the algorithm some simple guidelines to maintain its flexibility and suitability, needed for a better rescheduling. All the fitness functions proposed in this paper are extensions of the tedious fitness function, minimizing the tardiness of a schedule. Our fitness functions consisting of the tedious function combined with a penalty function are in the end of this section compared to the tedious fitness function and the fitness function proposed in [4]. The latter of those use a combination of tardiness t and idle-time penalty p. Normalizing both values to the interval $[0 \dots 1]$ the fitness value for an individual in the Branke-Mattfeld fitness function would be:

$$f_i = \frac{t_i - min_j\{t_j\}}{max_j\{t_j\} - min_j\{t_j\}} + \alpha \frac{p_i - min\{p_j\}}{max_j\{p_j\} - min_j\{p_j\}}$$

Figure 4:

With the parameter alpha being the weighting factor. Both fitness function described in the following, is inspired by Branke & Mattfelts solution with a penalty function, but instead of punishing early idle-time, we have decided to punish a schedule for either running small jobs in the beginning of a schedule or for choosing a job with many operations in the start of a schedule. Both choices will be described in the following section.

The first penalty function we made, was based on the idea that it seems easier to reschedule smaller than larger jobs, since a larger job clearly suspends a machine for a longer period than a smaller job. In other words, when a new job arrives, the front part of the schedule will be fixed permantly, while the backlog may be rescheduled according to future needs. It is in this step, that our intuition says, that it would be easier to reschedule smaller jobs. Therefore, we suggest to explicitly penalise small jobs, that starts early in a schedule, in addition to the original fitness, i.e. the tardiness of the schedule. Consider the two schedules pictures in figure 5. Both may have the same tardiness, but they differ in their distribution of small jobs. Schedule A has a larger amount of small jobs started early in the schedule, while B has scheduled the larger jobs in the beginning. Thus out fitness function would give B a better fitness than A. As in the case of early idle-time, the penalty is weighted with a decreasing function, but in our case we have chosen to let the function be exponential. The reason is that tests showed, that a linear function wouldn't punish an individual enough, for having a huge amount of small jobs in the start of its schedule. Thus our first fitness for a individual I, where we use a combination of tardiness t and a penalty p for small jobs with early start time is given by, the same equation as in figure 4, but with p being the penalty calculated with our new penalty function.

If the first function seems very simple, the second may seem even more trivial. Here the penalty function is trying to finish as many jobs as possible in the beginning of the schedule, by punishing an individual if it schedules an early start time to an operation from a job with many operations. The underlying idea in this approach is to finish jobs with few operations as soon as possible. By doing this we avoid the scenario, were a job which have finished all of its operations except one or two,

builds up a huge tardiness since its few remaining operations keeps getting rescheduled, without ever getting an early start time. As a consequence, in the second approach in addition to the mean tardiness fitness, we propose a penalty function favouring the early scheduling of operations from nearly completed jobs.

Consider the two schedules depicted in figure 6. Again assuming the tardiness of both schedules being equal, our approach would favour schedule B instead of schedule A, since B finishes the small jobs very early in its schedule. As in the case with the latter penalty function, the amount of penalty on an operation depends on its start time, thus the penalty is weighted with a linearly decreasing function. With this in mind, the fitness function for an individual equals that in figure 4, but with our new penalty function as p and where α again is the weighting factor.

Among the other penalty function that we considered is the two negations of the proposed functions. That is the functions favouring small jobs in the beginning of a schedule and operations from larger jobs getting scheduled earlier than operations from smaller jobs. These functions have been implemented but not tested, so no results are available yet. The same goes for combinations of all the proposed functions, including the Branke-Mattfeld fitness. Our intuition tells us, that especially our proposed fitness function could benefit from a combination with the Branke-Mattfeld fitness, since all the proposed penalty function actually isn't punishing early idle-time, hence we feel that a combination would only make the proposed function even stronger. All combinations with Branke-Mattfeld have been implemented, but at this stage aren't tested due to lack of time.

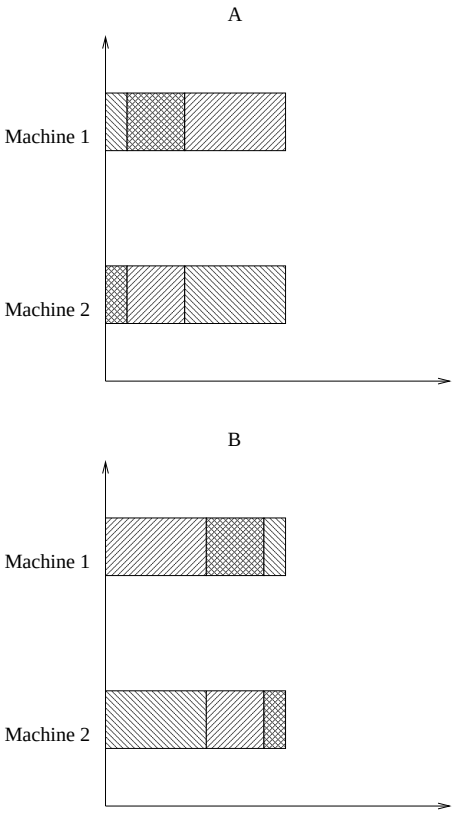


Figure 5:

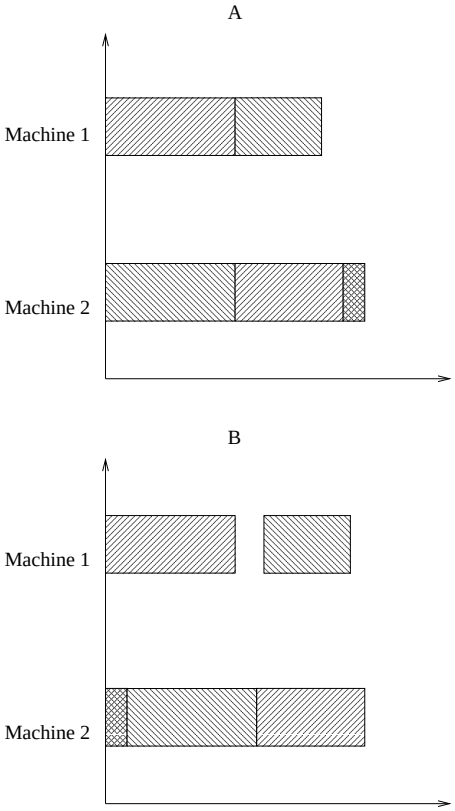


Figure 6:

4 Results

The simulation environment used in the following is widely used for simulating manufacturing systems e.g. [4] [1]. The workload in a manufacturing system depends on the number of operations in the system, which await processing, i.e. the inter-arrival times of the operations can tell whether a system has a high or low workload. Thus the mean inter-

arrival time λ can be prescribed by dividing the mean processing time P by the number of machines m and a desired utilisation rate U, i.e. $\lambda = P/mU$. The other attributes in the simulation environment is defined by the following attributes:

1. The system consists of 6 machines.
2. Each job in the system passes 4 to 6 machines resulting in an average of 5 operations in each job.
3. The machine order in an operation is generated randomly from a uniform distribution.
4. Each operations processing time are uniformly distributed in the range [1,19] resulting in a mean processing time of $P = 5 * 10 = 50$
5. The inter-arrival times are exponentially distributed based on λ by using different utilisation rates U.
6. The utilisation rate U is either 0.7 or 0.9, representing a relaxed or an excessive workload. Normally the utilisation rate 0.8 representing a moderate workload is also tested, but due to lack of time we have chosen to omit this rate.

The simulations have been run with a population size of 100 individuals, fitness tournament selection with an elite of one, the PPX crossover with probability of 0.6 and the "move" mutation operator with a mutation rate of 0.1. For each parameter setting for α and U. Ten different simulation have been performed each consisting of 500 jobs. In each simulation job 1 to 100 and 401 to 500 have been discarded in order to circumvent distortion effects [3]. Thus the following results are calculated as the mean tardiness of job 101 to 400 averaged over 10 different simulations. We are aware that 10 simulation isn't enough to prove the efficiency of a fitness function, but it may be enough to get an idea whether a function is useful or not. Hence a further investigation is required to test the actual efficiency of the function. With this in mind the goal of this investigation is to shed light on whether the two proposed function could be expected to improve the mean tardiness in the Dynamic Job Shop Problem. For each $U \in \{0.7, 0.9\}$ and $\alpha \in \{0.25, 0.50.0, 75, 1.00\}$ experiments are performed to investigate which role the parameters play's in the two functions.

α/U	0.7	0.9
0,25	18	27
0,50	15	24
0,75	15	29
1,00	12	28

Figure 7: Results for Branke fitness function.

Figure 7 lists the improvements with the Branke Mattfeld fitness in percent against EA runs with just tardiness considered as fitness ($\alpha = 0$). This table is only included to illustrate, that we may have been a bit unlucky with our tests with U = 0.9, since the improvement with the Branke Mattfeld fitness is considerable better than described in [4]. From Figure 7 we see that the results almost equal the ones obtained in [4], although the improvement with a utilisation rate of 0.9 is relatively larger in our tests, this indicates the results obtained for the simple tardiness fitness should be regarded as a bit higher than the actual value.

α/U	0.7	0.9
0,25	-33	19
0,50	-5	24
0,75	-19	8
1,00	-7	5

Figure 8: Results for large operation first fitness function.

Figure 8 lists the improvements with the "large operations first" fitness in percent against EA runs with just tardiness considered as fitness ($\alpha = 0$). From the table it seems clear that the "large jobs first" fitness don't look that well. With a workload of 0.7 it is increasing the mean tardiness with more than 30%, as described earlier the improvements with U = 0.9 is possible due to lack of simulations. The fitness function could possible be improved by combining it with the Branke- Mattfeld fitness, since in the implementation of the "big operations first" fitness, we actually are rewarding early idle-time. Another reason behind the

poor results might be, that fixing the large operations in the beginning of schedule, results in the schedule being choked in huge operations, giving the small operations a relatively huge tardiness.

α/U	0,7	0.9
0,25	15	31
0,50	5	29
0,75	-5	27
1,00	5	30

Figure 9: Results for minimise jobs fitness function.

Turning to the "small jobs first" fitness the results gets more amusing. Figure 9 lists the improvements with the "small jobs first" fitness in percent against EA runs with just tardiness considered as fitness ($\alpha = 0$). Admitted the results don't look that impressing with U = 0.7, but even though the improvements only where about 8% in average, these results hadn't the variance we observed in the Branke-Mattfeld fitness, which could be useful in different applications. With a utilisation rate of 0.9, the "small jobs first" fitness actually beats the Branke-Mattfeld fitness and even though 10 simulation isn't enough to prove this penalty function is better with a high workload than Branke-Mattfeld its gives us a hint in that direction and should be tested in the future. Like in the case with "large operations first", we dosn't punish early idle-time in "small jobs first", so another interesting idea could be to add a idle-time penalty to the "small jobs first fitness".

Summarising the results presented for the large operations first indicates that this way of penalising an individual isn't the right way to go, since almost all values of α combined with a workload of 0.7 only makes the mean tardiness larger, while the improvements with U=0.9, might disappear when running more simulation, although this needs further investigation. Different is it with the "small jobs first" fitness, almost all parameter settings shows promising results, especially the results with U = 0.9 looks very promising since we manage to beet the Branke-Mattfeld fitness in this area. Remembering that the "small jobs first" fitness doesn't punish early idle time so a combination with Branke-Mattfeld looks very promising and should be tested in the future work.

5 Conclusions

In this paper we have explored two different, but very simple ways of penalising a schedule to improve its flexibility regarding adaption to changes in the enviromenment. The first way is favouring large operations in the beginning of a schedule, since intuition tells us rescheduling small jobs seems easier than large jobs. The second idea tells the scheules to finish jobs consisting of one or two operations as soon as posible in schedule, which would decrease the number of jobs in the simulator and hence might give a better result.

Our results showed that favouring operations from small jobs in the beginning of a schedule might improve the performance, exspecially with an excessive workload. The "small jobs first" fitness showed remarkable results with improvements up to 30%. But with only 10 simulations with each parameter setting, we don't feel we have enough information to give an estimate on how much this penalty function improves the performance, hence further tests are needed.

The results obtained with the "large operations first" penalty function where mixed. With a relaxed workload the penalty function actually increased the mean tardiness by up to 30 %, while it actually yielded excellent improvements with a excessive workload. Despite the latter we dont think any improvement in performance could be gained from this function.

There remains nummerous areas for future research. First of all, the actual effect of the "min jobs first" penalty function should be tester further, since 10 simulation isnt enough to give an estimate on how well the function perfomance compared to other function e.g. the Branke-Mattfeld fitness. Another interresting field for further invistegation is the combination of the Branke-Mattfeld fitness with the "min jobs first" penaly, since the latter actually rewards early idle-time, hence a combination is likely to yield even better results.

References

[1] T. E. Morton A. P. J. Vepsalainen. Priority rules for job shops with weighted tardiness costs. *Man. sci.*, 33(8):1035–1047, 1987.

[2] C. Bierwirth, D. C. Mattfeld, and H. Kopfer. On permutation representations for scheduling problems. *Lecture Notes in Computer Science*, 1141:310–??, 1996.

[3] Christian Bierwirth and Dirk C. Mattfeld. Production scheduling and rescheduling with genetic algorithms. *Evolutionary Computation*, 7(1):1–17, 1999.

[4] J. Branke and D. C. Mattfeld. Anticipation in dynamic optimization: The scheduling case. In Hans-Paul Schwefel Marc Schoenauer, Kalyanmoy Deb, Günter Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, editor, *Parallel Problem Solving from Nature - PPSN VI 6th International Conference*, Paris, France, September 16-20 2000. Springer Verlag. LNCS 1917.

[5] Teofilo Gonzalez and Sartaj Sahni. Open shop scheduling to minimize finish time. *Journal of the ACM*, 23(4):665–679, October 1976.

[6] Shyh-Chang Lin, Erik D. Goodman, and William F. Punch. A genetic algorithm approach to dynamic job shop scheduling problems. In Thomas Bäck, editor, *Proceedings of the 7th International Conference on Genetic Algorithms*, pages 481–488, San Francisco, July 19–23 1997. Morgan Kaufmann.

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

Evolving Robot Behaviours

Martin Knudsen, Lars Nielsen, and Tomas Toft

Abstract— **The objective of this paper is to describe the development of three Khepera robot behaviours using a genetic algorithm (GA). The behaviours were evolved using a custom designed simulator and transfered to the real world. The evolved behaviours all performed satisfactory in the simulator. However, only one transferred successfully to the real world, mainly due to oversimplified aspects in the simulator. It is concluded that using elitism seems to improve GA performance when confronted with noisy, dynamic fitness functions.**

1 Introduction

The original aim of the project was to evolve a robot controller for playing robot soccer and transfer it to the real world. The idea for this was to use a behaviour based approach with a central decision maker and several simple behaviours. The decision maker was planned as a motivation network, however due to lack of time it was not implemented. The behaviours could be either hand-coded or evolved.

For evolving, two possibilities were considered: genetic programming or evolving parameters for a hand-coded behaviour with a genetic algorithm (GA). We chose the latter for this project.

The reason for not using genetic programming, was that we had a reasonably good idea about the optimal algorithm (find ball, goto ball, shoot ball into opponents goal). For the behaviours it was the same, we had an idea, but needed to tune the parameters. We did not want the GA to spend time "reinventing the wheel".

Having decided on implementing and running only the simple behaviours, we looked for a simulator, but found none that met our demands. They were either too specific (fine tuned for one task) or too general (modelling details to a degree we did not need, while leaving out details important to us). In both cases we would need to extend the simulator to fit our needs. Thus we decided to implement a simulator of our own. The simple behaviours were then evolved in the simulator, and finally transferred to the real world.

2 The world

This section describes the real world environment of this project. Furthermore we discuss the use of genetic algorithms for real world problems.

2.1 The Khepera, ball and environment

Figure 1: The Khepera robot (5 cm. dia.)

Figure 1 shows a Khepera robot. It is approximately 5 cm. in diameter and has two wheels with separate motors. The square shaped chip on top of it is a processing unit that powers a real time Operating System. The robot communicates by serial communication with the outside world, but can run fully autonomous using on-board batteries as well. The basic module has only one type of sensor that measures the proximity of objects by means of reflected light. 8 proximity sensors are placed on the side of the robot, 6 distributed on the front and 2 placed to the rear. Several plug-in modules exist, among which a 64 pixels B/W camera is the most interesting with respect to soccer playing robots. Each pixel has a resolution of 8 bits going from dark to bright.

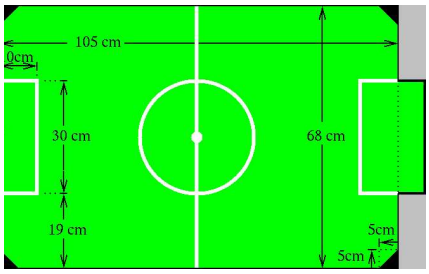


Figure 2: The soccer pitch

The bounding world of the Khepera robot consists of a soccer pitch as depicted on figure 2. The pitch is lit by two powerful projectors casting indirect light ensuring close to uniform light conditions. The ball used during soccer matches is a yellow tennis ball. Competing robots wear black and white stripes to distinguish them from walls, goals, and the ball.

2.2 The GA and the world

Since the aim of this project is to evolve robot behaviours using a GA, the question of how to calculate fitness of individuals becomes urgent. One could test and evaluate each behaviour manually using a real robot, ball, etc. However, time consumption alone (several minutes per evaluation) makes this approach intractable in practice. The counter strategy is to perform fitness calculation using a computer simulation of the real world. The main purpose of building a simulator is that it can greatly shorten and automate this process. It might be an idea though to combine the two approaches by letting a few generations of the GA be performed using real robots now and then, in an attempt to minimise the 'reality gap'. This gap is traditionally the major source of error in simulators since it is virtually impossible to model noisy physical environments accurately. We have chosen to rely fully on a simulator for evolving the behaviours.

2.3 Transferring from simulator to reality

Deciding to use a simulator to evolve controllers for the robot, we now face a new difficulty: *Transferring the evolved controller to the real world.* Even though the evolved behaviour works perfectly in the simulator, this does not have to be the case when the same algorithm is run on a robot in the real world.

Nick Jakobi gives an approach in [2]. The first difficulty one encounters, is that the simulator cannot possibly hope to simulate everything. One must restrict oneself to simulate only the important aspects. But even these cannot be accurately modelled.

The second difficulty is whether the simulated aspects are correctly modelled. Or at least modelled in such a way, that they can be used in a similar way as in the real world. Furthermore we must be absolutely sure, that the evolutionary algorithm will not evolve anything based on any part of the simulator, e.g. always starting pointing *exactly* towards the ball (which cannot be reproduced in reality).

One way to get around that reality might be modelled incorrectly is to add random noise. But even this has to be done with caution, as this adds a new aspect, that the genetic algorithm can use, if it is given the chance. Several runs of the simulator with different degrees of noise have to be carried out to compensate for this. Moreover, reality *is* a very noisy place and if this noise is not modelled somehow the evolved controller will end up behaving more or less randomly, and will (probably) not exhibit the desired behaviour.

Facing all these difficulties we begin designing the simulator.

3 The simulator

The simulator is a so called minimal approach designed to model only the most important issues in the environment of the robot. It has both a graphical and a non graphical mode. The graphical mode is used mainly for debugging, i.e. an intuitive way of validating the simulator and checking the performance of behaviours. The non-graphical mode is for calculating fitness during GA runs, since the massive overhead due to painting etc. is redundant here.

3.1 Implementation

The simulation is implemented using a discrete model. I.e. time is divided into so called time steps. For each time step new positions of the

moving objects are calculated by taking into account physics like speed, acceleration, direction, energy, bouncing, and pushing. Since the real world is a continuous environment the accuracy of this approximation correlates with the length of each time step. The shorter the time step the better the approximation. We have chosen a time step length of 20 ms There are several reasons for this:

- The scheduler of the real-time OS on board the Khepera robot works by assigning each process a 5 ms time slot. Beside the controller process, processes for updating camera and other sensor inputs are needed. 20 ms time steps hence allows for 3 additional processes.
- The fastest sensors (proximity sensors) update with 20 ms intervals, hence making it redundant to process any faster.
- The camera updates at a maximum of 50 ms intervals. Having a time step length equal to this or longer has the effect that the camera is up to date in all calculations. This is not the case in the real world.

Each time step the position of moving objects are updated, the different sensors are read if they are ready, and finally the robot controller is called to convert the new inputs into actions (setting the speed on each wheel). The modelling of physics of both sensors and motors has been based on the specifications in the Khepera User Manual [3].

Up until now it is all quite simple. However, in the real world the different types of physics modelled by the simulator are very noisy and time varying. How should one cope with that? We have chosen to use two strategies:

1. Ignore the aspect in the simulator, and assume that some filter mechanism deals with it in real life.
2. Approximate the physics and add plenty of random noise to ensure controllers cannot rely on them too much.

Among the aspects ignored are changing light conditions. They affect the readings of both camera and proximity sensors. However by assuming a filtering or calibrating module in the sensor update processes, these aspects will be transparent to the controller.

Among the aspects modelled in the simulator are the unpredictable movement of the tennis ball and the readings of the proximity sensors. The fact that the ball is a tennis ball makes it extremely difficult to model correctly, since tennis balls have a characteristic lowered line shaped like an 'S' going around it. When moving slowly this has the effect that the ball can change direction suddenly. However, it is possible to approximate this behaviour by changing the direction by a random amount of degrees with random intervals. At least this makes it difficult for the controller to rely on a perfect line of movement by the ball.

The proximity sensors are not modelled truly correct. In the real world they measure proximity as a function of distance and angular deviation to an object and its colour. Our simulator does not take the angular deviation into account but distance and colour alone. The effect is that sensors deliver values that are increasingly more wrong the more objects deviate from the sensor angle. This imposes a clear potential to lower the chances of success when transferring the evolved behaviours to the real world. This lack in accuracy is definitely a candidate for future work.

Apart from these examples, focus has been on adding adjustable noise to as many physical aspects as possible to enable us to avoid evolved controllers adjusting their parameters according to errors in the simulator rather than to characteristics of the real world.

4 Simple behaviours

This section deals with the simple behaviours that we developed for use in the simulator. The main structure of the simple behaviours are hand-coded. But each behaviour is specified by a set of parameters such as thresholds, speed settings, and other values. We used the parameters as genes and used a GA to evolve them.

4.1 The behaviours

The behaviours we have decided to focus on are *follow-wall*, *dribble*, and *obstacle avoidance*.

They are not too difficult to hand-code in the real world, but the aim here is to evolve a controller for the behaviour, thus avoiding having a human spending time fiddling with the parameters. The objective is to make the robot behave sensibly in the real world, not just in the simulator. Another objective is to examine how to define fitness functions. As we shall see later on in section 6, this is far from trivial.

Follow wall

This behaviour guides the robot along a wall. If no such wall is found, it moves around at maximum speed. The idea is described in figure 3.

```
forever {
  if (too close to the wall)
    turn away from the wall
  else if (too far away from the wall)
    turn towards the wall
  else
    drive straight forward
}
```

Figure 3: The follow wall behaviour

All we need to do now is to define *too close to*, *too far from*, etc. (see figure 3) as functions of the sensor readings and perhaps some internal state, such that they can be used as genes.

Dribble

The objective of this behaviour is to dribble a ball as straight and as fast as possible without losing touch of it. The behaviour is controlled by parameters such as threshold values, maximum speed values etc. The algorithm is described in figure 4

Obstacle avoidance

The *obstacle avoidance* behaviour is "randomly" walking around an area (the soccer field) with

```
forever {
  if (the ball is not near us OR
      the ball is right in front of us)
    go straight
  else if (the ball is to the right)
    swerve to the right
  else
    swerve to the left
}
```

Figure 4: The dribble behaviour

several round, non-mobile objects. The objective is to move as fast as possible without hitting walls or any of the objects.

```
forever {
  if (no obstacles near us)
    go straight
  else if (more obstacles to the left)
    turn to the right
  else
    turn to the left
}
```

Figure 5: Simple obstacle avoidance

The (easy) algorithm for doing this is making a Braitenberg creature[4]. The algorithm (in pseudo code) can be seen in figure 5. The general idea is that if we see something in front of us, we decrease the speed of the wheels (possibly resulting in backward motion). Most weight is placed on the front sensors, less on the sides, and none on the rear ones.

To improve the robot, we have added a simple memory to the program, that remembers if we are turning left or right, or just driving straight. The purpose is to let the robot escape narrow, dead-end corridors.

The code for this algorithm is seen in figure 6. It uses the simple obstacle avoidance mentioned above.

5 Genetic algorithm

To evolve the various behaviours we have chosen to use a standard genetic algorithm. Some reasons for this choice are both simplicity and

[4]Braitenberg creatures are simple feedback controllers, that transform sensor input directly to actuator output (e.g. setting the speed of a motor) [1]

```
forever {
  if (not too much in front of us)
    simple obstacle avoidance
  else { // turning
    if (last decision was not turning)
      decide turning direction
    turn in turning direction
  }
}
```

Figure 6: Advanced obstacle avoidance

speed of the algorithm. Besides in our problem domain of robotics it is not of paramount interest to get the exactly optimum solution, which a more complicated GA would have a better chance of finding. An almost optimal solution is good enough for our application, due to the nondeterminism of the fitness functions.

5.1 Implementation

We use a real-encoded EA. One round of the EA is described in pseudo-code below.

```
doRound()
{
   evaluate();
   generation++;
   select();
   crossover();
   mutation();
}
```

We use a tournament based selection scheme. Two individuals are chosen randomly to compete. The one with the best fitness survives to the next generation. This implies that more than one copy of a good individual can survive - i.e. better individuals have a higher chance of getting more copies of their genes to the next generation.

Alteration of the population consists of first applying a crossover operator and then a mutation operator. The crossover operator is a N-weight arithmetic crossover (i.e. a different weight for each gene) We tried to use both elitism and non-elitism. This was done because we would often loose the best individuals.

The mutation operator used is a linear decreasing Gaussian mutation.

6 Fitness functions

Fitness functions are of great importance and the design should be considered carefully. There is a difference between fitness in the simulator and the real world. In the simulator fitness can be calculated numerically whereas in the real world this is not an option. These issues as well as fitness functions for the simple behaviours will be discussed below.

6.1 Simple behaviours

The following fitness functions were used for the simple behaviours in the simulator. All fitness functions are calculated as a weighted sum of a number of terms. The fitness values are actually punishment values, hence the GA tries to minimise the fitness. Each weight determines the importance of one term :

$$\delta_{fitness} = \sum_{i=1}^{n} w_i t_i$$

Follow Wall

The behaviour aims at making the robot follow the wall at a constant distance as fast as possible without bumping into it. The robot is placed close to the side wall facing it at a 30 degree angle. The fitness is calculated over a period of 1 minute. At each time step the following terms are updated and added to the total fitness:

t_1 = Deviation in distance to wall within the last second.

$$t_2 = \begin{cases} 1 & \text{too close to wall} \\ 1 & \text{too far from wall} \\ 0 & \text{otherwise} \end{cases}$$

t_3 = Deviation from maximum speed.

Dribble

The fitness of the *dribble* behaviour depends on its ability to dribble the ball straight and fast. The robot is placed facing the goal with the ball right in front of it. The run ends when the ball touches the end wall or a maximum of 30 seconds have elapsed. The terms are calculated at the end of the run and depicts how

well the behaviour has solved the task.

$$t_1 = \begin{cases} dist_x - dist_{xmin} & dist_x > dist_{xmin} \\ 0 & \text{otherwise} \end{cases}$$

$$t_2 = \begin{cases} dist_y - dist_{ymin} & dist_y > dist_{ymin} \\ 0 & \text{otherwise} \end{cases}$$

$t_3 =$ time elapsed

where $dist_x$ is the distance to the end wall, and $dist_y$ is the deviation from the straight line.

Avoid obstacle

The main aim of the behaviour is to avoid bumping into objects, but at the same time to move around as quickly as possible. Hence the fitness parameters concern distance to objects and robot speed. The robot is placed in the football arena with a number of randomly distributed round objects of random size. Each time step the following terms are updated and added to the total fitness:

$$t_1 = \begin{cases} 1 & dist_{obst} < dist_{collision} \\ 0 & \text{otherwise} \end{cases}$$

$$t_2 = \begin{cases} dist_{close} - dist_{obst} & dist_{obst} < dist_{close} \\ 0 & \text{otherwise} \end{cases}$$

$t_3 = speed_{max} - speed_{cur}$

where $dist_{collision}$ is the minimal distance to an object defined as a collision.

$dist_{close}$ is the minimal distance the robot is allowed to be near an object, before it is punished.

6.2 Conflicting objectives

The main problem with designing fitness functions in this domain is that the problems are actually highly multi-objective. They all share a conflict between precision and speed. One needs to decide beforehand the importance of each term, and this adds to the list of parameters that needs tuning before running the GA.

We ran into a very illustrative example of the importance of these weights during the evolution of the *follow wall* behaviour. We accidentally put too much weight to the term t_1 describing how good it was at keeping a constant distance to the wall. This (of course) resulted in an evolved behaviour were the robot stopped at an ideal distance to the wall and just stood still. All behaviours that actually dared to move were punished heavily for deviating even the slightest bit in distance in comparison to the relatively small reward for driving faster.

6.3 Fitness in the real world

In the simulator a fitness function can be designed and calculated objectively. When transferring a behaviour from the simulator to the real world, one has to evaluate the behaviour in the real world. This is done in order to determine both the ability of the simulator to model the real world and also of course to compare evolved solutions with manual solutions. Evaluations in the real world have to be done in a more subjective way in our problem domain. This further complicates the transfer from simulator to reality.

7 Results

This section describes the results obtained by running experiments in the simulator and in the real world.

7.1 Results from the simulator

We ran the various behaviours in the simulator, evolving them using the GA. We tried both with and without elitism (the 3 best individuals of each generation will survive). One important thing should be noted here, which can also be seen on the graphs for the best fitness. Since there is some random noise added to the simulator, it is not guaranteed that an individual with the same genes will obtain the same fitness in two different runs. This is also why the graph for the best fitness using elitism is not strictly decreasing. The best individuals are copied to the next generation, but it might be that they obtain a worse fitness, even though the genes are the same, due to the random noise.

20 runs of the GA were performed and best fitness and average fitness of each generation

were calculated. Each individual behaviour was run 5 times in the simulator with random noise and the average fitness was used.

Dribble

See figure 7 for a graph showing fitness and table 1 for GA settings for the evolution of the *dribble* behaviour.

mutation rate	0.3
crossover rate	0.8
population size	50
number of generations	50

Table 1: GA settings for evolving *dribble*

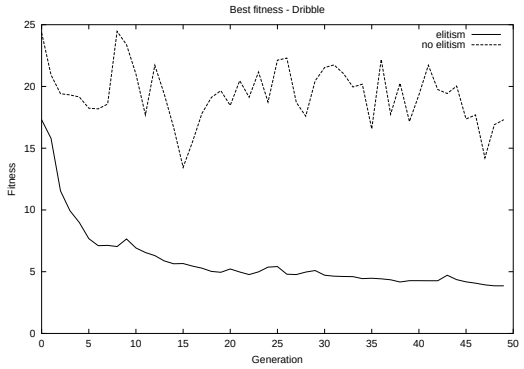


Figure 7: Graph showing best fitness for dribble.

Follow wall

See figure 8 for a graph showing fitness for the *follow wall* behaviour and table 2 for GA settings.

mutation rate	0.3
crossover rate	0.8
population size	50
number of generations	50

Table 2: GA settings for evolving *follow wall*

Avoid obstacles

See figure 9 for a graph showing fitness for the *avoid obstacle* behaviour and table 3 for the GA settings.

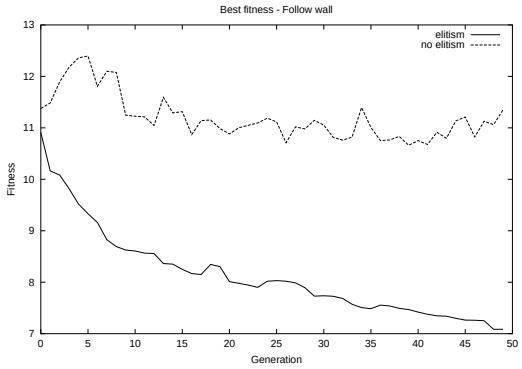


Figure 8: Graph showing best fitness for *follow wall.*

mutation rate	0.3
crossover rate	0.8
population size	25
number of generations	25

Table 3: GA settings for evolving *avoid obstacles*

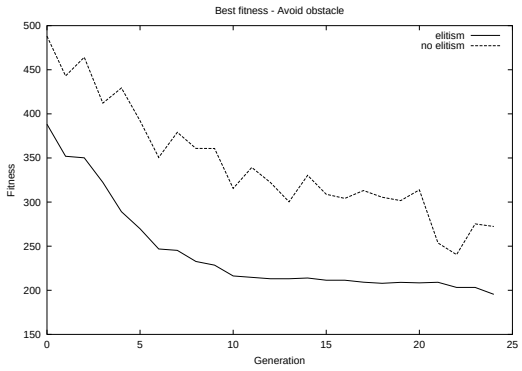


Figure 9: Graph showing best fitness for *avoid obstacles.*

7.2 Results from transferring to real world

We tried to transfer the best of the evolved individuals to the real world. This was done for each of the behaviours described above. The transfer tests the precision of the simulator and the quality of the evolved behaviours. All behaviours were transfered directly to the Khepera Robot and run in the real environment.

- *Dribble*

  The transfered version of this behaviour performed very poorly in the real world. It could not follow the ball for very long periods of time - often it seemed that

it had problems detecting the ball. The primary reason for this is that in the real world the ball is a 3D object and in the simulator the ball is 2D round object, which in 3D would correspond to a cylinder instead of a sphere. So since the sensors of the robot are near the floor, and the ball is of course round, this means the equator of the ball is closer to the robot than the bottom and top of the ball. This would then make the ball seem farther away than it really was. Another problem is the incorrect modelling of the proximity sensors in general.

- *Follow wall*

  The main problem with the transfered version, was that the robot would sometimes pull too far away from the wall. The wall would then be too far away from the proximity sensors to detect, which would make the robot lose track of the wall and stop following it. One major reason for this, is the modelling of the proximity sensors, that is not precise enough, as described in section 3.1.

- *Avoid obstacles*

  We observed that this behaviour performed very well in the real world. Actually it was much better than a simple hand-coded obstacle avoider and much faster. The only small problem could be seen if some of the obstacles were balls, then the robot would sometimes touch them. But this is due to the same reason with the modelling of the ball as described with the dribble behaviour. The evolved behaviour was quite impressive to watch in the real world.

8 Conclusion

We successfully evolved all three behaviours in the simulator. Elitism, although usually considered leading to premature convergence, in our case lead to keeping fit individuals alive, particularly in the face of a nondeterministic fitness function (due to noise). This seems to lead to better individuals faster, without losing too much diversity. In particular, considering the graph for the non-elitism runs of the *follow wall* behaviour (figure 8).

Concerning the simulator, we must conclude that implementing one is difficult. Trying to design a minimal simulator quickly ends up in fiddling with many parameters, and the end product is far from minimal or simple.

Transferring the controllers to reality was less successful than the evolution of controllers in the simulator. Some of the problems are due to oversimplifications in the simulator. The noise in the real world is different than the noise we modelled and the sensors reacted very differently in the simulator than in reality.

However the *obstacle avoider* transferred beautifully. We believe, that this was possible because it does not depend so heavily on the imprecisions mentioned above. Furthermore the *avoid* behaviour is less complex than the *dribble* and *follow wall* behaviours, which are both regulating tasks.

The fitness functions also proved to be an important area. Poor design of these easily lead to evolving behaviours different from the intended. The reason for this is, of course, that although it is easy to describe the problem to another human, giving a precise, mathematical definition is far from trivial. More so the objectives we found are actually combinations of several, conflicting sub-objectives leading naturally to a problem of multi-objective optimisation.

9 Future work

- **Improved simulator :** Better modelling of the proximity sensors and the ball!

- **Motivation network :** It would be nice to put together all the different behaviours into a fully functioning robot controller. One way of doing this that we looked into is to use a motivation network as described by T. Krink in [4]. The idea is inspired from biology and the basic idea is to map the robot sensors and internal states to a number of motivation variables. These mapping functions are specified by some genes which can be evolved. Each motivation variable corresponds to a behaviour. The behaviour

with the highest motivation factor is then chosen and executed.

- **Advanced GAs :** Using more advanced approaches than a simple GA might improve the performance. A co-evolutionary approach or an island model might be useful to improve the generality of behaviours.
- **Multi objective optimisation :** It might prove very useful to attack the problem of conflicting objectives by using a more general optimisation technique than manual parameter tuning.
- **Manual runs of the GA :** Running the GA on real robots in the real world for the last few generations means that we don't have to fine tune the controller in the simulator. This might reduce noise problems in the real world.

References

[1] David W. Hogg, Fred Martin, and Mitchel Resnick. Braitenberg creatures. Technical Report TR, MIT Media Labs, 1991, 10 pages, 1991.

[2] Nick Jacobi. Evolutionary robotics and the radical envelope of noise hypothesis. 1997.

[3] K-Team. *Khepera User Manual*, 5.02 edition.

[4] Thiemo Krink. Motivation networks - a biological model for autonomous agent control. *Journal of Theoretical Biology*, 2000.

Whist With a Twist

Karsten S. Jørgensen, Martin E. Jørgensen, and Mads B. Enevoldsen

Abstract— **This project aimed at producing a computer player for Ligeud, a variant of the Whist card game, using an evolutionary algorithm. Two different representations of the computer player strategies were used with the algorithm. Both used fuzzy logic to make decisions throughout the game, but one proved decidedly better in evolving.**

1 Introduction

Whist is a family of card games along the following lines: in each game, the four players compete to win the bid. The highest bid wins, and the winning player decides the trump suit and possibly also the card whose owner will be his partner for this game. Play starts, and the bid winner and his partner now must collect the number of tricks stated in the bid. If they succeed, they win some money (or points), if not, they lose.

For the purpose of this project, we chose a local variant of Whist, called "Ligeud". The aim was to create a computer player for Ligeud, using an evolutionary algorithm. This report describes two different representations used to model the computer player strategies, a number of measures introduced to enhance the evolution of strategies, and the experiments conducted.

2 Representations

The first approach was to use a kind of fuzzy logic with a great number of "Strategy Tops" in $\mathbb{R}^n$. The second approach used an arithmetic tree to represent a function $f : \mathbb{R}^n \to \mathbb{R}$ together with a small constant number of "Strategy Tops" in $\mathbb{R}$.

2.1 Multidimensional fuzzy logic

As described in the introduction, a game of Whist is roughly divided into two parts: bidding and playing. Accordingly, we have divided a strategy into two parts. Let's first consider playing, since bidding is somewhat more subtle, and the representation used is more easily explained for playing.

Playing

The decision to make during play is the following: Given a set of cards and the knowledge of which cards have been played in this game, which card should we play now? Our aim was to describe a function from the state space of all things known to the player to a space of decisions, which could be used to select the best card to play at this point. Instead of selecting a particular card, we divided the possibilities into play-decision categories:

- Smallest
- Just under
- Just over
- Greatest
- Smallest trump
- Trump just under
- Trump just over
- Greatest trump
- Joker

where e.g. "smallest" means playing the smallest non-trump card on your hand, and "trump just over" means playing the smallest trump card you have that is higher than the best card on the table, thus (perhaps) collecting a trick by beating the others by a narrow margin.

At first glance it may seem that there is no need for categories such as "Just under", but a moment of thought shows that it is a very useful category indeed for e.g. the bid "Nole" under which the player can get *at most* one trick.

A play decision category maps deterministically to some card on the player's hand, so the strategy only has to decide what kind (or category) of action it wants to take.

The state space was constructed from a number of relatively simple parameters such as: how many hearts do I have, what is the trump colour, am I absolutely sure that the player across from me is renonce in clubs, which card has the player on my left played in this round, etc. The complete list of these parameters, or dimensions, can be found in appendix 1.1. All in all, this amounted to a 37-dimensional space, so to sum up: Given a point in 37-D, the function must decide upon a play category.

$$f : S \rightarrow \begin{cases} \text{smallest} \\ \text{under} \\ \text{over} \\ \text{greatest} \\ \text{smallest trump} \\ \text{trump just under} \\ \text{trump just over} \\ \text{greatest trump} \\ \text{joker} \end{cases}$$

where $S \subset \mathbb{R}^{37}$

For this we were inspired by fuzzy logic membership functions. Instead of mapping each input variable to a different set of membership "tops", we chose to map all 37 dimensions in our input to a set of membership functions, or "Strategy Tops". Our representation is thus a set of Strategy Tops, each with its own play category, height, slope and placement in 37-D state space. If a given situation, as represented by a point p_1 in state space, is close to a Strategy Top T(category c, height h, slope s, point p_2), the probability of category c will be

$$P(c) = h - dist(p_1, p_2) * s$$

where dist(p_1,p_2) is in essence the Euclidean distance

$$\sqrt{\sum_{i=1}^{37}(p_{1_i} - p_{2_i})^2}$$

but modified by a normalization

$$dimsize_i = upperbound_i - lowerbound_i$$

and a stretch factor since values are restricted to an interval in each dimension, and the intervals have unequal size among the dimensions. For instance, some dimensions are "boolean" i.e. range from no (0) to yes (1) while others are card values which range from 0 to 14. The idea is that a large stretch factor allows us to assign less importance to strategy tops in the given dimension by enlarging the distance between any two points in that dimension. Thus a strategy may evolve different levels of importance for different dimensions.

$$dist(p_1, p_2) = \sqrt{\sum_{i=1}^{37} \frac{stretchFactor_i \cdot (p_{1_i} - p_{2_i})}{dimsize_i}^2}$$

The probabilities for each of the 9 play categories are given by the tallest top of that category at p_1, and the numbers given for all categories are normalized before use. So the output of a "lookup" in the decision space during play is a 9-tuple of probabilities which add up to 1.

The actual decision is a probabilistic choice among the categories, according to he probability found for each play category. This preserves some non-determinism, which we felt was necessary. Ligeud is not a game of perfect information like chess or checkers – a player cannot always know what the other players are holding. Thus, it is not possible to always judge exactly which card is the best to play, and a completely predictable player would probably also be easier for a human opponent to beat.

Bidding

Bidding proved to be harder to evolve, perhaps because an important aspect of it is gambling.

We took the same approach as to playing, choosing a set of decision categories for bidding just as we did for playing: pass, nole, low, high, low clubs, and high clubs. E.g. bidding "low" would choose the lowest non-clubs bid that is higher than the last bid (since you must raise the bid of the last player, or pass).

The dimensions chosen can be found in appendix 1.2.

2.2 Arithmetic tree functions

The multidimensional sets of StrategyTops soon showed a preference for very large sets, taking several megabytes of disk space to describe a single strategy, and slowing the overall process of evolving strategies using the EA. Also, the computation time needed by the MultiDim strategies when playing against a human player were unsatisfactory on slow hardware. The introduction of arithmetic tree strategies were an attempt to speed things up.

Playing and bidding

Playing and bidding each has its own tree composed of nodes and leaves. A node is a common arithmetic operator: plus, minus, times, division, modulo, power, square root, sin, cos, absolute value, min, max, exp or log, and a leaf is either a constant or a variable v_i where $i \in \{1, 2, \ldots, 37\}$.
Given a point in state space, the tree computes a function value which will serve as the one variable mapped to the fuzzy membership functions. For the arithmetic trees, we confined the number of membership tops to one per category. Just as for the MultiDim strategies, the output is the set of probabilities for all the categories described. The tree strategy contained one tree for playing and one for bidding.

3 EA technicalities

3.1 Crossover

Since we had two different strategy representations, there is both a multidimensional and an arithmetic tree crossover. Of course they are only applied if a uniformly distributed random number is less than the crossover rate.

Multidimensional Crossover

We form the offspring of two multidimensional strategies by first choosing randomly a number between 20 to 80 to represent the percentage of strategy tops that should be copied from the first parent into the offspring. Then for the second parent choose a number between 20 and 80 denoting the percentage of strategy tops that will be copied to the offspring. The offspring can therefore contain either more or less strategy tops than its parents.

Arithmetic Tree Crossover

An arithmetic tree strategy consists of two trees and a decision top for each possible decision (9 for playing and 6 for bidding). Each top is specified by three floating point values: a place, a height and a slope. The crossover of the decision tops of two arithmetic tree strategies is simply the decision tops produced by taking the average of the floating point values of the parents' tops.

The two parent arithmetic trees are then crossed by either taking the sum of the two parents and dividing by two (see figure 1) or by choosing a random subtree from the first parent and inserting it randomly in the second.

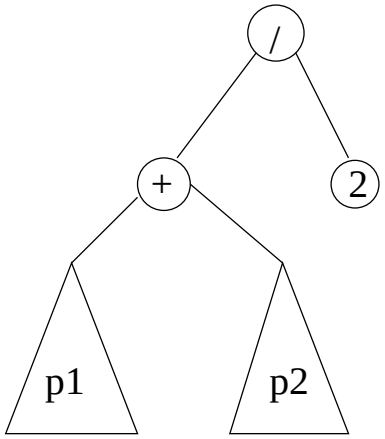


Figure 1: An offspring is formed by the average of the two parents

3.2 Mutation

As was the case with crossover we have two different mutation operators. Each is of course applied according to the mutation rate specified in the EA.

Multidimensional mutation

The representation of a multidimensional strategy contains strategy tops. Each is specified by 39 floating point values: a place in $\mathbb{R}^{37}$, a height and a slope. Each of these floating point values is mutated by adding a Gaussian distributed random variable. Of course if it is one of the coordinate values, the value has to stay within the bounds of the corresponding dimension. So if the value exceeds the bounds it is rounded up or down accordingly. By mutating all the floating points in the strategy, we are then done because now we have a strategy with different tops in $\mathbb{R}^{37}$.

Arithmetic tree mutation

The decision tops of the arithmetic strategies are mutated as follows: First a random number (a Gaussian distributed value with mean 0 and standard deviation 1) is decided on. Then different constants for each of the three kinds of attributes of a decision top (place, height and slope) are multiplied to the random number. This is done to ensure a not too drastic mutation that preserves the feasibilty of the decision top. The result is then added to each individual floating point value specifying the top if the mutation rate tells us to.

The playing tree and bidding tree are each mutated by either swapping two randomly chosen subtrees or replacing a leaf with a new randomly generated subtree. The latter could cause the trees to grow rather explosively, so we definded a maximum size of the trees.

3.3 Selection

Given the complex nature of our strategy repesentation as described in section 2, it is obviously hard to decide at a glance whether a strategy is good or not, much less compute a value to express the quality of a strategy; a simple and absolute fitness value is not readily available. Since we are dealing[5] with a card game, the natural method for selection seemed to be 4-way tournament selection. We pick four random strategies from the current generation and let them play against each other a number of times. The strategy of the most successful player (i.e. who has made the most money) is then copied to the next generation.

In card playing there is quite a bit of luck involved e.g. in the order of the cards after a shuffle. To take this randomness out of the equation it is important that each of the tournaments consist of many games, and certainly at least four. This number of games allows each of the players in turn to be the lead player (the player who is the first to bid and play). Similarly, to make sure that all players have the role as lead player equally often, the number of games should be divisible by four.

Of course, the outcome of the tournament depends somewhat on the relative positions of the players. For instance a certain strategy might be effective if its player is seated to the right of a player with a very aggressively bidding strategy. For this reason the players are placed randomly around the table at the beginning of each tournament.

Then, to make absolutely sure that the absolute position of the players has no influence on the outcome of the game, each tournament is repeated four times. Between these repetitions the players are rotated around the table. At the end the results of all this is added up and the winner is determined.

4 Modifications

The earliest experiments showed a disconcerting lack of convergence towards anything. It seemed that the MultiDim strategies were not much better than a random player, and they kept growing bigger and bigger. Also, in many areas of the state space, they did not have any StrategyTops and thus gave no answers when asked for probabilities of the categories. For these reasons, we introduced some countermeasures that are explained in the following.

4.1 Increasing selection pressure

To raise the selection pressure and force strategies to get better, we added a hardcoded strategy as a player in each tournament. And we added the rule that a tournament won by a hardcoded strategy was invalid. As a result the EA strategies were forced to beat a hardcoded strategy to survive to the next gener-

[5] No pun intended.

ation. The resulting strategies were visibly better even after a few hundred generations. Pushing the envelope, we then tried adding another hardcoded strategy player to each tournament. This did not result in any noticeable improvement. On the contrary, the evolved strategies now seemed more likely to play it safe and pass during the bidding phase, because the two hardcoded players would raise each other's bids, thereby ensuring that no hardcoded strategy got itself a really low bid that was easy to meet. Passing too often during bidding often allows the opponents to get a low bid that is easy to win. When playing against a human player, these strategies lost badly, because they failed to ever win the bid. We consequently returned to having only one hardcoded player in each tournament.

4.2 Limiting strategy size

During the initial experiments, the best playing MultiDim play strategy evolved had over 6000 strategy tops, taking 4.5 MBs to represent in a clear text file format. On average performance hardware, it was too slow to play against a fairly patient human player. To reduce the size of strategies, we introduced a maximum number of StrategyTops, killing off least used tops when the number increased beyond the maximum. In this way, we kept file sizes down to approximately $\frac{1}{4}$ MB per strategy, or approximately 700 tops.

4.3 Greedy learning

Another attempt to improve strategies was to use greedy learning when playing. After each card played, the strategy was told whether it was a good or a bad move, i.e. whether it resulted in earning a trick for itself or its partner or not. The StrategyTops were raised or lowered accordingly, creating a new one whenever necessary. This caused an explosion in the number of StrategyTops, emphasizing further the need for a maximum number of tops. However sound the greedy learning idea may seem, we have not been able to show that it had any beneficial impact on the strategies evolved. In fact we have conducted a couple of experiments that demonstrates the failure of greedy learning – see section 5.2.

4.4 Punishing the clueless

Because our MultiDim state space is so enormous, strategies would sometimes find themselves at a point in the state space where there was no strategy tops in sight. To make sure this would not happen too often, we introduced punishments for not giving a useful answer (i.e when $P(c) = 0\%$ for all categories c). The method was to subtract a small amount from the strategy's total winnings every time it returned an all-zero answer. The net effect of this on the evolved strategies is hard to estimate, but a visible side effect was that the winnings were much lower for all strategies in every generation, in fact often below 0 $. This suggests that the strategies kept on having "no-answer" areas in state space despite the punishments, and we suspect that a really good MultiDim strategy that never gives all-zero answers must be much larger than we allowed through the "maximum number of StrategyTops" countermeasure. However, the latter was a practical necessity in terms of computing speed and disk space requirements, for our project to proceed at DAIMI.

4.5 Injecting fit genes

Because our strategy representation caused the individual strategies to grow very large, our evolutionary algorithm as a whole was quite slow. So instead of starting from scratch every time the algorithm was run, we made it possible to "inject" better genetic material in the starting generation. This had two advantages: The injected strategy could continue its evolution and hopefully improve further, and the fierce competition would motivate the infant strategies to reach a high fitness level quicker than before.

4.6 A human teacher

Equipped with the ability to inject fit individuals into the population one may ask "So, what individuals should we inject?". Of course it could be a strategy that resulted from a previous EA run, but there is another option. You could train it yourself. In the Ligeud GUI it is possible to enter the tutor mode, in which every decision made by the human player is

recorded and used to train a multidimensional strategy. This strategy can then be injected into the population or played against immediately.

The tutor mode features do work, but can probably not be used in practice to create a full-fledged strategy from scratch. The vast number of different playing situations in a game calls for an equally vast number of recorded decisions before the strategy is able to play without being embarrasingly inadequate. This requires an extremely patient teacher with nothing better to do than play Ligeud for a month or two.

5 Experiments

5.1 Different setups

Throughout all of our experiments we have used a variety of different experimental setups in an attempt to get the best results as quickly as possible.

To co-evolve or not to co-evolve

One of the thoughts behind the introduction of the arithmetic tree strategies was to try an approach inspired by co-evolution. This meant having two subpopulations: one consisting of multidimensional strategies and one consisting of arithmetic tree strategies. In each tournament the different kinds of strategies would then fight it out, and hopefully this additional diversity in the competition would lead to better strategies.

Unfortunately these experiments were in general not as successful as their counterpart: the completely separate subpopulations. It was almost always the case that at a given time every member of one subpopulation were more successful than all members of the other subpopulation. So the expected synergy effect remained unseen.

Testing specific attributes

Many times during our experiments we have had to test one specific attribute of a strategy. For instance, the ability of a multidimensional strategy to make good playing decisions without learning during play or using punishments. To target such attributes precisely we have made it easy (for us) to specify when to use hardcoded decisions and when to use those of the evolved strategy. So in the example just mentioned, we would create an all-multidim population, set punishments to 0, switch off the greedy learning, and make the players use hardcoded bidding.

The performance of "partial" strategies obtained this way have overall been consistent with the performance of the combined combined effect. For example, a multidim strategy with well evolved bidding ability and hardcoded card playing and another multidim strategy with hardcoded bidding and well evolved card playing can be cut-&-pasted into a player that is just as good as (and often better than) a multidim player who has evolved bidding and playing simultaneously.

Manual testing

At the end of the day, the evolved strategies are meant to be played against via the Ligeud GUI. So after a suitable number of generations of evolution the final test of the quality of a strategy ensues in the GUI. The quality measure applied here is of course the impression of its skills that a well versed human player gets when playing against it.

5.2 Testing greedy learning

We conducted two experiments to show the impact (or lack thereof) of the greedy learning explained in section 4.3. First we run a normal MultiDim evolution for 300 generations (population size 25, mutation rate 0.8, crossover rate 0.8) with greedy learning switched off. Then we run the exact same evolution with greedy learning switched on. The quality measure used is a Bezier smoothed count of the number of hardcoded wins. The discouraging results can be seen in figure 2 in appendix 2.1. Notice that the greedy learners

show no sign of having an advantage over the other players.

In the second experiment we let a hardcoded player, a player making completely random choices at all times and two completely "empty" MultiDim players compete for 3000

tournaments, each with four games. In the beginning, the "empty" players have no strategy tops at all, but they are willing to learn. That is, the only way they can get new strategy tops is by greedy learning. During the run we keep track of the number of games won by each player. The results are depicted in figure 3 in appendix 2.1. Notice that the learners perform no better than the utterly random player.

5.3 Tuning EA parameters

Purpose and setup

To determine the optimal settings for the mutation and crossover rate we conducted systematic experiments. The resulting graphs can be found in appendix 2.1. What we needed to run these experiments was a good quality measure on an entire generation, so that we could measure the effect of specific mutation and crossover rates after a fixed number of generations. But because no absolute fitness measure for an individual exists it is not possible to just use the average fitness of the generation as the quality measure. Instead we set up the experiment with one hardcoded strategy in each tournament (as described in section 4.1). The quality measure was then the number of times the hardcoded strategy won a tournament – averaged over the number of tournaments held. The idea being that if an entire generation is of high quality, its members should be able to beat the hardcoded strategy more frequently.

Of course this does not ensure that EA generations with good (low) scores will be able to play well against *any* opponent. However, the hardcoded strategy is actually half-decent, so the quality measure is good enough for our purposes.

The experiments were run on a population of 30 individuals for 100 generations without the punishment of section 4.4 which could distort the results. First the bidding and playing abilities of multidimensional strategies were tested in separate runs, then the same was done for arithmetic tree strategies. The result is an average of 5 counts of hardcoded wins.

Results

The results of evolving the playing ability of multidimensional strategies are depicted in figure 4 in appendix 2.2. It is readily apparent that the best results are obtained by keeping a high mutation rate. This could be due to the immensity of our state space. So for quite a number of generations it probably pays off for strategies to explore the state space as much as they can. A high mutation helps achieve this. For the crossover rate the picture is quite muddled, but the lowest (i.e. best) value is at the high end of the scale. However, with only this 5 by 5 grid[6] to judge from, it is hard to draw any solid conclusions.

The results of evolving the playing ability of arithmetic tree strategies can be found in figure 5 in appendix 2.2. Also in this case the best values are achieved by a high mutation rate, probably for the same reason as above. Here it is quite clear that crossover rates in the area of 0.7 are superior when combined with the right mutation. This all seems quite reasonable.

What may seem less reasonable is the corresponding graphs for the bidding phase. They can be found in appendix 2.3. It appears that in both figure 6 and 7 the mutation rate does not matter much compared to the crossover rate, except in the extreme cases (corners of the graph). The crossover rate on the other hand really makes a difference. Both graphs have significantly lower (better) points when the crossover rate is low.

This may be an indication that our crossover operator is perhaps too drastic or needs some kind of improvement. Alternatively the phenomenon might be an inevitable consequence of the way we measure. A low crossover rate means that a higher percentage of a generation is winners of tournaments. To be a winner of a tournament you must beat a hardcoded strategy. So you get the best short term results (because our EA is extremely time consuming we measure after only 100 generations) by just having a population of strategies that have won a lot of tournaments. Why the bidding phase is more sensitive to this than the

[6] Which, sorry to say, took 5 powerful PCs more than 36 hours to produce.

playing phase is not clear.

5.4 Evolving strategies

As mentioned, the Multidim strategies fared better than the ArithTrees when used in the EA context. The trees never really improved visibly on the number of hardcoded wins. Figure 8 shows the number of hardcoded wins in a 1000-generation run of the EA with a pure Multidim population. The population started from randomly generated individuals and played 4 times 4 games in each tournament. Figure 9 shows a corresponding graph for a run with a pure ArithTree population. This population had some previously evolved trees (the best from several previous runs) injected in the first generation and played 8 times 4 games in each tournament. This double up in games per tournament in relation to the multidim experiment accounts for the difference in the number of hardcoded wins: approximately 200 for multidim and 400 for trees in the beginning of each run.

The two graphs show, however, a significant improvement over the generations in the multidim case whereas the trees do not seem to improve at all. This difference we believe reflects not only the differences in experimental setup, but a true difference in strength of the evolved multidims and arithmetic trees. Our belief is supported by the general impression left when loading and playing against the individuals from these and from other runs, but also by the following comparison experiment.

5.5 Comparing performance

To get an idea of how the overall performance of strategies is, we tested four strategies of different kinds against each other over 3000 tournaments each consisting of four games. There was thus no evolution or learning involved. The chosen strategies were:

- the hardcoded strategy,
- the oldest strategy from generation 850 of the multidim evolutionary run described in section 5.4,
- the oldest Arithtree of generation 900 of the tree evolution described above, and
- a strategy playing at random

Both the multidim run and the arithmetic tree run described in section 9 evolved both playing and bidding strategies simultaneously, but in the following comparison experiment, bidding was set to hardcoded, because the evolved bidding strategies were just too awful to allow them to win anything.

The play strategies' percentages of total wins converge towards levels of 54%, 23%, 12%, and 8% respectively [7]. The hardcoded strategy still by far outperforms the evolved individuals, even when they adopt hardcoded bidding. Interestingly, though, the multidim shows some respectable playing compared to the tree, which given time only just manages to distance itself from the completely random strategy.

6 Conclusion

The goal of evolving a computer player for Ligeud has been achieved, albeit not a very competent player. The two representations of strategies, multidimensional fuzzy logic membership functions and fuzzy arithmetic tree functions, showed a noticeable difference in performance after evolving in the EA.

This points to problems with the EA operators of crossover and mutation for our arithmetic trees, which is perhaps not surprising. A function built from arithmetic operators as described may be changed drastically by only a simple crossover or mutation operation, leading to the erratic changes in quality and lack of convergence seen for our trees.

The multidimensional strategies in turn responded better to the manipulations of evolution and actually improved over a number of generations. Still, a simple hardcoded strategy by far outperforms the multidims both in games won and in computing speed. The latter is the subject of possible optimizations.

6.1 Future work

Because of the slow pace of our EA, we first and foremost need to conduct more tests, more

[7] which sums to 97% due to a minor integer round down error

experiments, more tinkering with parameters and settings if we want to evolve a really capable Ligeud player.

Minor adjustments

There are some minor adjustments that we did not have time to try in our experiments. One is adjusting the mutation rate during a run according to a diversity measurement on the population, so that low diversity would imply high mutation and vice versa. The need for this is also suggested by our mutation and crossover tuning - lots of mutation is necessary in the beginning, but we expect that this may not be the case later on. We have implemented a diversity measure based on the average distance between all the tops of all the players in a population, but without test results we cannot know whether this is an adequate measure or not.

Another minor adjustment could be done on the multidimensional crossover operator: instead of selecting the tops to copy to the offspring randomly, the most frequently used tops could be chosen. This would hopefully improve the quality of the offspring so that the crossover operation could repair its

Trimming the tops

One unsolved problem in the multidimensional representation is that strategy tops can be strongly overlapping or even contained in one another. This could be remedied by merging tops that are within some fixed proximity range of each other. This would also help keep the number of strategy tops down.

Rethinking representation

Finally and most drastically, to improve the overall performance of our EA it would be a good idea to try to find a lighter version of the current multidimensional strategy while sustaining (or even improving) the current results.

One idea for an alternative representation could be one along the following lines: Instead of an arbitrary number of strategy tops placed in $\mathbb{R}^{37}$, a fixed number of tops could be placed in $\mathbb{R}$ for each dimension. The value of each dimension would then map directly to one fuzzy membership function deciding between the 9 (or 6 for bidding) play categories. In this manner each dimension would opt for a decision category. These 37 decision categories should then of course be narrowed down to one. This could be done by chosing the most popular category or otherwise. This representation is most certainly lighter than the current multidimensional one, but also obviously not able to express as diverse and powerful strategies. If the results of using such a representation were good, maybe the game of Ligeud does not require the complex strategies that we suspected.

Also, the bidding strategies never evolved a really good individual. An entirely different approach such as neural nets could perhaps give a better result.

1 Dimensions of the state space

1.1 Dimensions in the playing phase

Dimension number	Dimension name	Dimension range
1	The number of cards left on my hand	0-12
2	The number of clubs left on my hand	0-12
3	The number of hearts left on my hand	0-12
4	The number of spades left on my hand	0-12
5	The number of diamonds left on my hand	0-12
6	Face value of my overall best card	0-50 (50 is joker)
7	Face value of my best clubs card	0-14
8	Face value of my best hearts card	0-14
9	Face value of my best spades card	0-14
10	Face value of my best diamonds card	0-14
11	Average face value of all my cards	0-21
12	Average face value of clubs	0-14
13	Average face value of hearts	0-14
14	Average face value of spades	0-14
15	Average face value of diamonds	0-14
16	Number of cards played in this round	0-3
17	Bid winner	myself-right player
18	Trump	none-clubs
19	My partner	myself-right player
20	Left player is surely renonce in clubs	no-yes
21	Left player is surely renonce in hearts	no-yes
22	Left player is surely renonce in spades	no-yes
23	Left player is surely renonce in diamonds	no-yes
24	The player across is surely renonce in clubs	no-yes
25	The player across is surely renonce in hearts	no-yes
26	The player across is surely renonce in spades	no-yes
27	The player across is surely renonce in diamonds	no-yes
28	Right player is surely renonce in clubs	no-yes
29	Right player is surely renonce in hearts	no-yes
30	Right player is surely renonce in spades	no-yes
31	Right player is surely renonce in diamonds	no-yes
32	The suit of the left card on the table	none-clubs
33	The face value of the left card on the table	none-clubs
34	The suit of the card across on the table	none-clubs
35	The face value of the card across on the table	none-clubs
36	The suit of the right card on the table	none-clubs
37	The face value of the right card on the table	none-clubs

Table 1: Dimensions of the play phase.

1.2 Dimensions in the bidding phase

Dimension number	Dimension name	Dimension range
1	My number of clubs	0-12
2	My number of hearts	0-12
3	My number of spades	0-12
4	My number of diamonds	0-12
5	Face value of my overall best card	0-50 (50 is joker)
6	Face value of my best clubs card	0-14
7	Face value of my best hearts card	0-14
8	Face value of my best spades card	0-14
9	Face value of my best diamonds card	0-14
10	Average face value of all my cards	0-21
11	Average face value of clubs	0-14
12	Average face value of hearts	0-14
13	Average face value of spades	0-14
14	Average face value of diamonds	0-14
15	Last bid of the left player	nothing-ligeud (ligeud is 12 tricks)
16	Last bid of the player across	nothing-ligeud
17	Last bid of the right player	nothing-ligeud
18	My last bid	nothing-ligeud

Table 2: Dimensions of the bid phase.

2 Experimental results

2.1 Greedy learning

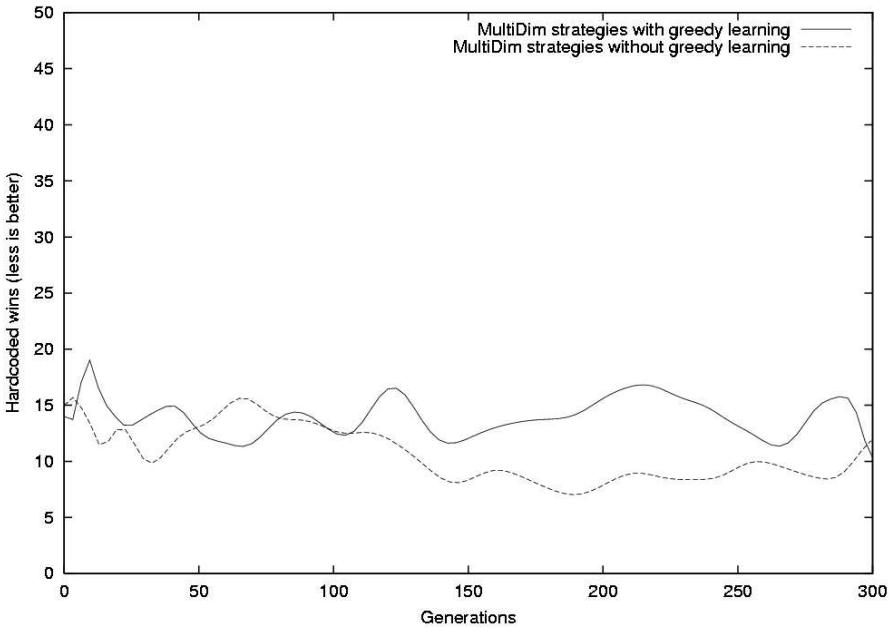


Figure 2: Evolving Multidims with and without learning

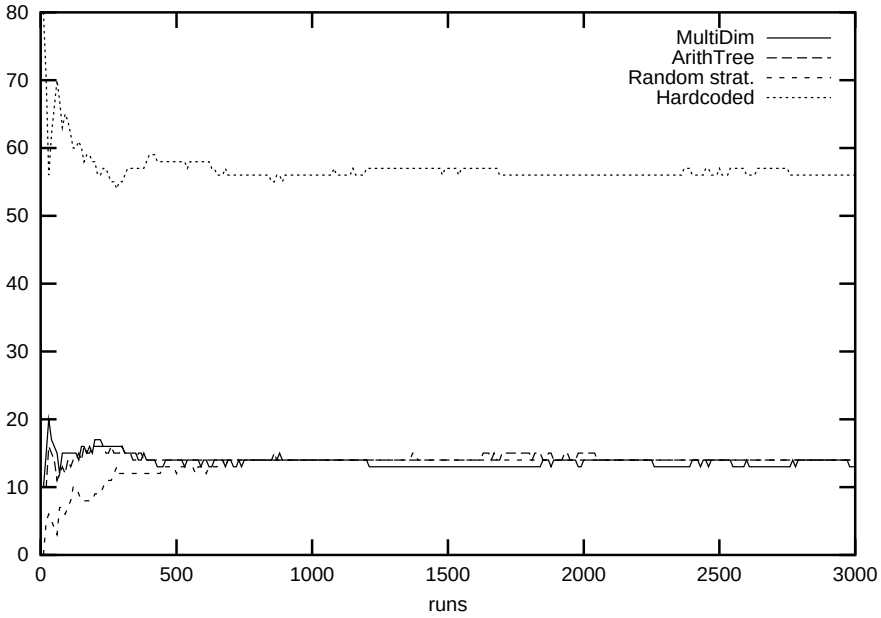


Figure 3: Percentage of total wins for learning multidim strategies against hardcoded and random strategies

2.2 Playing strategies

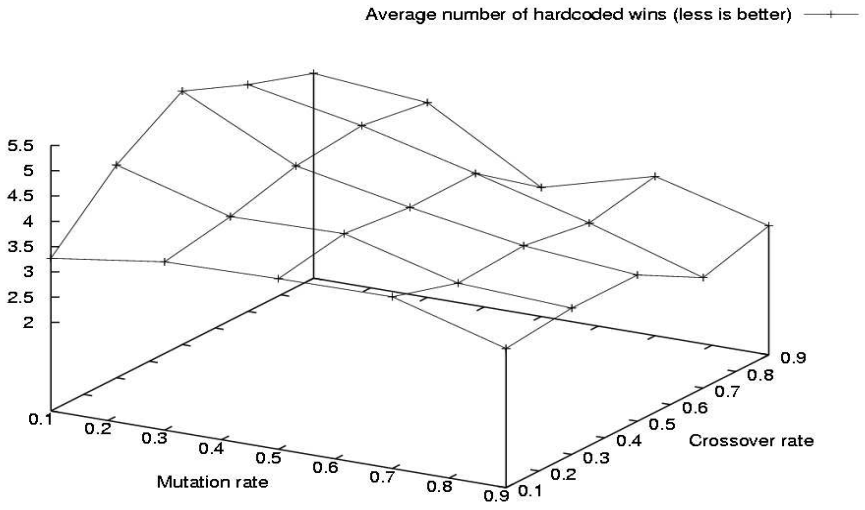


Figure 4: MultiDim.: Systematic tuning of the EA parameters for playing.

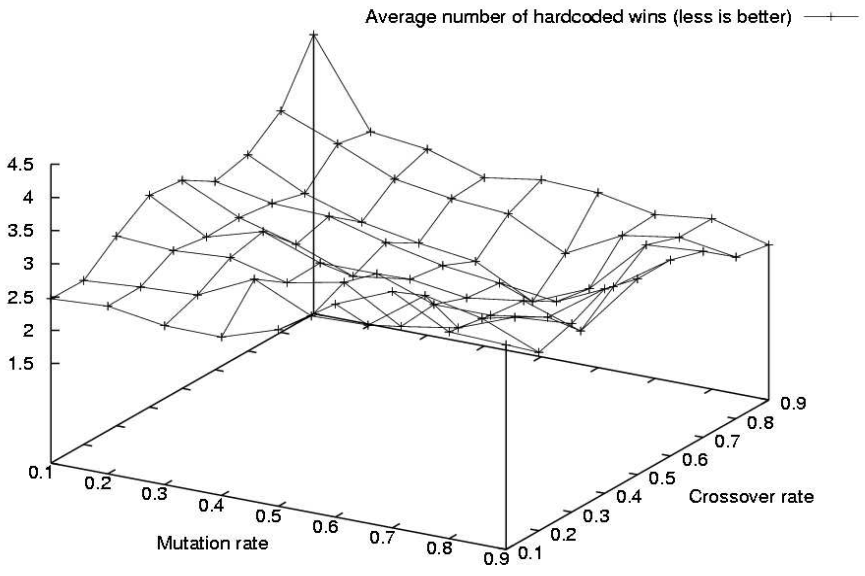


Figure 5: Arith.Tree: Systematic tuning of the EA parameters for playing.

2.3 Bidding Strategies

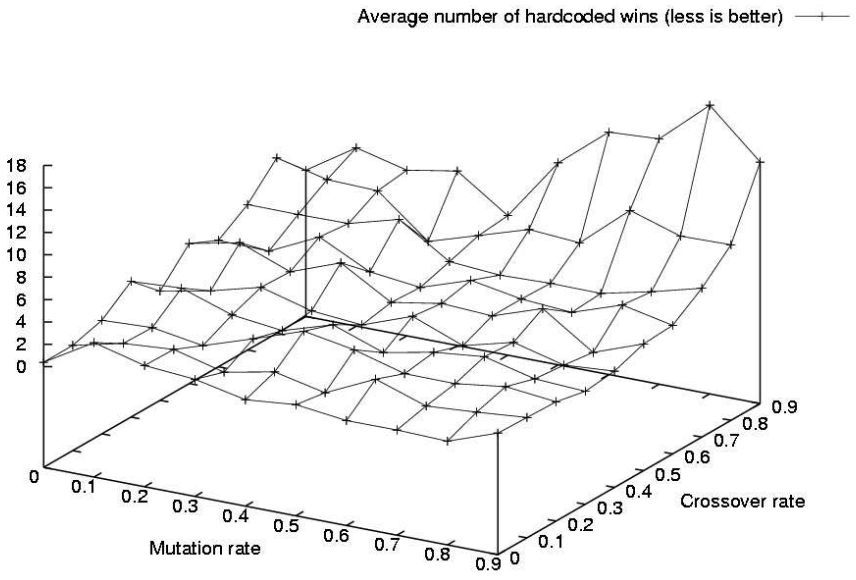


Figure 6: MultiDim.: Systematic tuning of the EA parameters for bidding.

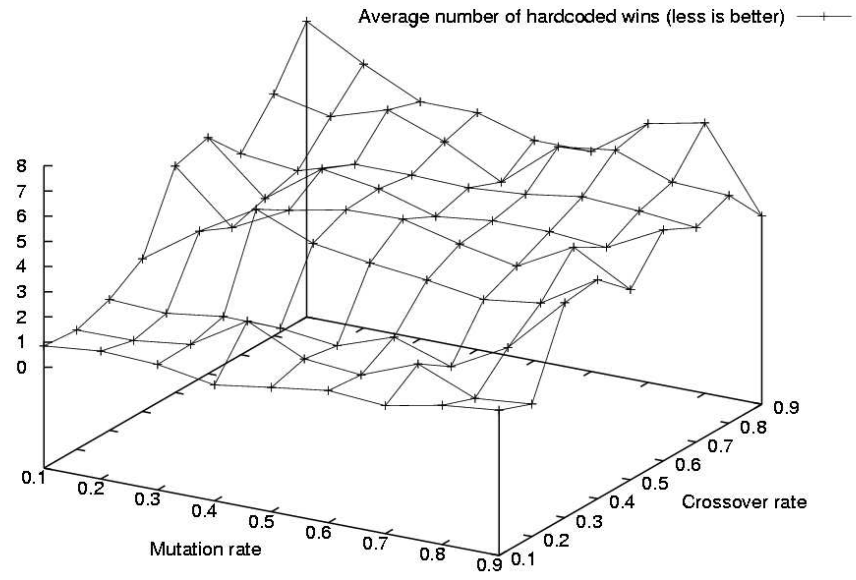


Figure 7: Arith.Tree: Systematic tuning of the EA parameters for bidding.

2.4 Evolving play and bid strategies

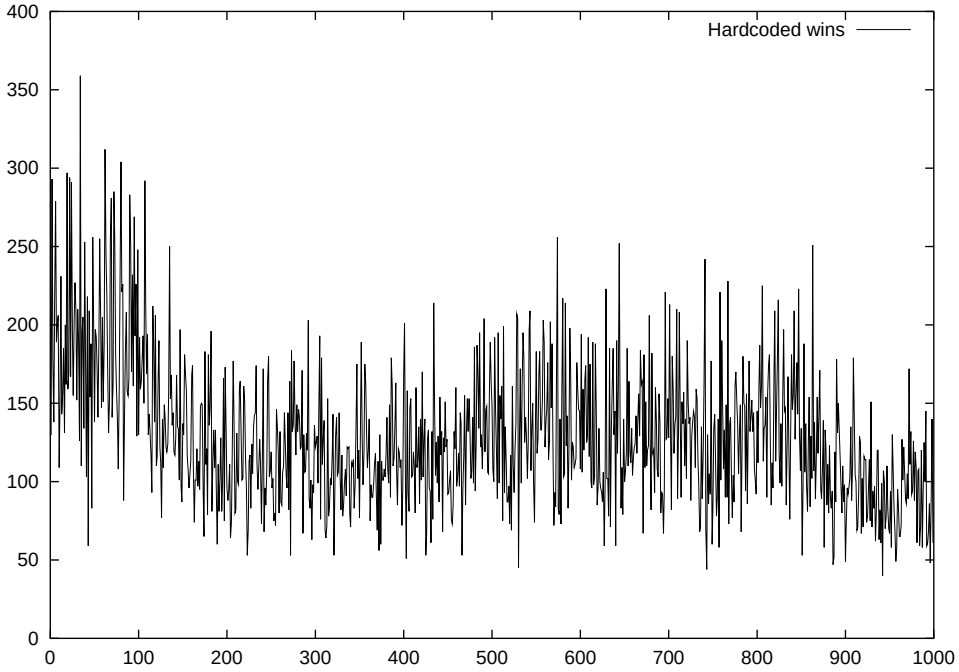


Figure 8: Hardcoded wins against evolving multidim strategies

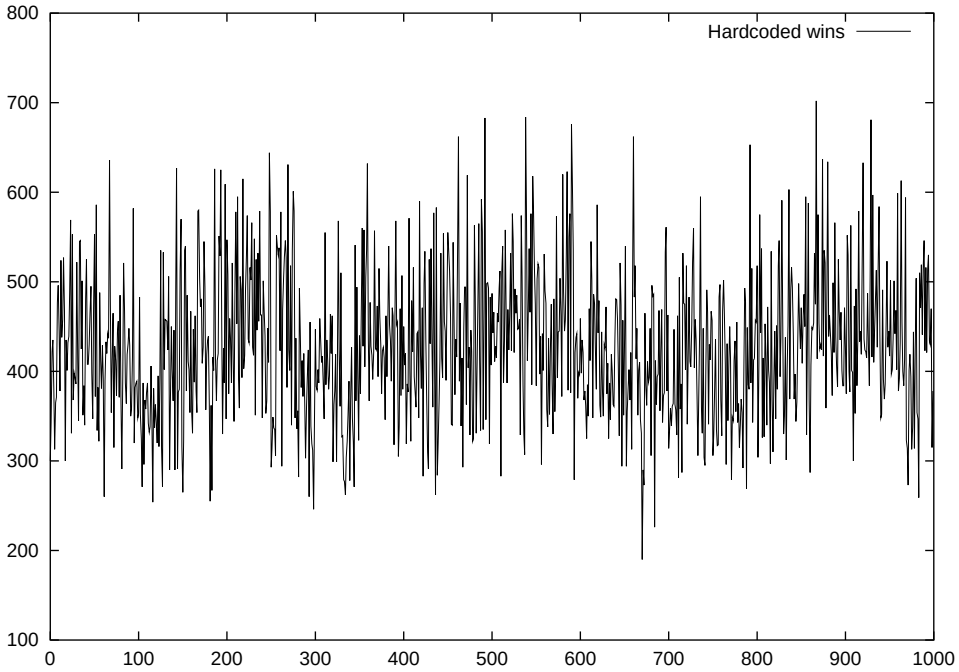


Figure 9: Hardcoded wins against evolving ArithTree strategies

2.5 Comparing performance

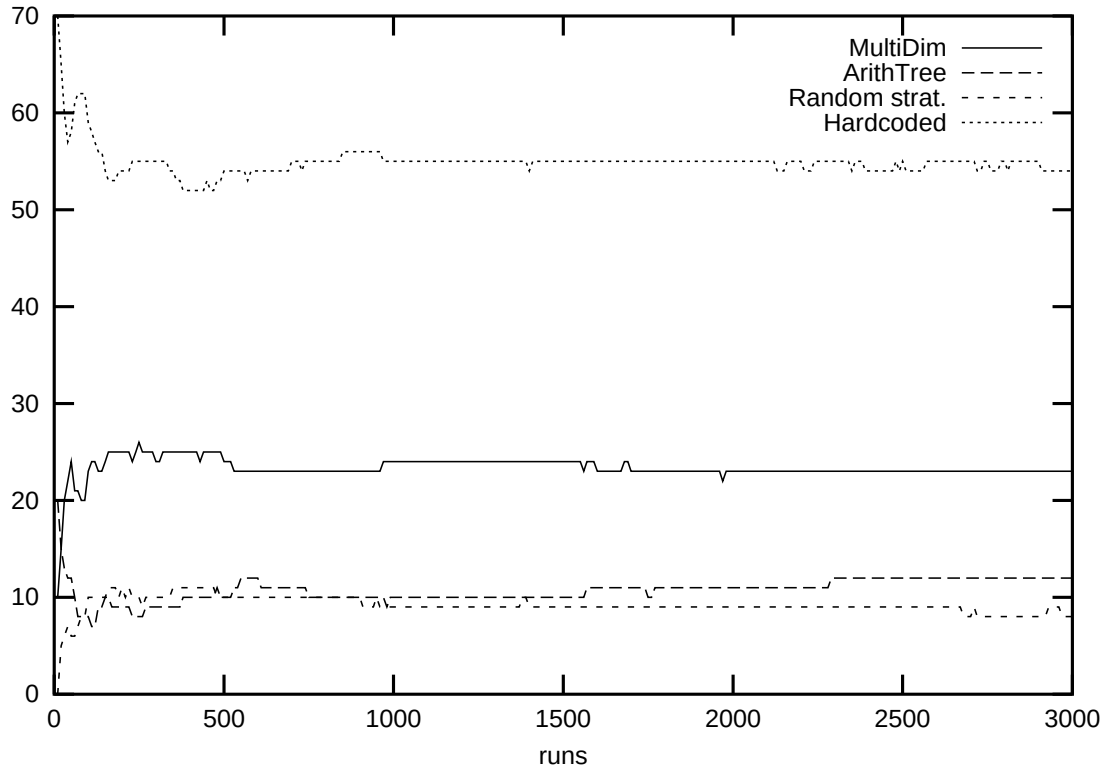


Figure 10: Percentage of total wins for 3000 tournaments, each consisting of 16 games

BREC Project 3

Martin I. Jensen and Thomas Rasmussen

Abstract— **This paper is part of a BREC-project [3] for which the objective is to investigate basic issues in evolutionary algorithms [4, 5]. The general description of an EA leaves a lot of room for creating offspring in many different ways. The goal of this paper is to look closer at the way offspring are created and to see if there are one or more methods or guideline that may outperform other methods or guideline. We will look at four kinds of methods to create offspring.**

1 Introduction

The way in which offspring are created [4, 5] may have a profound impact on the performance of the EA, where performance means quality of the results of the algorithm. We will try out four different approaches to offspring creation and systematically test these approaches in an EA. The first two variants are very similar. The first one picks two parents at random and makes one child, using crossover [4, 5]. The second picks a parent at random for each parent in the population and makes one child using crossover. Both of these algorithms then mutates and make a selection on the population [4, 5]. The third variant does the same as the second, except that is also mutates the child. It then makes a selection on the population. The fourth variant is different from the other three. Offspring in this variant replaces the parent if it has better fitness. This is done for both mutation and crossover, and no selection is done. Using the BREC-Framework [3] we implemented, ran and tested the above mentioned variants of offspring creation.

2 Variants

The following is a thorough description of the four variants of offspring creation that we experimented with in this project. The four variants of offspring creation all used an arithmetic crossover operator as described in the literature [4, 5].

2.1 Variant 1

This algorithm runs through all the parents and makes a probabilistic test on the crossover parameter. If the test succeeds the crossover is done and during this crossover operation, the algorithm picks two parents at random from the whole population and then makes a single child by crossing the two parents and puts this child in the new population. If the probabilistic test fails the algorithm simply copies the current individual to the new population. All the children go through mutation and selection. Normal distributed mutation to the whole genome is done, if a probabilistic test on the mutation parameter succeeds for each individual. After the mutation phase, the new population goes through a standard tournament selection. The pseudo code for the algorithm follows:

```
Initialize and evaluate population
while( not done ) {
 Find elite individual

 newpop = population
 for (i=0; i<popsize; i++)
  if ( random() < crossover) {
   Pick two random individuals I and J.
   newpop[i] = Crossover( I, J )
  }
  else {
   newpop[i] = population[i]
  }
 }
 population = newpop;

 Mutations is done on
 the population.

 Select next generation
 with tournament selection.

 Insert elite individual if
 it was lost in selection.
}
```

2.2 Variant 2

This algorithm runs through all the parents and makes a probabilistic test on the crossover parameter. If the test succeeds a crossover operation is done. During crossover the algorithm picks one parent at random from the whole parent population and then makes a single child by a crossover of the current parent and the randomly picked parent. And puts this newly created child in the new population. If the probabilistic test fails the algorithm simply copies the current individual to the new population. All the children go through mutation and selection. Normal distributed mutation to the whole genome is done, if a probabilistic test on the mutation parameter succeeds for each individual. After the mutation phase, the new population goes through a standard tournament selection. The pseudo code for the algorithm follows:

```
Initialize and evaluate population
while( not done ) {
 Find elite individual

 newpop = population
 for (i=0; i<popsize; i++)
  if ( random() < crossover_rate ) {
   Pick two random individuals I
   newpop[i] =
      Crossover( I, population[i] )
  }
  else {
   newpop[i] = population[i]
  }
 }
 population = newpop;

 Mutations is done on
 the population.

 Select next generation
 with tournament selection.

 Insert elite individual if
 it was lost in selection.
}
```

2.3 Variant 3

This algorithm runs through all the parents and makes a probabilistic test on the crossover parameter. If the test succeeds a crossover and mutation operation is done. The algorithm makes a crossover the same way as Variant 2. The new child is mutated with normal distributed mutation on the genome and put in the new population. That is no probabilistic test is done on the mutation parameter. If the probabilistic test on the crossover parameter fails no crossover or mutation is done and the current parent is copied to the new population. The new population then goes through a standard tournament selection. The pseudo code for the algorithm follows:

```
Initialize and evaluate population
while( not done ) {
 Find elite individual

 newpop = population
 for (i=0; i<popsize; i++) {
  if ( random()< crossover_rate ) {
   Pick one random individual I.
   newpop[i] =
      Crossover( I, population[i] )
   Mutate newpop[i]
  }
  else {
   newpop[i] = population[i]
  }
 }
 population = newpop;

 Select next generation
 with tournament selection.

 Insert elite individual if
 it was lost in selection.
}
```

2.4 Variant 4

This algorithm runs through all the parents while doing the following. A copy of the current parent is made and this new individual is mutated on its genome with a normal distributed mutation operator. If the new individual has better fitness than the current parent, then the current parent is replaced. Af-

ter the mutation face the algorithm then runs through the new population and picks two parents at random. From the two parents it makes a new individual by a crossover of the two random picked. If the newly created individual is better than the worst of the two current individuals, the worst is replaced. No selection is done. The pseudo code for the algorithm follows:

```
Initialize and evaluate population
while( not done ) {
 Find elite individual

 newpop = population
 for (i=0; i<popsize; i++) {
  indv = Mutate newpop[i]

  if indv better than newpop[i]
    newpop[i] = indv

 }

 for (i=0; i<popsize; i++) {
  Pick two random numbers i1, i2
  indv = Crossover( newpop[i1],
                    newpop[i2] )

  idnv replaces the worst parent
  if it is better
 }

 Insert elite individual if
 it was lost in selection.
}
```

3 Experiments

3.1 Settings and output

We used the following parameters for all the variants:

- population size = 100
- crossover operator : Arithmetic crossover, where the child is created in the convex hull defined by the parents.
- crossover probability = 0.9
- mutation operator : Normal distributed mutation, with variable variance 1/(1+t). (t is the generation number)
- mutation probability = 0.75
- genome representation : floating point vectors
- elitism was used

3.2 Test functions

We used the follow seven test functions:
Ackley F1 20D (min):

$$f(\mathbf{x}) = 20 + e - 20e^{-0.2\cdot\sqrt{\frac{1}{20}\sum_{i=1}^{20} x_i^2}} - e^{\frac{1}{20}\sum_{i=1}^{20} cos(2\pi\cdot x_i)}$$

De Jong F4 (min):

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

Griewank F1 20D (min):

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

Rastrigin F1 20D (min):

$$f(\mathbf{x}) = 100 + \sum_{i=1}^{20} x_i^2 - 10\cdot cos(2\pi\cdot x_i)$$

Rosenbrock F1 20D

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

Schaffer F6 (min):

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

Ursem multimodal F8 20D (max):

$$f(\mathbf{x}) = 2\cdot cos(2\pi\cdot\prod_{i=1}^{20} x_i) - 4\cdot(\sum_{i=1}^{20}(x_i+1)^2) + (2/n)\cdot\sum_{i=1}^{20}(cos(2\pi\cdot x_i))$$

All but Schaffer, which were run with 50000 evaluations, the rest were run with 200000 evaluations. All variants were run 50 times and data was collected during each run with the BREC-Framework.

3.3 Results

Results obtained from experiments. Average best fitness of 50 runs at the end of each the runs. SD = Standard diviation.

	Avg ± SD
Variant 1	4.3050±1.4441
Variant 2	3.7967±1.3308
Variant 3	4.0085±1.1213
Variant 4	3.1593±1.0897

Table 1: Ackley F1 20D

	Avg ± SD
Variant 1	7.2515E-09±7.0844E-17
Variant 2	8.1832E-16±3.6358E-17
Variant 3	1.0971E-09±1.0692E-17
Variant 4	3.4769E-13±1.6375E-14

Table 2: De Jong F4

	Avg ± SD
Variant 1	0.0633±0.1412
Variant 2	0.0470±0.1681
Variant 3	0.0613±0.2113
Variant 4	0.0177±0.0213

Table 3: Griewank F1 20D

	Avg ± SD
Variant 1	8.4572±2.7433
Variant 2	8.4174±3.0959
Variant 3	7.8005±2.6004
Variant 4	8.4971±3.7348

Table 4: Rastrigin F1 20D

	Avg ± SD
Variant 1	45.2995±80.0230
Variant 2	34.1214±51.8390
Variant 3	112.5946±319.4000
Variant 4	27.1350±27.6800

Table 5: Rosenbrock F1 20D

	Avg ± SD
Variant 1	1.4192E-02±1.3032E-02
Variant 2	1.3804E-02±1.3307E-02
Variant 3	1.5507E-02±2.2212E-02
Variant 4	6.0858E-09±4.4233E-09

Table 6: Schaffer F6

	Avg ± SD
Variant 1	2.2620±2.2421E-03
Variant 2	2.2623±6.4699E-08
Variant 3	2.2608±1.1017E-02
Variant 4	2.2623±7.3365E-07

Table 7: Ursem Multimodal 8 20D

3.4 Graphs

Recording the average best fitness, where the number of evaluations mod 1000 equals 0, made the first set of graphs. The reason for this was the noisiness of the graphs when using all evaluations. The graphs can be found in figure 1-7.

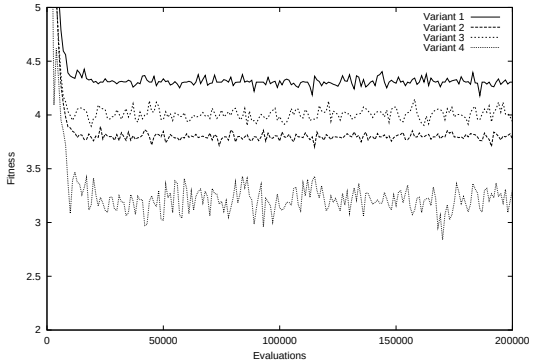


Figure 1: Ackley F1 20D

For the second set of graphs the average best fitness were recorded at every generation. The generation number was then multiplied by the population size (100). This multiplication makes these graphs similar to the other graphs. The reason for making these graphs was that they were less noisy than the others. The graphs can be found in figure 8-14.

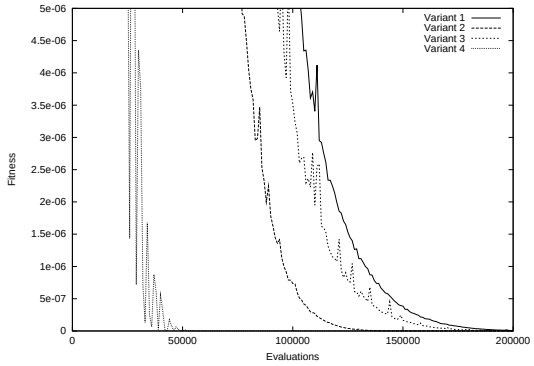


Figure 2: De Jong F4

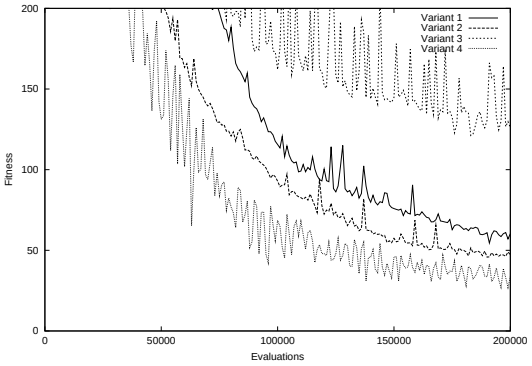


Figure 5: Rosenbrock F1 20D

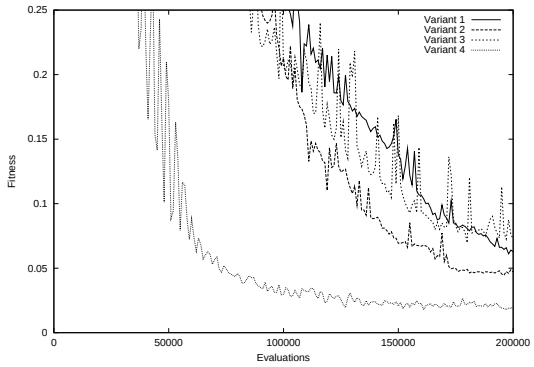


Figure 3: Griewank F1 20D

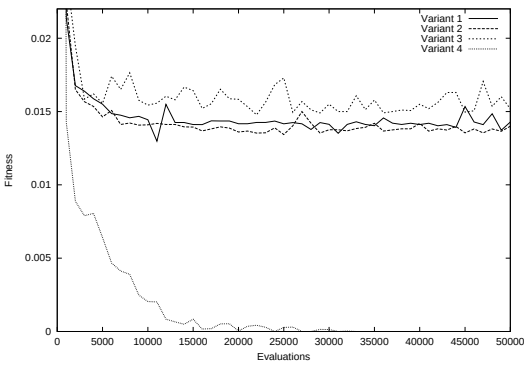


Figure 6: Schaffer F6

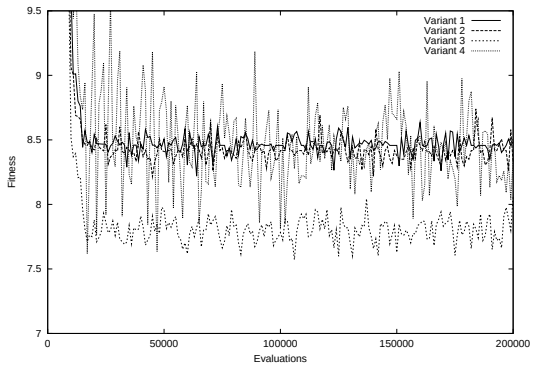


Figure 4: Rastrigin F1 20D

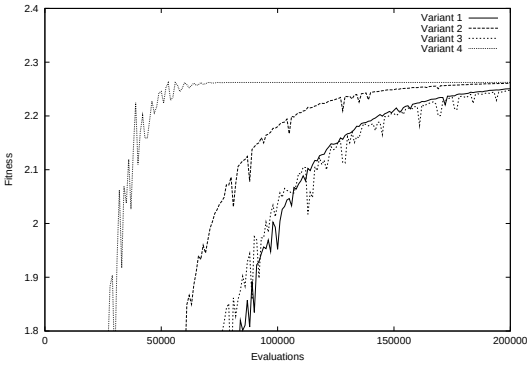


Figure 7: Ursem Multimodal 8 20D

3.5 Conclusion

It is clear to see that Variant 4 is by far better than the other three variants. It finds the best optimum value in four out of the seven tests. And the second best optimum value two times. Most of the time, Variant 4 converge to a near optimum solution faster than the other and the solution in most of the cases is also better than the others. Variant 2 comes out of the tests with nice results as well. It has better results than Variant 1 and 3, but not as good as Variant 4.

4 Discussion

The results of our experiments with different variants show that Variant 4 can make better results when it comes to real numerical optimization problems. The reason for this may be that the only offsprings that survives are those with better fitness. This is not the case for the other three variants, where offspring with

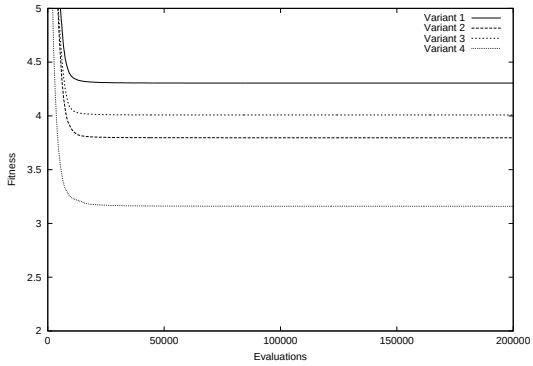


Figure 8: Ackley F1 20D

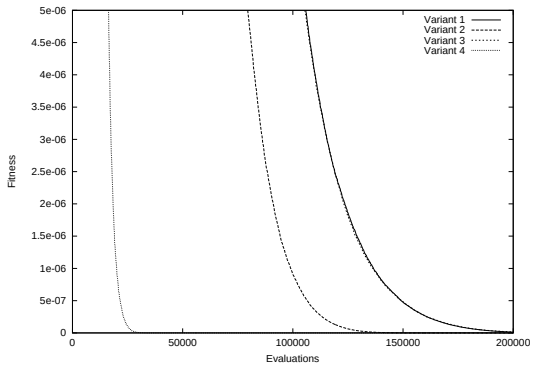


Figure 9: De Jong F4

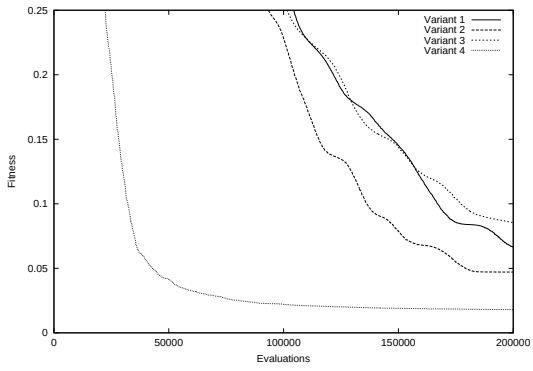


Figure 10: Griewank F1 20D

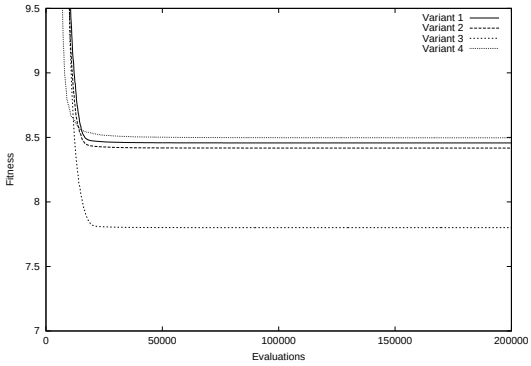


Figure 11: Rastrigin F1 20D

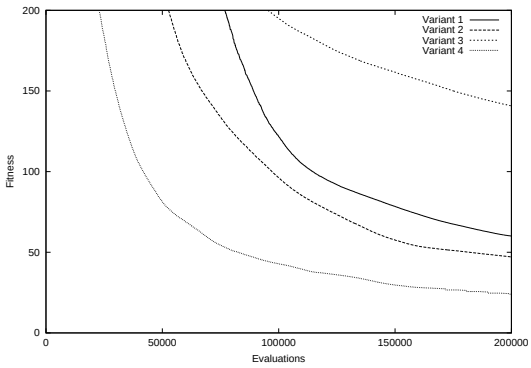


Figure 12: Rosenbrock F1 20D

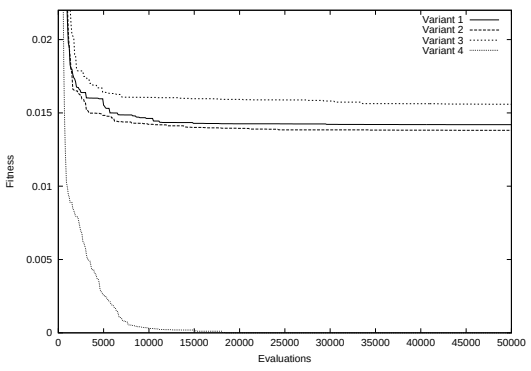


Figure 13: Schaffer F6

worse fitness can survive several generations. Another reason for Variant 4's better performance, may be that there is no selection done on the population level. The selection is done if a mutation or a crossover is better than the parent. This means that in theory it is possible for the population to contain several near optimal solutions, but with genomes far apart from each other, in the genome-space. This is some form of island-based algorithm [5], where each individual constitutes an island, and migrations are done by the mutation- and

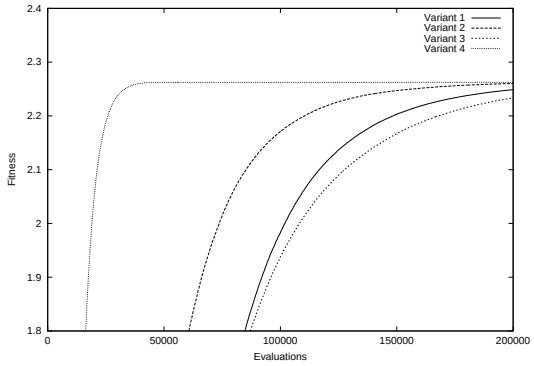


Figure 14: Ursem Multimodal 8 20D

crossover-operators.

References

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

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

[3] Rasmus K. Ursem. http://www.evalife.dk/toec2001/brecproject.php.

Optimizing Bus Schedules for Aarhus Sporveje

Sabrina Nielsen and Thomas Lindgaard

Abstract— **This report describes our attemps to minimize waiting times for the busses run by Aarhus Sporveje through use of an EA. It contains our model, the ideas used in the EA and our experiments. It also discusses problems and considerations encountered during the project.**

1 Introduction

The purpose of our project has been to minimize the time you need to wait when changing from one of the busses run by Aarhus Sporveje to another, In other words we wanted to minimize the waiting time just by shifting the scheduled departure times of the routes while keeping the current routes and the current interval between departures and thereby making the schedules for the different routes "fit together more nicely".

We had two main objectives in mind:

- **To apply EAs to a real world problem**
  We thought it would be a reasonable challenge to model a problem from the real world and examine which problems such a model would imply and whether we could construct an EA that could face up to those problems. The reason why we chose to experiment with minimizing waiting times was that it is an intance of a very common problem, namely scheduling.
- **To experiment with fitness vectors versus fitness values**
  Similarly, we felt it could be an interesting strategy to let the individual genes in the genome have their own fitness such that an individual would be equipped with a fitness vector rather than a single fitness value. The idea was then to make the crossover dependent on the genes individually. The risk in this approach is the possibility of two genes counteracting each other.

1.1 Outline of the idea

The idea for the project was first of all to make a reasonable model of the bus routes and their stops. Secondly, we wanted to create an EA that used individuals consisting of a genome with a gene for each route and contained both a corresponding fitness vector and a scalar fitness value. Lastly, there remained the task of acquirering the real world data from the bus schedule.

Our experiments were mainly aimed at examining two things:

1. Whether or not the EA produced better results when using the fitness vector rather than the fitness value – would it benefit (significantly) from the extra knowledge.
2. Whether EAs are at all suitable for solving the problem - when using our model, at least.

2 Model

Obviously the real world is a rather complex thing to deal with. Hence, we have found it necessary to work with a simplified model of it. Our model only contains a subset of the actual bus routes and stops due to the restrictions described below.

For bus routes it is assumed that they always travel the same path and that they travel back and forth between two stops (rather than run around in circles). In the real world it is not completely uncommon that the busses run by Aarhus Sporveje have some subset of the path divided into two so that only every other bus passes that subset. There is also a single circular route. In our data collection we have pretended that busses always travel the same path and we have ignored the one that travels in circles.

2.1 Routes

A bus route is equipped with an id, which is the number that route is identified by in the real world. It also holds an interval telling how often that route runs (in minutes between departures). The interval is static which excludes the possibility of modelling the fact that most busses in the real world run more rarely in the evenings and on weekends. Finally it contains a list of stops on that particular route and how long it takes to travel from the first (or last) stop on the route to any other stop on the route – this information is of course necessary to accurately decide where on the route the bus is at a given point in time.

It is assumed that the bus passes a given stop at the same time each hour. This implies that the interval can be no more than 60 minutes, but when working with Aarhus Sporveje this restriction is not particularly severe since only one route exceeds this limit. We also keep no information as to when the first and last busses are scheduled – in our model the busses run day and night.

2.2 Stops

The bus stops are somewhat simpler in that they are identified by a unique id, a name and information about the bus routes that pass them.

A stop may represent more than one stop from the real world since it represents "both sides of the road" – the stops for each direction are only represented as one stop in our model. If there is a road cross the four directions are incorporated into one stop. This of course makes it possible to model the fact that passengers often have to cross the road between busses rather than just switching to another bus travelling in the same direction but following a different route up ahead.

2.3 Representation of time

As described previously all representatinos of time in our model are done in minutes. We have had several descussions whether or not this choice resulted in too discrete a spectrum and whether or not better schedules could be achieved using a finer resolution. We have concluded, however, that such a schedule would not be applicable in the real world anyway. Thus we have not persued this any further.

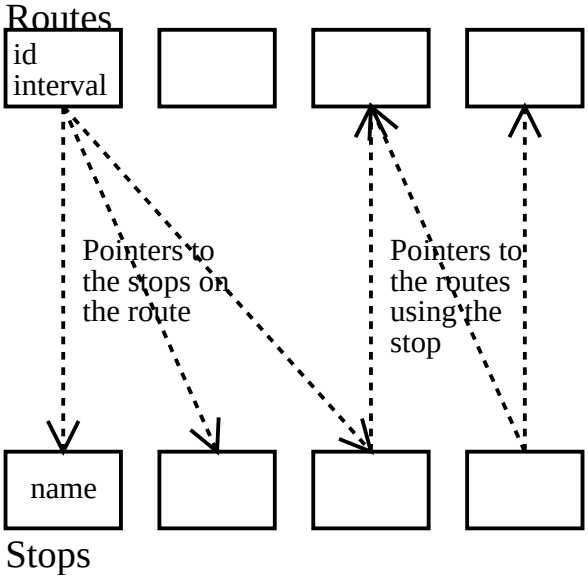


Figure 1: The relations between stops and routes

3 The EA

Our EA is implemented using the template on page 13 in [4].

The individuals consist of a genome containing a gene for each route and a starting time for each of the route's endpoints. If for instance the route has an interval of 20 minutes, the starting time for the first endpoint will be a number between 0 and 20, while the starting time for the other end will be set to the first starting time plus the travel time plus 0 to 10 minutes. This interval is added to make room for more flexible schedules. The two starting times are all that the algorithm can move around. The individual also has a fitness vector with an entry for each gene and a scalar fitness value.

3.1 Fitness calculation

The most complicated aspect of our algorithm is the evaluation of fitness. Here we had to find a way to translate into a single number the total amount of time that all the busses

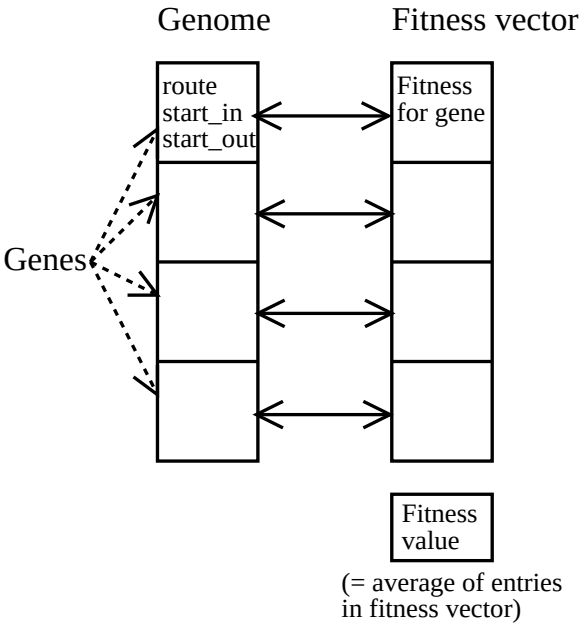


Figure 2: Schematic overview of an individual

would have to wait for all the other busses at all the stops going in two different directions.

We also had to decide exactly how we wanted the optimization to be: Should the average waiting time be as small as possible or should there be a stop somewhere on the route where the waiting time for some bus was as small as possible. What we chose was a combination of the two: We felt that it was reasonable to expect that people planned their travels a bit themselves. The optimal situation would therefore be, that at a given stop – no matter what route the passenger used to get there and no matter what route (s)he wanted to switch to – there would be an arrival of the first bus where the waiting time for the departure of the second bus was minimal.

Hence, for each gene (ie. each route) we run through the stops it passes and for each of these search for the departure the passenger should choose, if (s)he planned to switch to another in the set of busses using that stop. In other words, if the passenger for instance wanted to take route 1 (running 6 times an hour) to stop A and here switch to route 2 (running only once an hour), it would be silly not to get on the bus among the six running every hour that would arrive just before route 2 visits stop A. Having calculated all the minimal waiting times we take the average and use this as the fitness for that particular gene.

It should be noted that we find it optimal to arrive two minutes before the desired route departs in order to have time to cross the road or to leave room for heavy traffic. This decision has no greater influence on our calculations in that we simply pretend that we arrived two minutes later (which means that if we arrived just one minute before the departure of the other route we were too late in the eyes of the EA).

3.2 Selection

The selection of the next generation is done by tournament selection. Each individual is compared to one other and the fittest one survives (hence the fittest individual will always survive).

Since one of the objectives in our project has been to experiment with fitness vectors versus fitness values, we have implemented two ways of comparing fitness of individuals: Comparison based directly on the vectors and comparison based on the average of the sum of the vectors – a scalar value. Later it has turned out that our way of comparing vectors does not give a total order and we have had to rely solely on the scalar values (see 6). This does not destroy our experiments, though – we can still use the vectors for crossover.

3.3 Mating

After half the population has been discarded by the selection, the surviving individuals breed to get the population up to full size. All individuals parent at least one new individual. The new individual is created in one of two ways depending on whether we use fitness vectors or values:

- **Fitness vectors**
  The starting times in each gene is created from the parents' genes by means of a weighted average where the fittest gene is weighted by the crossover rate and the other by (1 - crossover rate).
- **Fitness values**
  If we use fitness values the genes from the

fittest parent is weighted by the crossover rate.

3.4 Mutation

The way we decided to do the mutation is to mutate the individual genes by adding a number between -1 and 1 to the starting times with a probability defined by the mutation rate. Since we only work with a smallest time unit of one minute, the mutation can only move the starting time of a route by minus one, zero or one minute at a time. This seems a good strategy for routes running several times an hour, but it may not work as well for the routes running only once every hour – a minute may not matter as much for second type of routes as the first (see 7).

4 The experimental setup

As described previously, an essential part of our project has been to experiment with fitness vectors versus fitness values. Thus our experiments have been performed twice: One setup where crossover was made on the genes individually and once where the crossover depended solely on the overall fitness of the mating individuals.

For the evaluation of a run of the EA we used the fitness value for both kinds of experiments, since it must be expected that a good run had a low fitness value even though the crossover was based on the fitness vector with the fitness value being the average of the entries in the vector. Also this simplifies the job of comparing the two kinds of runs.

First we examined how many generations were nessecary for the population to converge and how large the population should be. It turned out that after 25 generations practically all individuals had the same starting times even with quite large populations. There seemed not to be much difference in the final results whether we had 25 or 250 individuals in the population, so we decided to run our experiments with 25 individuals for 25 generations.

The actual experiments were run with crossover rates ranging from 0.2 to 0.8 with an interval of 0.1 and with mutation rates ranging from 0.0005 to 0.0025 with an interval of 0.0005. Each combination was run 10 times and an average calculated.

5 Results

The results we have obtained from this project are not impressive. The two sets of experiments are shown in tables 1 and 2.

5.1 Table 1: Fitness vectors

A bit to our surprise the results seem to be completely independent of both crossover rate and mutation rate – all runs simply seem to give an average best waiting time of a little less than five minutes. Our first suspicion was that this was perhaps because the individual genes counteracted each other and forced the individuals to stay "bad" since there could be a dependency between two genes such that if one of them was good the other would automatically be bad. This does not seem to be the case, though, since we get more or less exactly the same results when using the fitness value.

5.2 Table 2: Fitness values

After running this set of experiments we were even more surprised. Not only are the results completely independent of mutation rate and crossover rate, there also seems to be no difference from the results produced by the previous experiment.

Overall there really is no lesson to be learned from the experiments regarding population size, mutation and crossover rates or the use of vectors or scalar values.

6 Problems defining ordering on fitness vectors

During this project we have encountered various problems.

The most serious was to find a suitable comparison on fitness vectors that reflected that we actually worked on vectors rather than scalars. Our first choice was to say that the fittest vector was the one with the greater number of fit/smaller entries. It turned out however that this ordering was not transitive. This caused

severe problems for the algorithm since no individual could be said to be the fittest.

Anti-transitivity can illustrated by the following example:

$$\begin{array}{lcl} A & = & \{\ 2\frac{1}{2} \quad 1 \quad 2 \quad 1 \quad 1 \quad \} \\ B & = & \{\ 1\frac{1}{2} \quad 2\frac{1}{2} \quad 1\frac{1}{2} \quad 1 \quad 1 \quad \} \\ C & = & \{\ 1 \quad 2 \quad 1 \quad 2 \quad 1\frac{1}{2} \quad \} \end{array}$$

We see that using the proposed ordering we get the following chain $A > B > C > A$, by observing that A has one entry which is smaller than the corresponding entry in B whereas B has two entries which are smaller than the corresponding entries in A. Similarly for the other combinations B-C and C-A. Obviously this chain constitutes a cycle and thus the ordering can not be transitive.

We found no reasonable solution except to simply compare the fitness values that obviously reflect the quality of an individual, but this solution does not really express the fact that we deal with vectors rather than scalars.

7 Extensions

We have thought of a couple of extensions that could me made to our project.

- Defining a way of weighting the stops such that stops that are passed by many busses or placed in central locations (passenger intense locations) are weighted higher than less frequented or more remote stops. This could then be used to reward individuals that have low waiting times at these stops which would be a good thing since these stops probably host many bus changes.

- Another way of improving the algorithm with respect to getting results usable in the real world is to supply the stops with "preferred arrival times". This should be used to make sure that for instance stops located next to schools are visited just before eight o'clock rather than shortly after.

- Furthermore several aspects of our model could be extended to more realisticly mirror the real world. For instance an entire day or maybe even week could be modelled rather than just one hour to achieve more flexibility in the schedule. Similarly a route that in reality has two paths could be split into two routes with each their path, but with the restriction that they should have a specific interval between them at stops visited by both of them.

- A thing that could be done to improve the actual algorithm would be to mutate within a certain fraction of the interval size rather than within static values [-1:1].

8 Conclusion

We are slightly annoyed, to say the least, that our experiments have supplied us with no information whatsoever to conclude on our idea about supplying the individuals with a more detailed fitness than just a scalar value. A tempting conclusion would be that it makes no difference and is thus superfluos. The lack of change across mutation rates and crossover rates, however, leads us to believe that our model and the relations between routes and stops are so complex that not even a vector is sufficient to represent the quality of an individual – perhaps even more dimensions are needed... or perhaps we have a hidden bug somewhere.

We do not feel that the project has supplied us with any information that justify using EA's for the problem adressed. We are not discouraged, though. We still believe there is a possiblility that an EA could find an optimal bus schedule if given enough information.

9 Source and executable

The source code and an executable can be found in /users/u972035/job/evalife/Sporveje.zip.

The program is developed on a Windows XP platform using Borland C++ Builder.

10 Contact

Sabrina Vestergaard Nielsen,
u972247@daimi.au.dk

Thomas Lindgaard,
u972035@daimi.au.dk

References

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

	0.2	0.3	0.4	0.5	0.6	0.7	0.8
0.00050	4.6727	4.6806	4.6804	4.6495	4.6592	4.6427	4.6624
0.00100	4.6829	4.6599	4.6931	4.6704	4.6511	4.6494	4.6263
0.00150	4.6698	4.6664	4.6747	4.6614	4.6109	4.6631	4.6411
0.00200	4.6822	4.6761	4.6728	4.6580	4.6401	4.6466	4.6438
0.00250	4.6741	4.6675	4.6451	4.6615	4.6519	4.6535	4.6242

Table 1: Population size: 25, number of generations: 25, fitness vectors

	0.2	0.3	0.4	0.5	0.6	0.7	0.8
0.00050	4.6534	4.6576	4.6479	4.6663	4.6459	4.6807	4.6582
0.00100	4.6848	4.6466	4.6561	4.6563	4.6326	4.6661	4.6418
0.00150	4.6494	4.6681	4.6615	4.6668	4.6282	4.6811	4.6675
0.00200	4.6669	4.6458	4.6616	4.6325	4.6552	4.6777	4.6656
0.00250	4.6610	4.6581	4.6581	4.6550	4.6594	4.6596	4.6504

Table 2: Population size: 25, number of generations: 25, fitness values

Pacman Evolver

Lasse Westh-Nielsen and Per Jefsen

Abstract— **This document describes the design of a genetic programming system, designed to evolve pacman players. The motivation stems from an interest in evolving something "stronger" than just solutions to problems, in this case evolving agent behaviour in simple physical surroundings. From this project we have found this to be possible indeed, since the system is able to evolve quite good pacmen relatively quickly. We also liked the idea of having a graphical simulator, capable of illustrating the behaviour in a manner that is close to the way we as humans would measure the quality of a pacman player.**

1 Introduction

The goal of the project is to use genetic programming techniques to evolve controller programs for pacmen. This is, programs that control the way in which the pacman moves around in, and (hopefully) finishes the playfield. The original idea included ghosts (and coevolution of pacmen and ghosts), but we found it hard enough to evolve pacmen that could just finish the field relatively cleverly. The goal of the pacman is thus to eat all cheeses in a 19x19 playfield[8], using as few moves as possible.

The idea in genetic programming (GP) is to supply the genetic system with programming constructs, from which meaningful programs can arise [1]. Like in an evolutionary algorithm (EA) [4], we have a population of individuals. In EAs, the individuals represent potential solutions to a given problem, whereas GP individuals are *programs*.

An attempt has been made earlier at evolving pacman controllers, by John Koza [1]. He succeeded in evolving controllers that were somewhat good at playing pacman. His system included ghosts. We have chosen a different approach at the evolution, since Kozas system includes terminals[9] like "Run away from nearest ghost" and "Progress towards nearest cheese", and others that make it almost trivial to evolve relatively good pacmen, since a pacman with nothing but the latter of the mentioned constructs is not among the silliest. More in what we believe to be the spirit of evolutionary computation, we will try to evolve the programs, instead of coming up with templates like the above mentioned. After all, evolved solutions often surprise the human supervisor, so it may not pay to feed the system with constructs imagined to be meaningful by the programmer.

[8] See screenshot on page 86

[9] see next section

2 The Strategy Tree

In our project, the candidates are represented by tree structures (hereafter referred to as strategies), that can be evaluated at any given time (the evaluation possibly depends on the state of the playfield and the player). Every individual in the population (every pacman) has a strategy. This strategy is evaluated every time the pacman has a choice to make, which is at every cell in the playfield. The basic movement unit is one cell in the field.

The strategy tree is, as mentioned, evaluated every time the pacman has to decide where to go next. At first, our implementation was a bit naive, as an evaluation of a strategy at a given point always yielded just one direction, namely the direction that the strategy suggested to go in at that time. A strategy was free to choose an illegal direction, as the simulator takes care of this. Two kinds of illegal moves can be made. First, a pacman can for instance be heading along a long straight path to the right, and suddenly choose to move down, in which direction there is a wall. In this case, the simulator ignores the desire, and tells the pacman to continue in the direction is had before.
Second, imagine a strategy that always evaluated to "go right". Soon, that pacman will hit the wall, in which case the pacman is simply stopped. In most real pacman games, it is

possible to stop the pacman by running into a wall, which the evolved pacmen were also allowed to do in our system. A strategy like the one mentioned, that always evaluated to "go right", will of course be unable to complete the field, and will receive a high fitness value (selection of course favoring low fitnesses).

With this initial approach, a rather large strategy tree was needed if a pacman should be able to finish the field in any reasonable number of moves (thus making it very difficult for the system to evolve good strategies). Therefore, we modified the evaluation process to no longer yield a direction, but instead four numbers, each representing the score that the strategy gives a particular direction at the evaluation time.

From these four numbers, the ones representing illegal moves are set to zero, and a small value is added to all the legal ones. Then a selection is made probabilistically between the legal directions, so that the higher the score a direction has, the more likely it is to be chosen. This way, a legal direction is always chosen, and a small amount of nondeterminism is introduced. This proved to be a more realistic choice, making it easier for the system to evolve pacmen capable of actually playing pacman somewhat intelligently (!).

Below the various terminals and nonterminals from which the strategy trees are constructed are briefly described. The terminals are the "information units" that the nonterminals can combine into trees, which can be evaluated at any given time. Some of the terminals inform of the status of the playfield or the pacman when evaluated, while others are just constants. Thus, a tree (or a subtree) with only constant terminals can be simplified. This may not always be an adventage, which is the reason why we have included a *simplify*-mutation operator (so that not all trees are simplified)[10].

[10] normally, "redundant" information such as a subtree with constant terminals can prove to be useful later, i.e. exons in DNA sequences

2.1 Nonterminals

The nonterminals are the tree nodes that have one or more subnodes. A node has one of the following types:

- **Integer**
- **Boolean**
- **Direction**
- **Points**

`Integer` nonterminals

- **Plus**. This node has two subnodes, both of type `integer`. When this node is evaluated, we evaluate both subnodes, sum up the resulting values and store the sum as the value of the **Plus**-node.
- **Mult**. Equal to **Plus**, except that the result is the product instead of the sum.
- **DistanceToCheese**. This node has one subnode of type `direction`. When evaluated, the result is the distance (in cells) to a cheese in the direction specified by the subnode, or a large number if no cheese is found, or a wall is encountered before a cheese.

`Boolean` nonterminals

- **AND**, **OR** and **NOT** are the usual boolean operators with 2, 2 and 1 operands of type `boolean`, respectively.
- **LessThan** is the strict numerical '¡' operator that compares the values of 2 subnodes of type `integer`.
- **Equal** is the comparator, in our case overloaded to handle all applicable types, i.e. `integer`, `boolean` and `direction`.
- **IsWall**. This node operates one a subnode of type `direction`. It returns true in the case where the cell adjacent to pacman in the given direction is a wall.

`Direction` **nonterminals**

- **Reverse** operates on one subnode of type `direction`, and it returns the opposite direction of the one given.

- **If** is a 3 argument conditional operator. It has 1 subnode of type `boolean` and 2 subnodes of type `direction`. If the boolean subnode evaluates to true, **If** will return the value of the first direction subnode; otherwise, it returns the value of the second direction subnode.

`Points` **nonterminals**

The root of any strategy tree has type `points` (since we modified the system to the point-giving approach).

- **AddPoints** is the construct that allows a strategy to specify which direction(s) that are feasible at the time of evaluation. It has two subnodes, one of type `integer` and one of type `direction`. When evaluated, the integer specifies the number of "points" to be added to the specified direction.

- **AND** is used to combine **AddPoints** nodes (and has nothing to do with boolean AND) into sequences. The root of a strategy tree is either a single **AddPoints** node or an **AND** node. Using **AND** nodes, a strategy can add points to more than one direction (which of course is essential to a good strategy).

2.2 Terminals

`Integer` **terminals**

- **Constant** integer value. In the initialization, constant integer values are assignes a random number. The evolutionary process can modify these values through mutation.

`Boolean` **terminals**

- **Constants**. The usual true and false boolean constants.

- **MoreUp**. When this terminal is evaluated, it considers the position of the pacman in the playfield, and returns true if the majority of the remaning cheeses are above the pacman; otherwise false is returned.

- **MoreLeft** is like **MoreUp**, except horizontal.

`Direction` **terminals**

- **Constants** include the four directions relevant for playing pacman, i.e. up, down, left and right.

- **DNC**. This is the **direction with nearest cheese** node, that has proven to be quite popular among evolved strategies. Earlier, we commented on John Kozas use of pre-thought-of "cheat-mode" terminals like "proceed in the direction of the nearest cheese" and so. This terminal is somewhat similar, except that the *action* is not included in the terminal, wherefore it is up to the evolution to add a suitable amount of points to this terminal, if it so chooses.
  If no cheeses are visible (in the sense that a pacman cannot see through walls) from the location of the pacman, we return **nodirection**, which is a dummy direction. This proved to be necessary, since no meaningful `direction`-value can be determined in this case. Another option would be just to return a random direction, but this would always infer some time periods of completely random walking in any strategy who chooses to make use of this terminal. By inspecting some of the strategies generated by the system, we found that many of the better ones made use of this terminal in combination with the `MoreUp` and **MoreLeft** terminals. This makes sense, since the **DNC** terminal is very good at "cleaning up" local areas, while the **MoreUp** and **MoreLeft** terminals guide the pacman in the direction of other unexplored areas when the local neighborhood is emptied.

3 The genetic programming system

The flow of the system is described here. Like in an EA, the idea is to evolve better and better individuals over generations, by means of selection, crossover and mutation carried out on a population initialized (somewhat) randomly.

3.1 Population initialization

We generate a number of pacmen, giving each of them a random strategy tree of a random, but controlled size. The random strategy generator is not counted on to generate meaningful programs, but rather to introduce many (and hopefully all) of the different constructs supported, so that the evolution has a better chance of combining them into meaningful programs. Thus, the random strategy generator is only used in the beginning of the evolutionary process.

3.2 Fitness evaluation

Here, the individuals in the population are tested for their ability to complete the playfield. We have built a simulator, into which a pacman can be inserted. The fitness is defined simply as the number of moves in which the playfield is completed, with an certain upper limit. That is, we do not differentiate between pacmen that can possibly complete the field in more than this maximum of moves, and pacmen that cannot complete the field at all.
The simulator can be used in two ways. First, it can simulate the run of a pacman as fast as the computing environment allows. The only information that is extracted from this simulation is the fitness. Second, we can pick an individual from the population, insert it in the simulator, and graphically watch how it behaves. Note that the same pacman will (probably) not complete the field two times in the same way, since the strategy evaluation is somewhat probabilistic. More about the fitness evalutation in the section covering the experiments.

3.3 Selection

Since the mutations in GP generally introduce rather substantial changes in the affected individual, we have added a small catch to the selection process. When the fitnesses of the individuals are measured, the best 10% of them are made untouchable by mutation. This way, we do not run the risk of having some of the best strategies destroyed by mutation. After all, the mutation is expected to bring about at least as many negative changes as positive.
For natural selection, we use *tournament selection*. As many times as there are individuals, we select two individuals at random. Their performance is measured in the simulator, and a copy of the faster (fitter) one replaces the slower one. By modifying the existing population (instead of creating a whole new population), we are sure to maintain some of the fittest individuals.

3.4 Crossover

The system is initialized with probabilities for crossover (p_b) and mutation (p_m). The crossover operation used selects two parents at random, and with a probability p_m, it computes two offspring by exchanging subtrees of corresponding types in the strategies. That is, we locate a random point in the strategy tree of parent 1. Next, we select at random a node **of the same type** in parent 2, respecting the size of the strategy tree of that parent. These are then swapped.

3.5 Mutation

There are many types of mutation available in GP. With probability p_m, an individual is mutated with a random of the following mutations:

- **Subtree swap**
  A random subnode with two subtrees of the same type is selected, and the subtrees are swapped. For example,
  `if DistanceToCheese(left) < 5 then ...`
  can be mutated into
  `if 5 < DistanceToCheese(left) then ...`
  and so on.

- **Constant mutation**
  Here we modify the constant terminal values. An integer can be changed to a number up to 50% higher or lower, and constant directions can be assigned another direction at random (i.e. `up` can be changed to `down` and so).

- **Grow**
  Here, we replace a terminal with a subtree of the same type. For example, a constant direction `left` can be replaced by a subtree that instead computes a direction using some of the terminals that inform of the state of the playfield or the pacman.[11]

- **Shrink**
  This is the opposite of the `Grow` mutation operator. A subtree is replaced by a simple constant value of the corresponding type. It is operators like `Grow` and `Shrink` that make the mutations in genetic programming induce more profound changes in the individuals, compared to the usually quite small changes made to potential solutions in an evolutionary algorithm for numerical problems.

- **Operator switch**
  This mutation operator replaces a node with another node that has the same number and types of subnodes. For example, **3 + 4** can be mutated into **3 * 4**, or a boolean **AND** could be replaced by **OR**.

4 Experiments

For this section, we have evolved two generations of pacmen. The first one is evolved from an entirely random initial population, the second one included a relatively good hardcoded strategy for the evolution to "continue". Since the fitness evaluation is quite nondeterministic, we have to test a strategy by letting it simulate a run more than once, and then use the average number of moves required to finish the field as the fitness measure.

[11] Due to technical problems with the implementation, this operator is currently disabled.

Performance graphs for the two evolved populations can be seen on pages 84 and 85. The test setup is a population of 50 individuals, evolved over 100 generations, with the fitness evaluation performing 10 simulations of each strategy. A larger population is not much of an advantage, see conclusion. The probabilities for crossover and mutation are 30% and 40%, respectively.

The experiments have shown that a strategy that gives points exclusively to the **DNC** terminal, can sometimes finish the field very quickly (300 moves or so). This is understandable. The **DNC** node is good at clearing up local areas, but as soon as no cheeses are visible (as described earlier), the pacman will perform completely random movements (because the **DNC** node will then not add points to any direction, making the decision a random selection of legal directions). At other times it essentially performs random walk the majority of the time, and may require 10.000 moves or more to finish the field. The hardcoded strategy is quite stable in the sense that it always uses between approximately 400 and 700 moves to finish the field. The hardcoded strategy looks like this:

Add **10** points to [if **MoreUp** then **Up** else **Down**] AND Add **10** points to [if **MoreLeft** the **left** else **right**] AND Add **30** points to **CurrentDirection** AND Add **60** points to **DNC**

This quite simple strategy is able to *always* finish the field quite fast illustrates the strength of the **DNC** terminal. In fact, when the hardcoded strategy is included in the initial population, the evolution seems to end up with two kinds of strategies. One kind is mostly copies of the hardcoded strategy, with only minor changes in the points given. The other kind is mostly strategies that add a lot of points to **DNC** only. These strategies survive because of their sometimes very fast runs, which cover up for their usually slower performance (than the hardcoded).

5 Conclusion

Since the introduction of the **DNC** node, the evolution has become somewhat trivial. In a sense, the node is a bit "too good". By adding many points to this node, a pacman will have a tendency to finish large areas of cheeses with a minimum of useless wandering around. Only when no cheeses are visible from the location of the pacman, the strategy has to come up with some way of locating unexplored areas. But often, more than one half of the field can be cleared by a single "good run" by the **DNC** node. As can be seen from the performance graphs on page 84 and 85, the evolutionary process finds good strategies almost immediately after it starts. The average fitness improves over generations, but the best individual usually does not improve that much.

One way that the process could be lead to create better (but not that much) strategies is to build into the fitness function some measure of deviation, so that there is some punishment for large variations in required number of moves. This way, we could evolve more reliable strategies, although a different strategy might in special cases be faster.

There are 194 cheeses in the field initially. A pacman walking around at random will finish the field in usually around 8.000 moves, so at first we just hoped for something better than that.

The evolution comes up with at least one strategy that can finish the field in about 600 moves relatively quickly. Taken into concideration that there are 194 cheeses, there is not that much room for improvement, which takes some of the motivation for inventing new tree nodes away. The success of the pacman is not solely creditable to the evolutionary process, but more likely to the obvious strength of the **DNC** node.

The best run we have seen so far is 330 moves, by an evolved strategy, that was pretty bad in general, so in the record-setting run, it was just lucky.

The graphical simulator has been a useful tool in the development of new strategy tree nodes, as the introduction of new constructs usually yield new behaviour properties, that are hard to evaluate otherwise. In this fashion, we have used the graphical simulator as sort of an aesthetical fitness function. By observing the behaviour of the pacmen in the simulator, it becomes more clear what type of nodes are needed to achieve further improvements.

The implementation has certain drawbacks, among these is the relatively large amount of code it takes to introduce new tree nodes. For each new node, several methods must be implemented. This could have been automated if a different style of implementation had been chosen.

References

[1] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

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

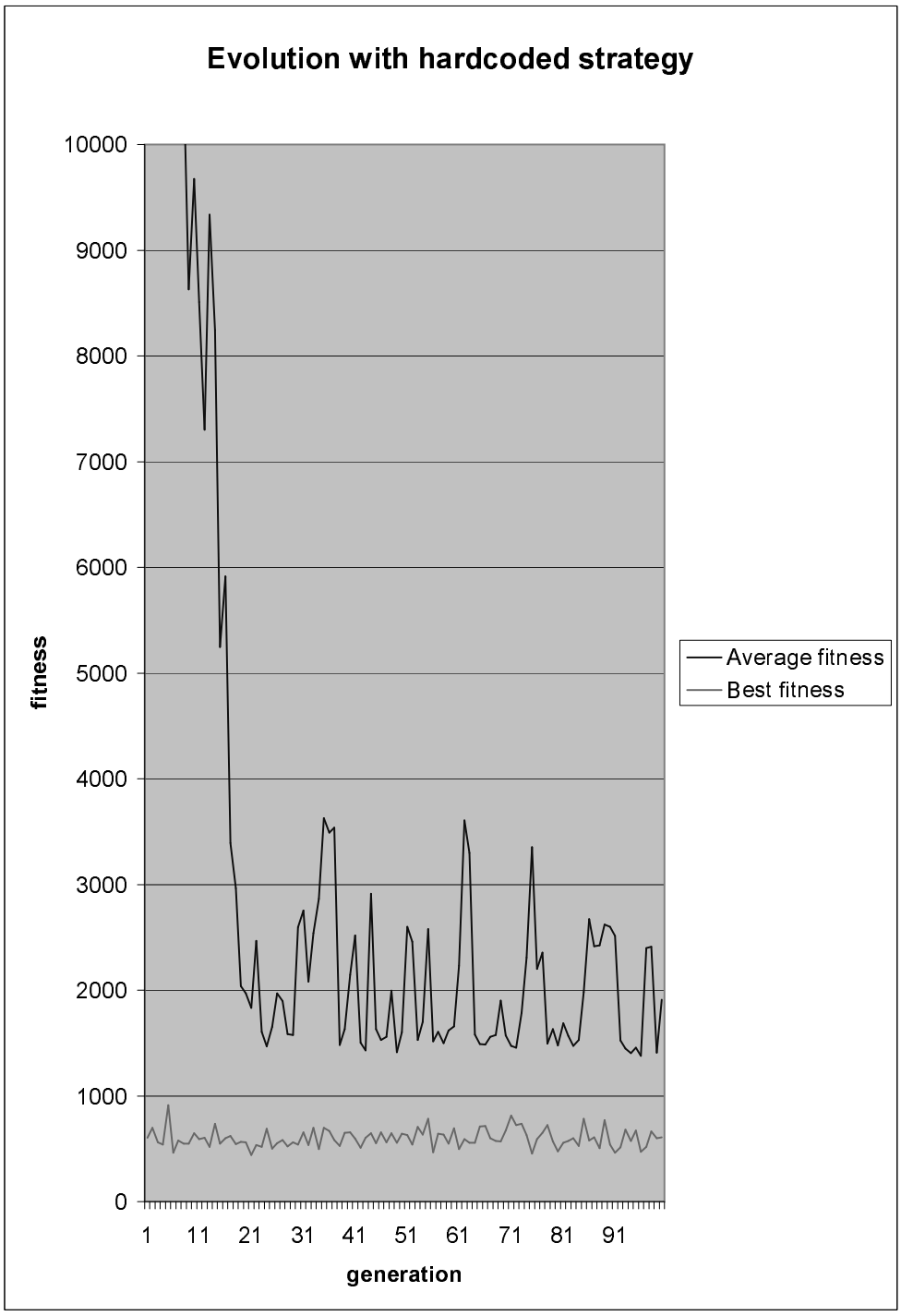


Fig. 1: Evolution with hardcoded strategy

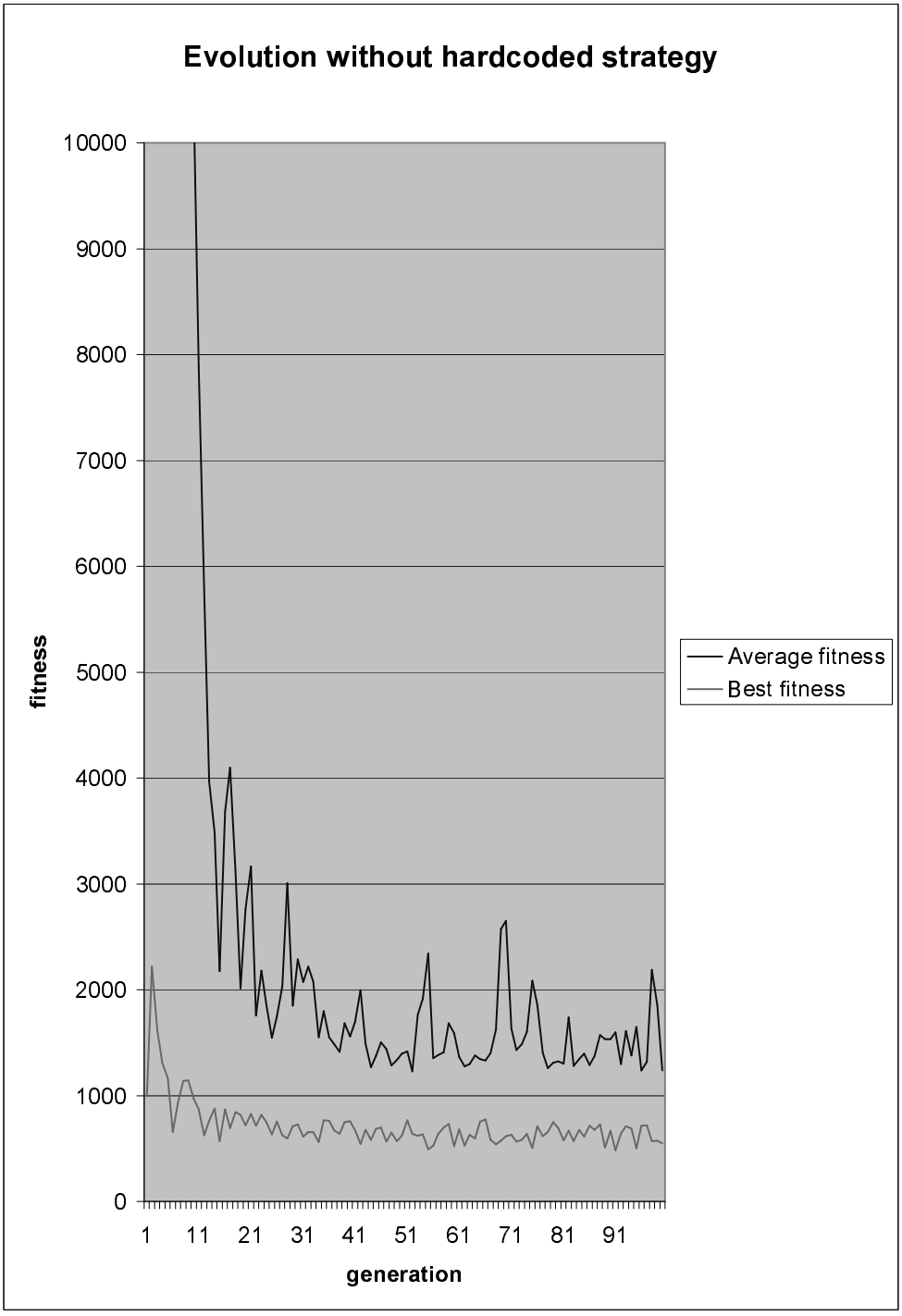


Fig. 2: Evolution without hardcoded strategy

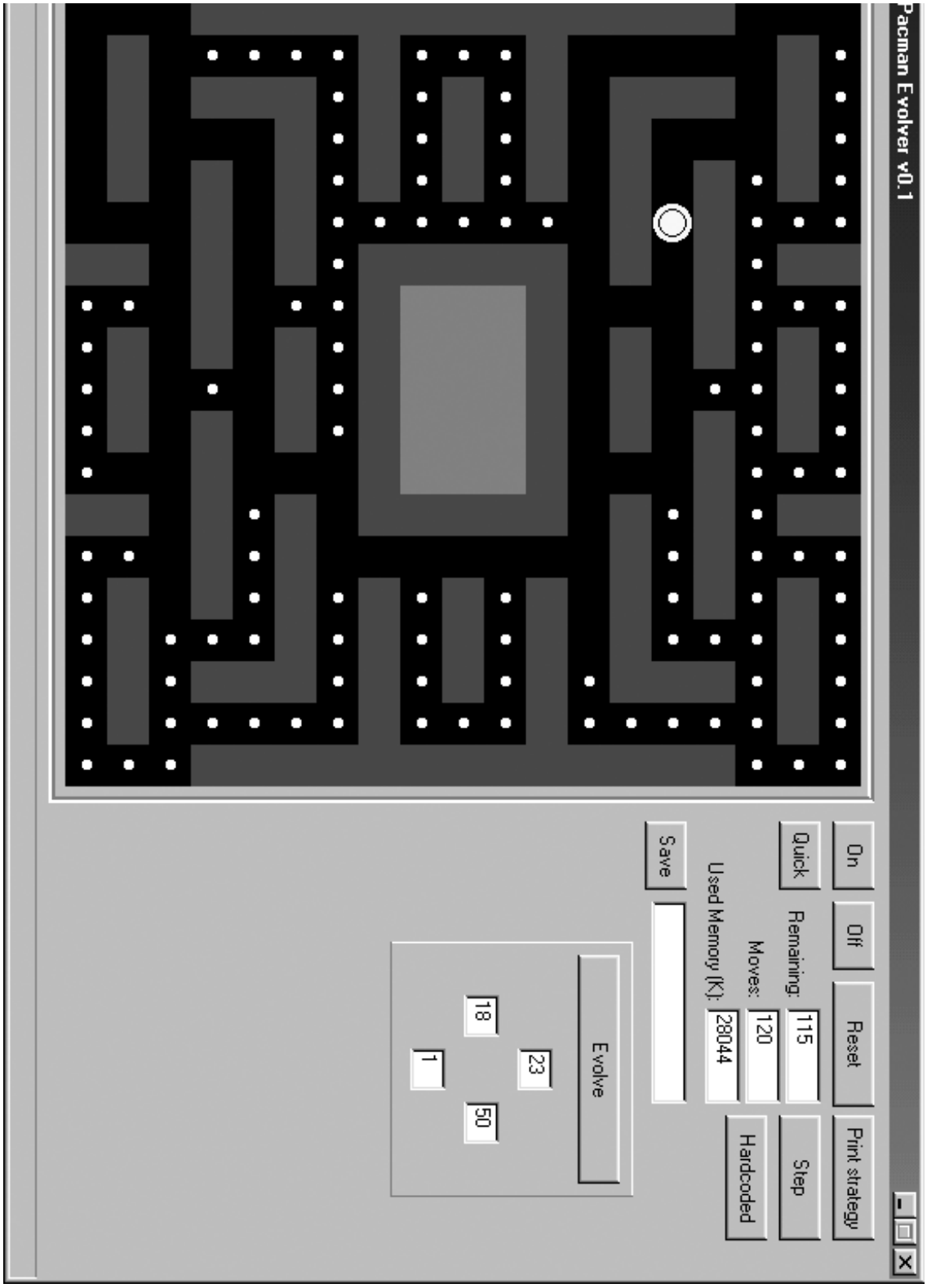


Fig. 3: Screenshot of the graphical simulator

Different Evolutionary Approaches to Football Expertise

Henrik Pedersen and Jan Midtgaard

Abstract—**This article describes three different Genetic Programming approaches to achieving football expertise. In each approach we try to predict the results of soccer matches on the basis of previous results. The goal of our project was to use the acquired evolutionary skills on some practical problem, and as a side-effect gain wealth. The approaches results in expert trees better at forecasting than random guessing, but not quite as good as a human expert.**

1 Introduction

The following sections describe three more or less similar approaches to gaining football expertise. All of them are based on Genetic Programming, and the purpose is simply to be able to predict the outcome of upcoming football matches on the basis of former results.

The three approaches cover

Class based trees This variant of decision trees is a more or less traditional Machine Learning technique.

Point based trees Inspired heavily by [4] where the idea is to award points to either of the teams.

Grammar based genetic programming Here, tree construction is based on a fixed grammar, and the parse trees evolve in the evolutionary process.

We have limited the involved dataset for the project to the premier league of English football. The results are from approximately 5 and a half season: 1996 up to December 2001. This set is divided in two parts:

A training set On which we evaluate fitness in the evolutionary process, and

A validation set On which the developed trees are tested and compared.

Since the trees base their predictions on previous results, season '96 is given as initial experience, thus the trees end up trying to predict seasons '97-'00, in total 1520 games. The current season ('01) has been chosen as our validation set, in total 133 games (see figure 1).

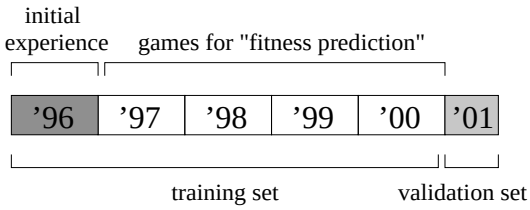


Figure 1: The data set.

2 What makes a good expert?

Before we even started to consider, what the trees should look like, we asked ourselves the following question:

What kind of knowledge is required to make a good football expert?

To answer this question we consulted some actual football experts (bookmakers, web-pages etc.), and came up with a bunch of ways to measure the strength of any team, solely based on the results of previous games. For instance one could ask:

What is the average number of goals Manchester United scored within the last four games?

We also discovered, that more subtle knowledge is used, like

What players are out because of injuries?

These kind of issues are of course much more difficult to deal with, and therefore we chose to ignore them.

So now we had a good starting point: Given a game, compare the strength of the two teams and give a suggested result as output.

2.1 Comparisons

The strength of a team is based on such things as average score, current position and so on. We have predefined 12 meaningful comparisons, some of which are described below. Each comparison returns a number. Positive values favour the home-team, negative values favour the away-team and values around zero favour none.

For instance the average score comparison looks like this:

avgscore(home-team) - avgscore(away-team)

Here are some of the comparisons:

Position Return the difference of the two teams position.

Points(int n) Return the difference between the points obtained within the last n games.

AvgScoreHere(int n) Compare the average number of goals scored, but only consider home-goals for the home-team and away-goals for the away-team, instead of just comparing the overall average. Again n is the number of games to go back in history.

Games Since Last... Now, imagine that there is a rule saying, that sooner or later any team will eventually have a draw. What this comparison does is to suggest the result, that is most probable when considering the number of games played since the last 1, X or 2. For instance if it has been 8 games since both teams played a draw, this comparison will suggest a draw, because sooner or later they will have a draw. Quite fuzzy, but worth mentioning...

Internal Return the point difference from games played between these two teams.

One important thing to notice is, that for some comparisons, you can specify how far to go back in history. Most of our comparisons go back 1... 5 games, but we can also tell the comparison to look at an entire season up till the current one. This might turn out to be very important for predicting results.

3 Class Based Trees

Our first approach is based on a very simple observation: Each game belongs to one of three possible classes:

1, X or 2

Therefore, if you are given a game, all you have to do is to classify it.

One possible way of solving this task is to construct a decision tree. Decision trees classify instances by sorting them down the tree from the root to some leaf, which provides the classification of the instance. Each node in the tree specifies a test of some *attribute* of the instance, and each branch descending from that node corresponds to one of the possible values for this attribute.

Koza discovered that a Genetic Programming approach was actually able to compete quite good with standard algorithms, such as ID3 [1]. Therefore we decided on trying this (see figure 2).

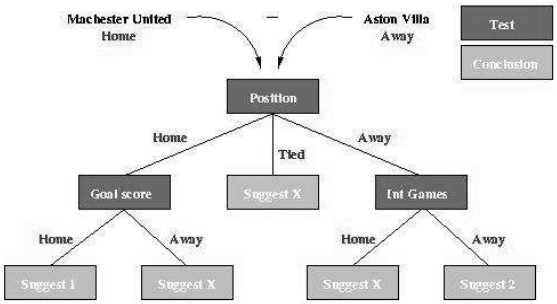


Figure 2: Example tree.

3.1 The Nodes

Each Node should represent some attribute for describing and classifying a game. This is where the comparison functions described above, come into the picture.

A node can have 12 different kinds, all implemented on top of the preprogrammed comparison functions.

Each node is at construction time assigned either two or three legs. In case it is assigned only two legs, the node decides to branch either the left or right child by evaluating the nodes function and inspecting the output. A threshold (`comp`) determines the separation point between branching left or right (see figure 3).

In case a node is assigned 3 legs another variable - `indexLen` - comes into play. This determines the interval of function (difference) values around `comp`, in which the node decides to branch the middle-leg (see figure 3).

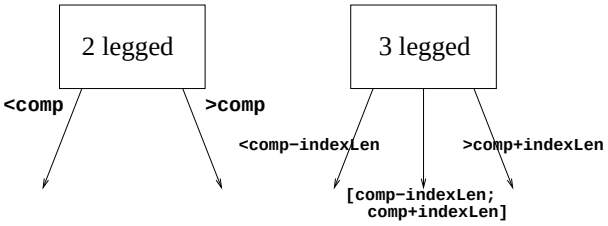


Figure 3: The nodes.

Now, there are two important things to notice here. First of all, some comparison-functions take an argument, telling them how far to go back in history for instance. This argument is saved as part of the Node and then used, when the Node actually calls it's function. Therefore, there are potentially much more Nodes than just 12. If each kind of Node should be represented in each tree, the trees would be enormous, which contradicts the *inductive bias* of decision trees (A preference for smaller trees over large trees [3]). To reduce the complexity we don't allow a path to have more than one occurrence of each Node type.

Second of all, in ordinary decision trees the attribute values are discrete, which is not the case in our trees, since many comparison-functions return *double* values. This makes it much more difficult to find the appropriate thresholds at each node. To handle this problem, we decided that each node can only have two or three children. This, first of all, reduces the size and complexity of the tree, because fewer thresholds have to be found. Furthermore it seems reasonable to have at most three children per node, since we only have three classes.

3.2 The Leaves

Each Leaf holds a value, which represents it's class. The possible classes are 1,X and 2, which corresponds to home-win, draw and away-win respectively.

3.3 The Evolutionary Algorithm

The EA used here, is just a simple standard EA. The main part of the algorithm looks like this:

```
initialize()
evaluate()

while(n < GENERATIONS)
  select()
  crossover(c_o_rate)
  mutate(mut_rate)
  evaluate()
```

Below each step is described in more detail.

initialise

Here we build a population of random trees, where each node has either two or three legs. The number of nodes in the tree is between 1 and an upper limit, *scale_factor*, which is one of the parameters for the EA.

select

Selection is done by tournament selection of size two. If two trees have the same fitness, we prefer the smaller one, because this is consistent with the assumed inductive bias, mentioned above.

crossover

Crossover is done the usual way by picking a node at random in both trees, and then switching the two subtrees obtained this way.

mutate

There are several kinds of mutation:

Expansion Expanding a leaf into new internal node.

Cutoff Cutting of a subtree at a random internal node, and substituting with a leaf.

Kind Switching Switching the type of the node into another node of random kind.

Constraints

We have made the constraint on trees in the population that having two or more nodes of the same kind on any path from root to leaf is not allowed, since it does not make sense consulting the dataset for (nearly) the same query twice. As described in Michalewicz's book [5] this is achieved by reducing fitness of the constraint breaking individual.

3.4 The Class Based Specific EA

The previous section pretty much captures the basic idea used in the Point Based Trees also. This section goes deeper into how the Class Based Trees are really evolved.

Constraints

It sometimes happens, that almost all games in the training-set ends at the very same leaf. In other words, there are trees suggesting that almost all games belong to the same class, which is not true. Therefore we search for such leaves and replace them with a randomly generated subtree.

We also make sure, that for no Node, all of it's leaves suggest the same class. If this is the case, we try to change one or more of the classes in the Leaves. Alternatively one could also shrink such a Node into a single Leaf with the given class.

The two steps

Actually, the evolving of Class Based Trees is separated into two parts. In the first part we try to find a set of Nodes, that are good classifiers. In the second part we then use these Nodes for constructing entire decision trees.

What does it take to be a good classifier? Well, in traditional Machine Learning techniques, one talks about *entropy*. We will not get into details here. The point is, that given entropy as a measure of the impurity in a collection of training examples, we can define a measure of the effectiveness of an attribute in classifying the training data. This measure is called *information gain*.

In step one we use the information gain as fitness. Mutation is only made on `comp` (see figure 3) and `goi`, which is short for *Games Of Interest* - how far should a given Node go back in history.

In step two we develop entire decision trees by using only the Nodes found in step one. Besides basic mutation we also mutate the Leaves, by changing their classes. Here the fitness is different. We simply return the number of correct classifications.

4 Point Based Trees

The point trees was inspired by TSang, Li and Butler's *Eddie*-article [4], using genetic programming to forecast horse racing results among other things. In this article the authors use a point giving system in which tree nodes ask expert questions and awards each horse points on the basis of the answer.

Using this idea each node in the point based tree asks expert questions regarding each game using the preprogrammed functions (see the *Functions*-section). The node then awards points to either the home- or away team, or maybe neither of the teams, and furthermore traverses down the left, right or middle leg respectively.

Each match is thus evaluated on the root of a point based tree, forecasting a home-win (1) if the difference between the home teams points and the away teams points is above a certain threshold `upper`, an away-win (2) if the difference is below a certain threshold `lower`, and a draw (X) otherwise.

4.1 Nodes

Each point based node has a scaling factor `scale`, which is multiplied with the function value before either of the teams is awarded the product of the two. In this sense `scale` determines how much influence the node should have on the final forecast.

In case a node is assigned only two legs, evaluation forces the node to favour either the home or away-team by awarding one of them its points, if the function value is greater or less than the node's threshold (`comp`) respectively (see figure 3).

In case a node is assigned 3 legs, the node can decide to assign points to neither of the teams, and traverse the middle-leg down the further evaluation if the function value is inside the interval determined by `comp` and `indexLen` (see figure 3).

4.2 The Point Based Specific EA

The EA on point based trees works as described in the last section on class based trees. The set of point based specific issues in the EA is described in the following section.

Node specific mutation

The EA on point based trees also performs a more specific mutation on the node-level of the trees. The algorithm traverses the tree, and with a certain probability `comp` or `scale` is mutated, corresponding to changing the "breakpoint" of the node and the node's total influence respectively.

Fitness

The fitness of point based trees is simply the number of correctly forecasted results. This number is thus between 0 and 1520.

Adding of genetic material

The point based algorithm adds new genetic material to the evolution, by adding a single random tree to the population in each generation. The new tree simply replaces a random tree in the existing population.

Constraints

In this approach we simply withdraw 100 from the fitness of an individual for every double occurrence of the same node type on a path from root to leaf.

Dynamic fitness

The fitness of each tree in the point based EA changes over time as the evolution progresses. The dynamic calculation is based on the following:

Avg_1 : Average point difference between the two teams for a 1.

Avg_X : Average point difference between the two teams for a X.

Avg_2 : Average point difference between the two teams for a 2.

All three numbers are averages over all results of the evaluation in the previous generation. New `upper` and `lower` values are then calculated as the averages between Avg_1 and Avg_X, and Avg_X and Avg_2 respectively. In this way each tree tunes its boundary points on the basis of its previous evaluation values. At the same time this means that each tree is not guaranteed the same fitness in the following evaluation. This dynamic fine-tuning is therefore stopped halfway through the generations, locking the boundary limits `upper` and `lower` for the rest of the evolution.

5 Grammar Based Genetic Programming

In 2000 two Chinese computer scientists, M.L. Wong and K.S. Leung, published a book on a system called LOGENPRO [5]. LOGENPRO is a framework, that combines inductive logic programming and genetic programming to do data mining. The basic idea is, that you provide it with some sort of building blocks for constructing logic rules, and then it will develop rules based on that.

What we did, was to find a hole new set of comparisons separated into three groups: Comparisons that we would expect to be relevant for predicting 1's, 2's and X's respectively. We came up with 33 such rules.

These comparisons were then used for building a set of logic rules, using only AND, OR and NOT (see figure 4).

Note: Unfortunately we haven't been able to find LOGENPRO anywhere, so we just

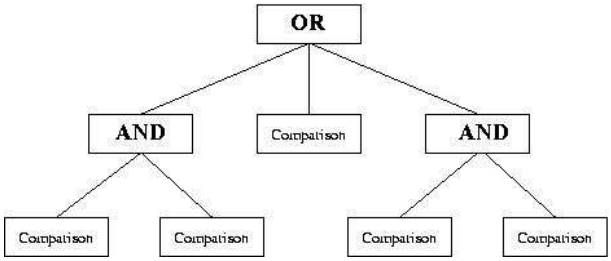


Figure 4: Example tree.

adopted the basic ideas and tried to make our own version. The development of this approach is therefore still on an early stage, but since we have spent a lot of time on it, we think that it is worth mentioning...

5.1 Comparisons

All comparisons take a form, that is more or less similar to this one:

$$avgScore(HOMETEAM) > avgScore(AWAYTEAM) + N$$

where `N` is a positive number, which could be considered, what is sometimes called a *handicap* in favour of the away-team. This rule is expected to be relevant for predicting 1's, because it is obviously only true, if the home-team scores more goals than the away-team.

Here is a rule, that we expect might be relevant for predicting draws:

$$avgScore(HOME) - avgScore(AWAY) < N \text{ AND } avgConc(HOME) - avgConc(AWAY) < N$$

This is true if the teams have a similar goal-score and a similar number of goals conceived. We guess, that in such case, the game might end up tied.

5.2 The Leaves

Each Leaf represents a comparison, and can therefore take 33 types. It holds the thresholds `N` and `goi`, telling how far to go back in history when evaluating the comparison.

The Leaves are evolved in a separate run, allowing mutation of `N`, `goi` and the Leaf-type. As with the Class Based Trees the fitness is just the *information gain*.

5.3 The Nodes

We have two kinds of Nodes, each having two children. There is an AndNode and an OrNode. Each of these has four variants: One for each combination when negating one, two or none of the children.

In this way each tree contributes with a logical rule, that is either true or false.

5.4 Developing Trees

At this point we have tried evolving the trees in many different ways, and we have tried many different kinds of fitness calculation.

At a very early stage we simply tried to evolve rules for 1's, X's and 2's separately, but it occurred to us, that finding a good fitness in this case was actually quite hard. The best fitness calculation was simply to give the tree a point if it evaluated to true on a given game, and the result of this game was the one we were looking for. On the other hand we would withdraw one point if the rule evaluated to true and the result was different than the one we were looking for.

Unfortunately it seemed, that most trees evaluated to the same truth-value for almost all games, resulting in very bad fitnesses. So, we adopted the idea behind decision tress: What about considering each rule as some kind of classifier, and then award it by it's ability to classify the games based on *information gain*? Why is this interesting? Well, if you can make even better classifiers, than the nodes in the class based trees, perhaps you could develop even better decision trees, by using these rules as Nodes.

The grammar based trees evolve the rules by first making a set of good leaves, or comparisons. Then we try to combine these with different variants of AndNodes, since and-rules are more powerful than or-rules. Then at the end we try to combine the best and-rules with different variants of OrNodes too see if this will give a better *information gain*.

6 Random forecaster

As a quality reference for the three approaches we have implemented a random forecaster. Even random forecasting is not completely trivial. For one thing draws do not occur as often as a win, and a loss occurs even less often as can be seen in table 1.

	1	X	2	total
training set	724	404	392	1520
percentage	47.6%	26.6%	25.8%	100%
validation set	50	45	38	133
percentage	37.6%	33.8%	28.6%	100%
total	774	449	430	1653
percentage	46.8%	27.2%	26.0%	100%

Table 1: The relationship between number of home-wins, draws and home-loses

The random forecaster was run on the validation set with parameters corresponding the last line in table 1. The results can be seen in table 2.

Run	1	X	2	Total
1.	24	11	6	41
2.	29	9	8	46
3.	28	12	10	50
4.	25	9	11	45
5.	20	13	16	49
Average	25.2	10.8	10.2	46.2

Table 2: Outcome from running the random forecaster.

The random forecaster ended up forecasting an average of $\frac{46.2}{133} = 34.7\%$ of the games in the validation set. It is worth mentioning that this corresponds to the one third, obtained from randomly forecasting each outcome with equal weights.

7 The results

Our results show, that the point based approach is by far the best, and therefore most of the testing has been made on this version.

7.1 Point based approach

Performance

The results of a single run of the point based algorithm can be seen in figure 5. The algorithm was run with a population size of 15 for 200 generations, and crossover-rate 0.4, mutation-rate 0.3 and node specific mutation-rate of 0.2. From each generation the fitness of the best individual is plotted in the figure.

There are several things to notice about the figure. For one thing there is a nice convergence toward fitter - best - individuals. Another thing that is worth noticing is the clear difference between the two halves of the evolutionary process. Because of the dynamic fitness, the fitness of the best individual in the population may vary from generation to generation. This results in a fluctuating first half. The second half is more continuous, and this part resembles more or less the fitness of a traditional EA or hill-climber. One notices however, that the fittest individual does not necessarily survive. This can be due to mutation of the best individual, by pure bad luck in the selection process or because the individual is exchanged with the random individual thrown into the population. In the first half this may also be because of the dynamic adjustment of `upper` and `lower` boundaries, on which the fitness is based.

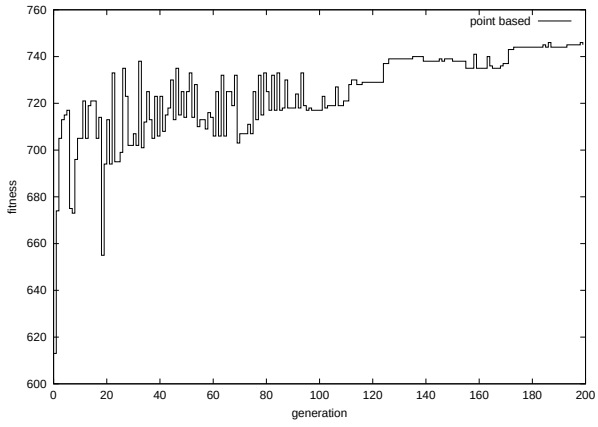


Figure 5: Evolving point based trees.

The reason for not explicitly keeping the fittest individual in the population is to insure diversity. With explicit keeping of the fittest

individual, the algorithm can get too focused on fitter individuals. With this less broad collection of genetic material, the algorithm may converge too fast, and end up in a local optimum.

7.2 Experiments on point based trees

Dynamic fitness

We have made a number of experiments concerning the point based tree approach. One important experiment was testing the concept of dynamic fitness. Does it make sense to move the boundaries `lower` and `upper`, and if so when should the tuning stop, if it should stop at all?

In figure 6 the result of running a static EA, a completely dynamic EA, and an EA stopping the dynamic tuning after one third of the generations is plotted. Each EA had 20 individuals, and equal crossover- and mutation-rate of 0.3.

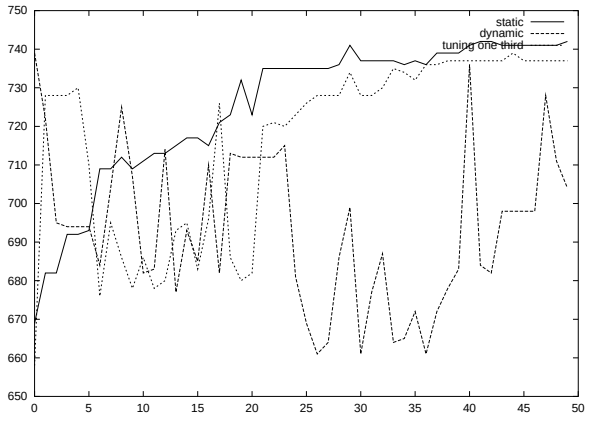


Figure 6: Comparing degrees of dynamic-Ness

It is noticed how poor the totally dynamic EA is capable of keeping it's population fit, because of the constant parameter tuning. The version adjusting only in the beginning seems more promising, and the static EA seems to perform best. This is not necessarily true in general, since the optimal `upper` and `lower` limits for each tree may be quite different, and especially from the hard-coded values assigned to the trees of the static EA. The figure thus illustrates the difficulty of optimising a varying function, a problem which is very hard, as is known from for instance *dynamic job shop scheduling.*

Number of legs

Another line of experiments were related to the nodes of the point based tree. These are by nature randomly made either 2 or 3 legged, but one could argue that they should be only 2 legged or only 3 legged overall. For one thing giving the middle leg option to the nodes just makes the search space even larger. Another more specific reason for keeping for instance only 2 legged nodes, could be to force the trees to separate the teams on a clear border, and thereby handing out more points than in a strict 3 legged version. In this way qualities of poorer teams could be acknowledged by getting points, more or less in a brute-force manor. On the other hand a strict 3 legged version would have an advantage on recognising X's, since every single node is given the option of not awarding any of the teams.

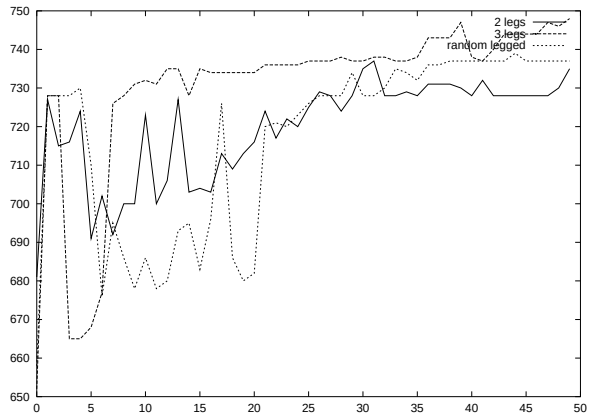


Figure 7: Comparing number of legs in nodes

In figure 7 the results of running a random-legged, a 2 legged and a 3 legged version of the point based EA with 20 individuals for 50 generations can be seen. The 3 legged version does seem to gain advantage from the third leg, and ends up with a better result than the random version. This may be due to the larger search space, which the random version has to handle. The random version performs quite well with this in mind, though.

Addition of genetic material

The last experiments were concerned with the addition of a new random tree in each generation. This can turn out to be a bad idea, since it substitutes a random individual in the population, in the worst case the fittest.

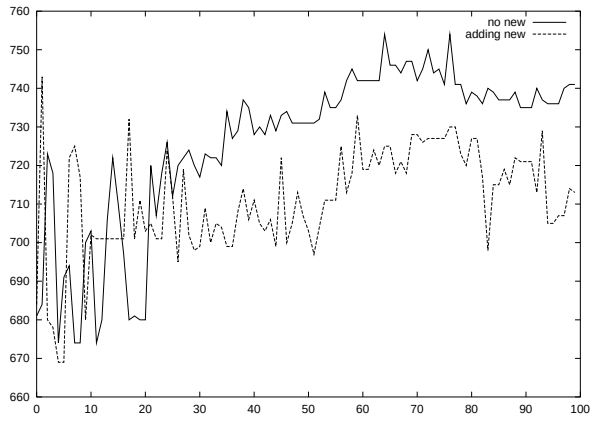


Figure 8: Addition vs. no addition of new trees

In figure 8 two runs of the EA is plotted. Both of the runs with same parameters and a population size of 20. As can be seen in the figure this adding does make quite a difference, substituting the fittest individual in the run already in the second generation. Cancelling this addition does not guarantee "the survival of the fittest" however, as can also be seen.

Conclusion of experiments

On the basis of the previous subsections a more or less optimal configuration of the point based EA, would be without the addition of random trees (or maybe letting it substitute the least fit individual in the population), with all nodes being (random or) 3-legged and maybe with a gradually lowering of the tuning of the fitness bounds. However we have not had time to perform such experiments.

Best tree

The most successful tree of the above, when later run on the validation set, has a fitness of 747. It was produced by the point based EA in a version with no random tree adding. The fitness corresponds to a forecasting 49.1% of the 1520 games in the training set. When run on the validation set the same tree was able to predict 57 of the 133 games, corresponding to 42.9% (see table 3).

	1	X	2	Total
Predicted	46	4	7	57
Percentage	92.0%	8.9%	18.4%	42.9%

Table 3: Validation results for the point based approach

One should notice that the tree has most difficulties forecasting a draw, all though a game ends more often with a X than with 2, at least according to the statistic material on our dataset (see table 1). The reason that it actually ends up predicting more 2's, must be that these are easier to predict on the basis of previous results, though occurring less often than X's.

The tree can be seen in figure 9. One can notice how the tree divides the games in two halves, by having placed an 'InternalNode' in the root. It does make sense, that among the things having influence on the outcome of a game, the outcome of the previous internal games between the two teams, is one of the more important ones. All paths from the root down has an 'avgGoal' or 'avgConceived'-Node of some kind. Again it makes good sense after the first classification, to further classify the teams and award points on the basis of their ability to score goals, and keeping their own goal clean.

7.3 Class based approach

The class based approach did not turn out that good compared to the point based approach. We believe that this is due to the fact, that there is no connection between the class, a leaf provides, and the path down to it. Remember that the comparisons are actually made to produce positive values, when the home-team is favoured, and negative values, when the away-team is favoured. This is used in the point based trees.

No tree broke 630 in fitness. But the exciting thing about this, is that one can expect the information gain to be high. What does this

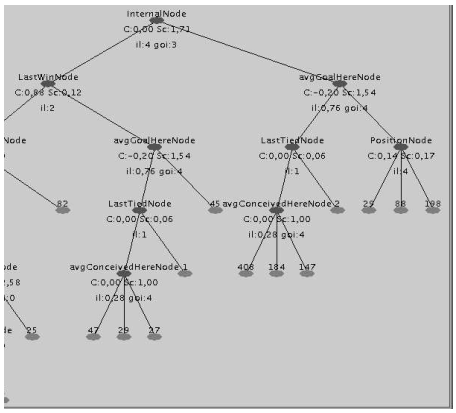


Figure 9: The best tree produced by the point based approach.

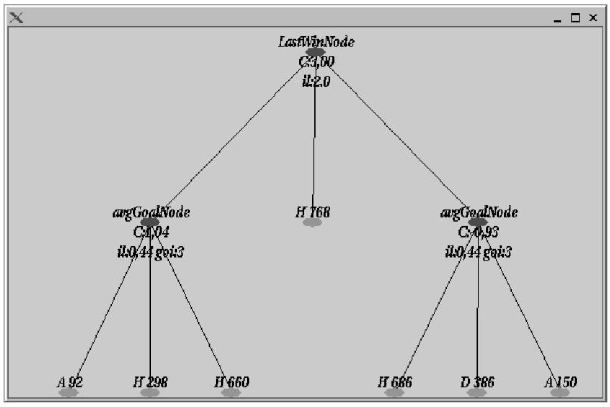


Figure 10: The best tree produced by the class based approach

mean? Well, for instance one tree of fitness 617, guessed 82% of the 1's in the training set, and almost no X's and 2's. This suggests, that one could maybe combine different trees, and make some kind of voting for a prediction.

It is worth mentioning, that the best individual was able to predict 56 games in the validation set (see table 4).

	1	X	2	Total
Predicted	45	3	8	56
Percentage	86.5%	7.8%	17.8%	42.1%

Table 4: Validation results for the class based approach

The corresponding tree is shown in figure 10.

This is a very small tree, and it's actually not that interesting, since most games end up in the same few leaves. By changing the class of these leaves, one could make this tree an expert on X's instead of ones.

7.4 Grammar based approach

All results for this approach are based on calculating information gain. We didn't really come up with any super-classifiers. Some of the best ones looked like this:

What is interesting though, is that these might actually turn out to be very useful as nodes in decision trees. Remember, that the final decision depends on evaluating more nodes. Therefore, what we see above is that we are able to construct nodes, that will increase the probability of for instance a 1, and maybe combining such nodes in a special manner, will give better decision trees, than the ones found in the class based approach.

	1	X	2
Percentage	60%	37%	43%
Percentage	10%	61%	57%

Table 5: Results for the grammar based approach

8 Conclusions

Not all three approaches to forecasting football results were equally successful, and not all approaches were given equal attention. Two of them outperformed the random forecaster though. This seems satisfactory when one bears in mind that the validation set does not resemble the training set much: The difference between the percentage of home-wins (1's) in the two sets is as high as 10%. Since home-wins seem the easiest result to forecast, this could have quite an impact on the performance of our approaches when tried validated.

In the project we have made a number of limitations. For one thing, none of the trees is given more background information than the previous results. A true expert would know all

details regarding players, injuries, team setup, etc...

Another thing is team-specific-Ness: With the current set of functions all nodes should reflect a general property: E.g. teams playing an away game against another better scoring team, has a disadvantage. The trees are not able to recognise a specific team - for instance Manchester United's difficulties playing away games against Nottingham Forrest. A true expert would indeed be team specific in this sense.

Finally the trees lack the ability to forecast surprises. The surprising element is an important part of the game football. It makes the games worth watching, since a result is never certain until the game has been played. However this same element makes life harder for our EA's.

Therefore we are satisfied with the quality of our predictions, even though they cannot match a true human expert.

References

[1] J. R. Koza. Concept formation and decision tree induction using the genetic programming paradigm. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature - Proceedings of 1st Workshop, PPSN 1*, volume 496 of *Lecture Notes in Computer Science*, pages 124–128, Dortmund, Germany, 1-3 October 1991. Springer-Verlag.

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

[3] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, Singapore, international edition, 1997. Chapter 3, Decision Tree Learning.

[4] Edward P. K. Tsang, Jin Li, and James M. Butler. EDDIE beats the bookies. *Software: Practice and Experience*, 28(10):1033–1043, 1998.

[5] M.L. Wong and K.S. Leung. *Data Mining Using Grammar Based Genetic Programming and Applications*. Kluwer Academic Publishers, Norwell, Massachusetts 02061 USA, 2000. Chapter 5, The Logic Grammar Based Genetic Programming System (LOGENPRO).

Evolutionary Fractal Image Compression

Aske S. Christensen and Kasper V. Lund

Abstract— **Fractal image compression is a compression technique, which relies on the vast amount of self similarities present in natural images. In this article we investigate an evolutionary approach to fractal image compression, and an approach based on local search. The local search approach is able to compress realistic images with a high compression factor while maintaining reasonable quality, but for compression aming at high quality it cannot compete with standard JPEG compression.**

1 Introduction

Fractal image compression is a compression technique, which relies on the vast amount of self similarities present in natural images. The self similarities are used to represent parts of an image as mappings from other parts of the same image. Figure 1 shows self similar regions in an image widely used as a benchmark for image compression algorithms. The mappings normally used consist of affine transformations from parts of the image combined with adjustments of contrast and brightness. For technical reasons the mappings must be contractive, but it is possible to use non-exact mappings if the compression is allowed to be destructive.

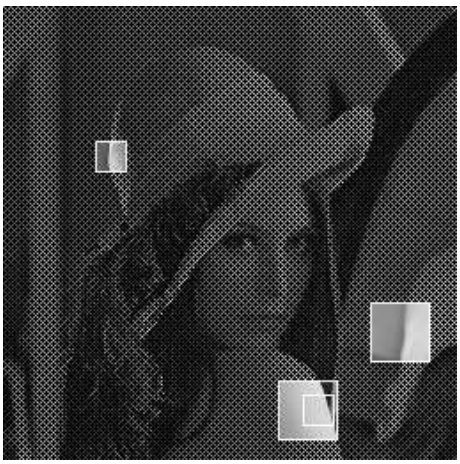


Figure 1: Self similar regions

During encoding the image is partitioned into a set of non-overlapping regions, and the image is searched for a good contractive mapping for each of the regions. One way of partitioning a given image is depicted in figure 2. It is possible to regenerate the original image (or an approximation thereof in case of non-exact mappings) from the regions and their associated mappings and therefore the regions and the mappings can be used as the representation of the image. In most cases this representation will turn out to be much more compact than the original pixel-based representation.

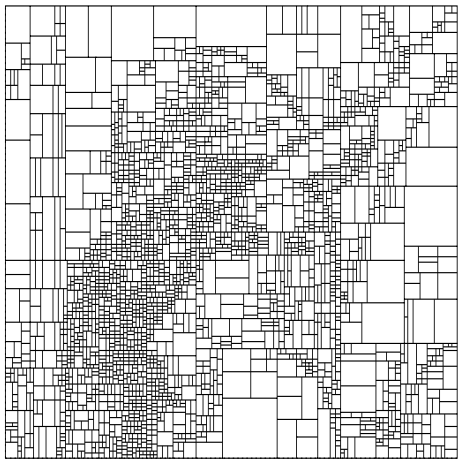


Figure 2: Partitioning during encoding

The decoding of a fractal compressed image has its mathematical founding in the contractive mapping fixed point theorem, which is explained in some detail in [1]. Starting from some arbitrary image, I_0, the decoding process iteratively applies the entire set of mappings found in the compressed representation to the result of the previous iteration,

$$I_i = \bigcup_{m \in M} m(I_{i-1})$$

where I_i is the result of the ith iteration and M is the set of mappings. The contractive mapping fixed point theorem states that if the set of mappings consists of contractive mappings then there exists a unique image I, which satisfies

$$I = \lim_{i \to \infty} I_i$$

The decoding process will terminate when this unique fixed point is obtained. Figure 3 visually illustrates how decoding works by showing four intermediate steps from a decoding process.

Figure 3: Decoding a compressed image

2 Representation

We partition the image using a quad tree, as described in [1]. Specifically, each node in the tree covers some square region of the image. A non-leaf node always has exactly four children, covering each of the four squares in the 2×2 symmetrical partitioning of the square covered by their parent. The root of the tree covers the entire image. Thus, the set of all the leaves of the tree describes a partition of the image into non-overlapping regions.

Each leaf contains all parameters for the mapping that is to represent the region covered by the leaf. These parameters consist of two things:

- The affine transform that is to be applied to the image to transform some part of it into the region covered by the leaf.
- The contrast and brightness adjustment that is to be applied to the transformed image region.

The affine transform is given by six numbers (u_{dx}, u_{dy}, u_0, v_{dx}, v_{dy}, v_0) such that the

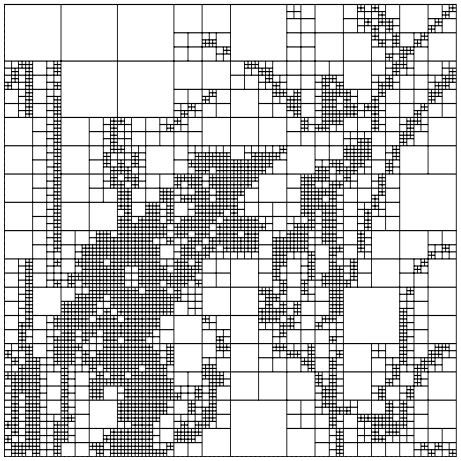


Figure 4: Quadtree partitioning

image coordinates (u, v) that contains the pixel to be transferred to the image region coordinates (x, y) is given by

$$u = u_{dx} \cdot x + u_{dy} \cdot y + u_0$$
$$v = v_{dx} \cdot x + v_{dy} \cdot y + v_0$$

Note that (u_0, v_0) is the image coordinates that transforms to the upper left corner of the image region.

The actual parameters that describe the transformation are the image coordinates of three corners of the parallelogram to be transformed into the image section. If (u_0, v_0) transforms to the upper left corner, (u_1, v_1) to the upper right corner and (u_2, v_2) to the lower left corner, and the side length of the region is l, then the transform is given by

$$u_{dx} = l^{-1} \cdot (u_1 - u_0)$$
$$u_{dy} = l^{-1} \cdot (u_2 - u_0)$$
$$v_{dx} = l^{-1} \cdot (v_1 - v_0)$$
$$v_{dy} = l^{-1} \cdot (v_2 - v_0)$$

Contractiveness of a transformation is the property that the distance between any two points in the image is decreased by the transformation. We define a transformation to be contractive by a factor of f, provided that all distances shorten by more than a factor of f by applying the transformation. Mathematically, the this is equivalent to all of the following inequalities being satisfied,

$$u_{dx}^2 + v_{dx}^2 > f^2$$
$$u_{dy}^2 + v_{dy}^2 > f^2$$
$$(u_{dx} + u_{dy})^2 + (v_{dx} + v_{dy})^2 > 2 \cdot f^2$$
$$(u_{dx} - u_{dy})^2 + (v_{dx} - v_{dy})^2 > 2 \cdot f^2$$

The contrast and brightness adjustments are numbers s and o describing a linear function on pixel values,

$$b = s \cdot a + o$$

where a is the value of the original pixel and b is the resulting pixel value. This function is applied to all pixels of the region after the transformation.

The optimal values for s and o are the values that minimizes the distance between the image region contents produced by the mapping and the actual contents of that region of the original image. With the standard mean square distance measure on images (n being the number of pixels and all summations implicitly over all pixels),

$$\frac{1}{n}\sum(a-b)^2$$

we want to minimize the quantity

$$\frac{1}{n}\sum(s \cdot a + o - b)^2$$

which is achieved by choosing o to be

$$o = \frac{\sum b - s \cdot \sum a}{n}$$

and by choosing s to be

$$s = \begin{cases} \dfrac{n \cdot \sum ab - \sum a \sum b}{d} & \text{if } d \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

where d is defined as

$$d \equiv n \cdot \sum a^2 - (\sum a)^2$$

To actually store the image to a file, we first decide on valid ranges and number of bits (at most eight) used to represent the eight mapping parameters. The tree is then serialized by a normal depth-first traversal with the following representations:

- Node: One zero byte followed by serialization of leaves.
- Leaf: Eight bytes containing each of the mapping parameters s, o, u_0, v_0, $(u_1 - u_0)$, $(v_1 - v_0)$, $(u_2 - u_0)$, and $(v_2 - v_0)$. If s is sufficiently small we only store s and o, since it does not make much difference which transformation parameters are used (because the result is almost a uniformly colored square).

The result is packed with `gzip` to achieve good compression. We expect that `gzip` is able to make good use of the fact that most of the time we do not use the full range of the bytes because of our restrictions on the number of bits used to represent the parameters.

3 Evolutionary Approach

The search space of the fractal image compression problem is huge. There exists an incredible number of different representations, and using evolving representations using an evolutionary algorithm seems like the obvious (albeit a bit simple-minded) approach. The population in our model consists of individuals, which are the representations described in the previous section.

The evolution of good individuals is clearly a case of multi-objective optimization. On one hand we want as good an image quality as possible, but on the other hand we have to make sure, that the compression factor remains high. We use the weighted sum approach, and compute the *fitness* of a given individual as

$$\text{fitness} = \text{quality} + \text{size weight} \cdot \text{size}$$

where the quality is the standard mean square distance measure between the original image and the approximated image found by decompressing the representation. The size is the size of the compressed serialized representation. The goal of our evolutionary algorithm is to minimize the fitness of the best individual, without getting stuck in local optima.

To maintain a suitable selection pressure we use *tournament selection* with *elitism*. Using elitism ensures that the fitness of the best

individual in the population can never reduce from one generation to the next, and we have found (experimentally) that elitism improves our algorithm.

There are several ways of mutating an individual based on a quadtree representation. We have implemented a mutation operator, which uses three different mutation schemes:

- Mapping parameters in leaves can be mutated by adding a random vector with entries from $U(-\sigma, \sigma)$ to the parameter vector.
- Leaves can be partitioned into nodes with four sub-leaves.
- Entire sub-trees can be pruned by turning nodes into leaves.

The crossover operator works by exchanging sub-trees in two different individuals. Two nodes (one in each individual) representing the same image region are chosen at random and exchanged. The result is that the sets of mappings that cover the image regions are swapped.

To give an impression of how the evolution proceeds, figure 5 shows the decompression of the best fit individual after 5, 20, 50, and 100 generations.

Figure 5: Early steps of evolution

The result of the evolution process (after 1000 generations) is shown in figure 6. It is interesting to see that the quality has not improved considerably compared to the best fit individual after 100 generations.

Figure 6: Lena in 3741 bytes

4 Local Search Approach

The evolutionary approach described in section 3 is not able to produce leaves with mappings of a very high quality. The regions of the image shown in figure 6 do not have much texture, which seems to indicate, that the contrast setting in the leaves corresponding to the regions is close to zero and that it is the brightness setting alone, that defines the color of the individual regions.

To experiment with alternative ways of doing fractal image compression, and to solve some of the problems inherent in the evolutionary approach, we have designed and implemented an additional compression algorithm, based on our existing representation. The algorithm described in this section is an iterative algorithm, which iteratively improves a single representation using a local search technique. The improvement process continues until a termination condition is satisfied.

In the setup phase of the algorithm, the source image is partitioned into four regions, which is represented by a node and four leaves. For each of the four regions, the source image is searched for a good mapping using a local search technique described later in this section. Since this operation optimizes the mappings in the leaves of the representation, it is referred to as *leaf optimization*.

Each iteration tries to improve the overall quality of the compressed image by improving the quality of the worst region in the image. The worst region is defined as the region,

which has the highest product of the square root of its size and the squared mean distance to the source image. To improve the quality of a region, it is partitioned into four smaller regions. Since the partitioning itself does not change the compressed image and the quality thereof, we apply leaf optimization to the four leaves that represent the new regions. Figure 7 shows eight intermediate images from the early stages of the compression, thereby illustrating the partitioning process.

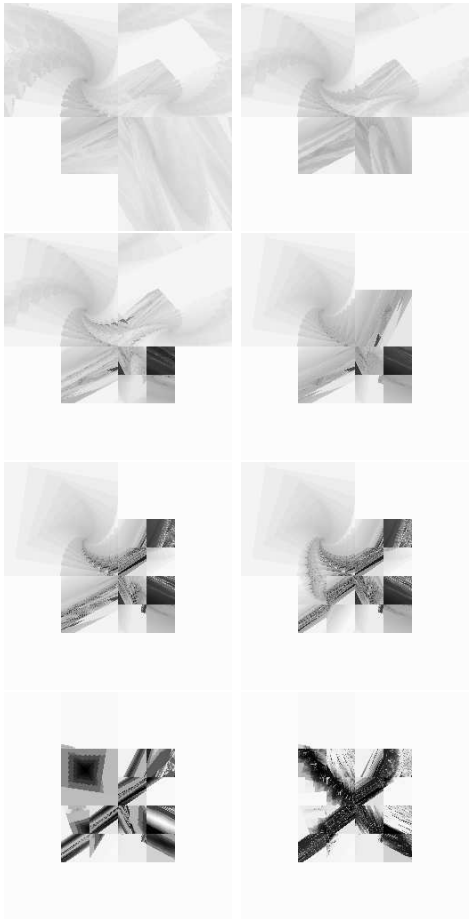


Figure 7: First eight iterations

The user can specify the compression goals in terms of the maximum allowed size of the compressed representation, and the minimum allowed quality of the decompressed image. The algorithm continues its iterations as long as both conditions are satisfied. To show how the settings affect the output we have compressed the same image as shown in figure 7 with two different maximum sizes. Figure 8 shows the results of the compression. Opposed to the size weight setting of the evolutionary approach, this is a much more intuitive and understandable way of expressing the compression goals.

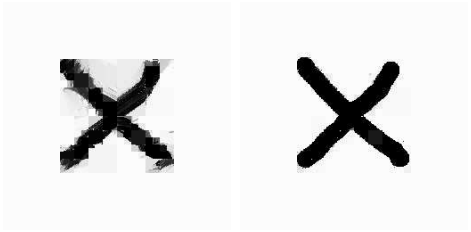


Figure 8: Effect of maximum size setting

To implement the leaf optimization part of our algorithm, we decided to use a simple local search method, commonly referred to as *hill-climbing* [5]. Given a leaf with an associated initial mapping our optimization routine tries to improve the mapping by randomly tweaking the mapping parameters. This improvement process continues until the random tweaking has not improved the quality of the mapping for a limited number of consecutive attempts. The threshold used to limit the number of unsuccessful attempts is known as the *stamina*, and its setting has a significant impact on the behavior of the algorithm. Figure 9 shows the effect of the stamina setting: The leftmost image is the result of decoding an image compressed with a low stamina setting, whereas the rightmost image has been compressed with a high stamina setting.

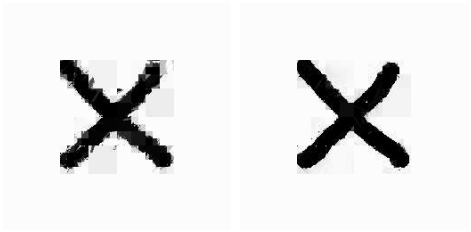


Figure 9: Effect of stamina setting

It is well-known that local search methods have a tendency of getting stuck in local optima. To help alleviate this shortcoming it would have been possible to use an algorithm based on *simulated annealing* [5] or *tabu search* [5]. Instead we decided to use an iterated version of the hill-climbing method. Each time the leaf optimization routine is used, the hill-climbing process is restarted a number of times; each time with a random initial map-

ping. The optimized leaf returned by the iterated hill-climbing method is the result of the iteration that produced the highest quality leaf. The number of iterations used per leaf optimization is known as the *climber count*, and like the setting of the stamina parameter, the climber count setting has severe implications on the efficiency of algorithm, and the resulting compression factor. To illustrate the effect of the climber count setting, we have compressed the same image using low and high climber count settings. The results are shown in figure 10.

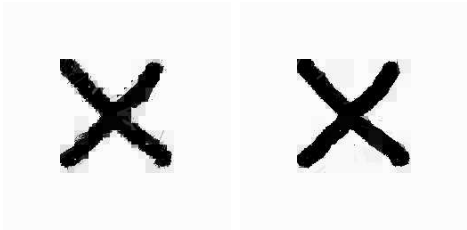


Figure 10: Effect of climber count setting

To get an impression of how the local search approach compares to the evolutionary approach, compare figure 11 to figure 6. Both figures show the same image, but the image in figure 11 has been compressed with the local search based algorithm described in this section. The compressed images are roughly the same size, but the quality of the local search based compression is much better.

Figure 11: Lena in 3800 bytes

5 Conclusions

To illustrate the (lack of) efficiency of the fractal image compression scheme described in this article, we have compared the local search approach to standard JPEG compression. Figure 12 shows the relationship between size and quality for both compression schemes. As can be seen from the graphs, JPEG significantly outperforms our algorithm for high quality compression. For cramming images into really small files, however, we do a little better than JPEG.

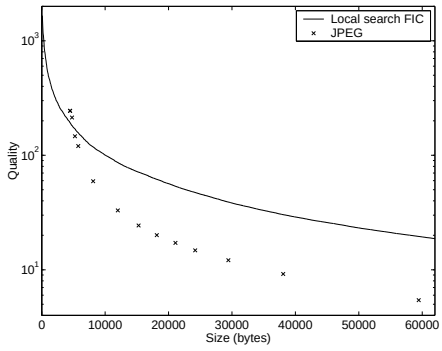


Figure 12: Comparison with JPEG

References

[1] Y. Fisher. Fractal image compression. ACM SIGGRAPH Course Notes 12, 1992.

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

Path planning in a 3 dimensional landscape

Guillaume Carré and Guillaume Farret

Abstract— **This report describes a genetic algorithms (GAs) based method which generate a path planning in a 3 dimensional landscape. Our approach leads us to a multi-criterion problem and also to different models for our pathway. Our program is suitable for both off-line and on-line path planning because of the fast response time by using this approach and the fact that this one doesn't need a continue landscape but just a discretized one. We first presents the models and the main ideas of this choices and then some simulations with 3D generated landscapes with *strange* shapes in order to show that this method can solve tricky problem. Finally we will see the results of a multi-criterion problem (Pareto front) and how this can be sensitive in some special cases. We finished with possible extensions for this project.**

1 Introduction

Searching a path between two points in a 3D environment is solving a problem that can be encountered in a large variety of real applications. A path planning software in combination with a GPS system in a car, the design of a new road constrained by factor like price and practicability for all type of cars, robot motion planning (*offline* if the path is calculated before any movement or *online* if the path is calculated in real time during the deplacement).

We have used evolutionary computation techniques to solve this problem which can be very tricky with a more classical approach. We have looked at some papers in this domain [2], [1] more to get a way of working on this problem than getting ideas. We first thought about two different representation which will be referred as *the discrete representation* and *the CheckPoint List*. The first goal that we set for this project was to design efficient operators to tackle our modelisation of the problem. Then we have designed some problems to verify the diverse behaviour of our algorithm. A not negligeable part of our project was also to find a good way to visualise the results of our algorithm and we figure out that some of the behaviour of our implementation were wrong while we thought to get good results. The third part was to test, analyse the results of the program and develop new idea to improve results. Then we discuss about further improvement and thing that we could have done if we had more time.

2 Preliminaries

Assume that path planning is considered to be a problem bounded in a rectangular space. The landscape is modelled with a discrete grid where each intersection point on this grid get some informations (height or others for a road building, presence of trees, kind of rocks...). A path between two locations is a way from a *start point* and a *destination point*: we suppose that the discretisation of the landscape is fine enough to say that the linear approximation between two adjacent points on the grid is a good approximation of the real shape of the landscape.

2.1 Path planning Problem

Problem: Find a pathway representation that link a start point S to a destination one D on the defined landscape.

Input: A m by n grid, S and D. These points are given by there coordinates in the landscape constraint.

Output: A pathway from S to D which optimise the given problem, this pathway depends of the model used to find it. It can be a sequence of adjacent points, a sequence of vectors with the direction and the size of this one, a sequence points which define a broken line or whatever. The only requirement is that we can plot the pathway on the 3D surface.

3 Models and genetical operators

The choice of the the models have to be lead by two questions:

- Is it a good one for the evaluation of the fitness ?
  i.e.: Can we easily know if this path is good or not.
- Will the genetical operations that we can implement be efficient ?
  i.e.: Our design fit well or not the way that the GAs works.

So, we modelled this problem in two different ways:

- The first one, (called discretisation in the gene) obviously optimise the first criterion [12] and we hope that with few additional works the second criterion could be realized.
- The second, (called checkpoint list) focus the modelisation around simple genetical operators in order to stay with a *classical* genetic algorithm that we know will work.

3.1 1^{st} representation : Discretisation in the Gene

This representation use the frame of the Standard GA with a tournament selection of size 2 and normal operator. Let's describe more accuratly this first model.

data representation :

The path is discretized all along the gene, so a chromosome is a sequence of adjacent points. For doing this, we look for intersection points of the grid which are the nearest from the line at every crossing point.

By doing this, we won't need any special computation to calculate the fitness of the gene. The risk is that the size of the gene could easily quickly grow up. This induce a lot of computation for the manipulation of this gene.

[12] the discretisation is done before the evaluation of the fitness, that save a lot of computation during the evaluation of an Individual.

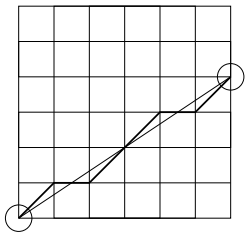
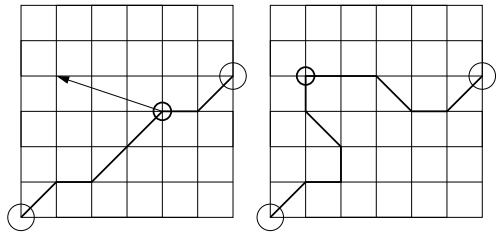


Figure 1:

mutation operation :

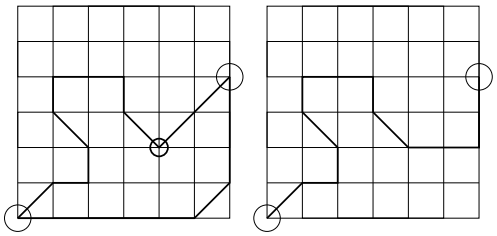


When a point has to be mutated, this mutation is done with a radius between 0 and a maximum along X and Y. In the same time we have to delete several points before and after the mutated one, in order to avoid to obtain tortuous pathway.

This operator introduce a new parameter: (RadiusX,RadiusY) and also another problem. We have to try different heuristics to calculate the number of points to delete: a constant number, $\sqrt{\text{mutation X}^2 + \text{mutation Y}^2}$, Max(mutationX, mutationY)... and maybe the optimal heuristic is function of the kind of problem that we have to solve. We are also not sure that this operator allow a good exploration of the search space and if does not lead every time to bad pathway that are more likely to be eliminated. So, with this model, the time of tuning will be longer for a classical GA.

crossover operation :

The crossover operation is quite simple: once a chromosome has to have a crossing over, a random point is chosen inside the chromosome and we link this path from this point to the other path at the nearest point. The generated offspring is the resultant path: the first part of the chromosome of the first parent and



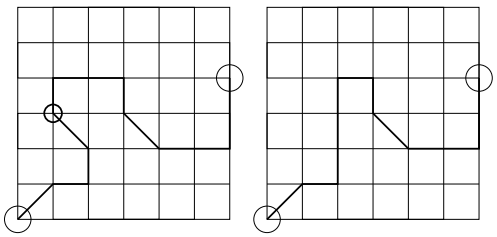
the second part of the chromosome of the second parent.

first parent: $[p_1, p_2, p_3, p_4]$
second parent: $[q_1, q_2, q_3, q_4, q_5, q_6, q_7]$
random split point: p_3
if $min(p_3, q_i)$ is for $i = 6$ then:
generated offspring: $[p_1, p_2, p_3, q_6, q_7]$

Here again, we have to pay attention to the distance that we use to calculate the closest point in the second parent (once the first split point has been chosen in the first parent). This distance can be of different type since we are on a discretized landscape. We can use the Euclidean distance, the Manhattan distance [13], the fitness function or other to decide where we have to join the two paths.

shrinking operator :

The two last operators (mutation & crossover) introduced quite a lot of points into our chromosomes. Such a number of points became a back draw after several computations because it introduce a lot of *noise* which conduct to a lot of inefficient pathway like loop or unexpected bad directions. The high number of points due to the modelisation increase this noise and reduce the efficiency of the GA. In order to get ride of this, we created another operator called shrinking.



[13]The distance between two points measured along axes at right angles. (also called rectilinear distance)

This one works like this: a point is randomly chosen in the chromosome, then this and several around him in the chromosome are deleted, finally the path is redrawn by linking the remaining points.

Individual: $[q_1, q_2, q_3, q_4, q_5, q_6, q_7]$
random delete point: q_3
delete radius: 1
shrinked Individual: $[q_1, q_5, q_6, q_7]$

The motivation for this representation was that the "whole" search space seems to be quite easily reached with this. The path can do some little curves and the fitness will be easy to compute because of the discretisation is already done.

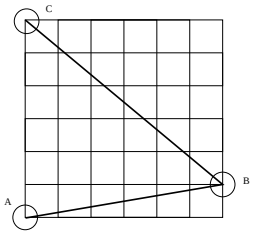
The problem is that we don't know if a GA works well or not on such problems. Most of the times, a lot of specificity have to be create when a representation is too far from a standard problem.

In order to check if this supposition was good, we did another model, really lighter but certainly less able to map the entire search space.

3.2 2^{nd} representation: The CheckPoint List

The second model follow also the frame of a classical GA with a steady state selection and based on multi objectives fitness attribution.

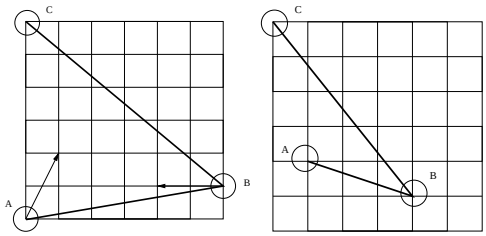
data representation :



With this model, the pathway is only the points which define the broken line. In this example this is [a,b,c]. This model is really smaller than the first one, but some computation will have to be done during the run to calculate the fitness (discretisation according

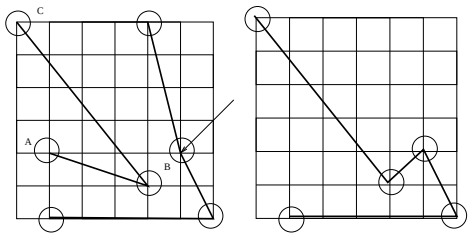
to a predefined sample rate to evaluate distance on the landscape for example).
We tried to write some operations that won't add a lot of point into the chromosomes. By doing this, we hope that we won't need some special operations like the shrinking operator. Let's see the mutation and the crossover operation:

mutation operation :



This mutation only move locally a point (or a list of consecutive points) of the path. This local move is constrained by a mutation radius which avoid the pathway to change to much. This new parameter is the more likely to be controlled by a power law (Sand Pile for example) to allow most of the time normal mutation and sometimes hyper mutation.

crossover operation :



A point of the path is selected at random in the first parent and we look for the nearest point in the other parent. Then, we link the two paths.

If there is really a lack of point into our chromosomes. We could try to write another crossover and mutation operations that could add some points. And then, using randomly both operators according to a given probability. With this kind of method, we could avoid potential reefs of this model.

4 The fitness function

A special fitness function

- Discrete representation
  For each step the gradient is computed and a function g is applied

  $$fitness(x) = \sum_{i=0}^{end} \mathrm{g}(\mathrm{gradient}_i) \ note\ [14]$$

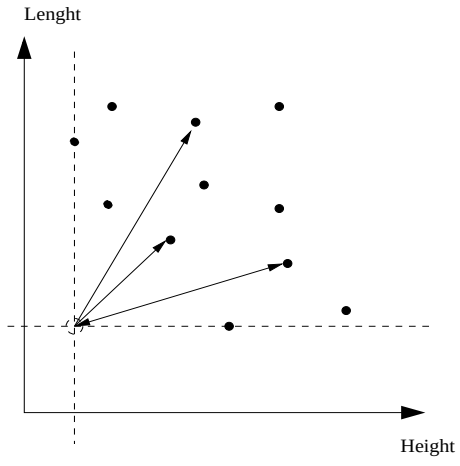
  If we want to insist on the length, we have to write a function g which will penalise a high gradient. On the other hand, a function with a little penalisation will be a shortest path in order to minimise the number of terms in the sum.

- The checkpoint list representation
  During the evaluation, two values are calculated for each gene :

  $$\text{the height} = \sum_{\Delta discrete} |\text{height climb}|$$

  the length = total plane distance of the pathway.
  (sum of euclidian distance between Check-Points of the pathway).

  Then, the fitness is the distance shown on this figure:



Then, we use this two values according to weights in order to find some solution which insist on one or the other criterion. This method is really common for the multi-criterion function.

[14] the gradient is the difference of height over the plane distance between two points.

5 Experiments

5.1 Introduction to the experiments

We do not show graphical results for the discretisation in the gene implementation with both of the problem because this method gave us too poor pathway.
There are several reasons:

- too static representation: the pathway is very "heavy to move" because of the too high number of points.
- too much induced computation: this method was very long too compute the genetic operation (crossover, mutation).
- inefficient operator: mutation operator and crossover lead to unexpected results due too the noise in the pathway.
- too much parameter control induced.

We focused our experimentation on the results of the "CheckPoint list" implementation. To do so we designed two problems to check some behaviours of this method.

5.2 The first problem

The first problem is defined as following:

landscape equation : $z = sin(x)^2 sin(y)^2$

xrange	=	$[-2\pi + 1; 2\pi - 1]$
yrange	=	$[-2\pi; 2\pi]$
start point	:	$(4.2, 6.2)$
end point	:	$(-3, -3.2)$

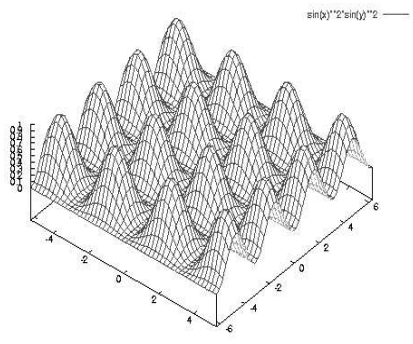
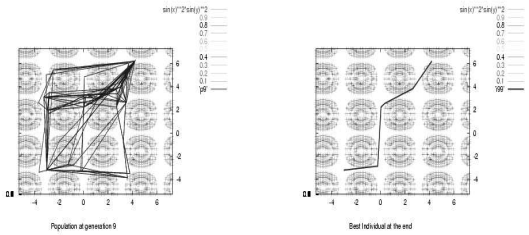


Figure 2: the bottle problem

This problem produce a landscape with high mountains that have to be avoided in order to minimise the 'difficulty' of the pathway. This problem is referenced after as the **bottles problem**.

The problem was too find a good compromise between "difficulty" and length of the pathway.



We can see that after a short number of generation, the algorithm has already found a few good solutions. The best solution is obtained after 40-50 generations. To find this solution we had to increase the weight for the length (for the weighted sum in the fitness). If we decrease this weight we obtain more angular pathway, with most of the time only one change of direction. The algo follows the first valley and directly goes to the goal when it's possible. Increasing the weight for the length leads to pathway that partially cross the mountain if this "cross" is not too difficult.

5.3 The second problem

The second problem is defined as following:

landscape equation :
$z = sin(10x)^3 sin(y)^3 - sin(10x+2) sin(y+2)^3$

xrange	=	$[-0.2; -0.02]$
yrange	=	$[0; 6.2]$
start point	:	$(-0.018, 0.3)$
end point	:	$(-0.18, 6)$

This problem is interesting because it provide us a mean to test our algorithm on a landscape with a huge local optimum for the optimisation of the landscape "difficulty". It will be referenced after as the **zig-zag problem**.

Here the problem was to find a good solution without a minimum length and that follow the less difficult steeple. Typically this problem could be the one encountered in the design

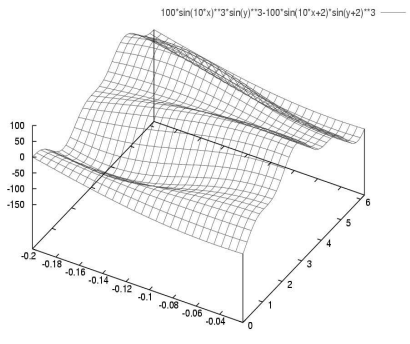
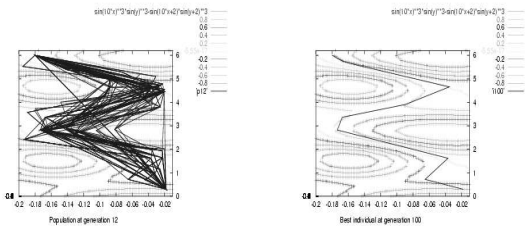


Figure 3: the zig-zag problem

of a road in a hilly landscape. We can see that the pathway follow exactly the best valleys to get to the goal. At the 20^{th} generation some pathway still cross the first mountain but the majority of the solution has already found the large zigzag. The further improvement of the path way are done in the beginning (downer right corner) to find the first little zigzag.



5.4 Multi objective optimisation

One of the problem when we first try to tackle this problem with a simple EA was to define a good fitness function. We first try to penalise the total height climbed with some function $fitness = e^{|gradient|}$ but we could not get good results on different kind of problems.
This kind of problem is likely to be solve with multi objectives optimisation because most of the time the maximum for both objectives (length & height) cannot be fulfilled. We often search for a good compromise between this two objectives.

We have computed this Pareto front directly from our representation. It has been obtained by running the program a "lot" of time with the vector (w_{height}, w_{length}) chosen at random to explore the whole front.

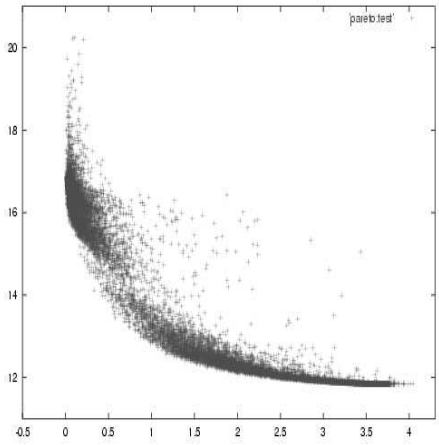


Figure 4: the pareto front on the first problem

We can observe that there are a very high density of points for couple of vector that are either favouring the length or the height of the pathway. It means that its quite easy to find a good solution if we want to insist on one of the two criterions. The bottle problem has been used to compute this pareto front.In the upper left corner, we can see solutions with a long pathway and a very easy difficulty and in the downer right corner solutions with a straight line from the start to the destination. The problem with this representation is that we don't have normalised the weight of the two criterions. It would have been interesting to have a good repartition of the point among the two axis to have a better interpretation. This could have been done for example in resampling non continuously both axis according to the number of points in a partition of the two axis. For example we could have done a partition of the solution space (x_{weight}, y_{length}) on the X axis and count for each part how many point there are inside and then increase the size of this part.

5.5 Conclusions on the experiments

With this two problems, we could see if our algorithms were efficient with some obstacles, like hills or mountains, that induce a lot of local optimum. The tests also allowed us to see the weakness of the discrete model. Such

results were quite hard to guess before, these tests lead us to think about the real behaviour of our algorithm.
An interesting result is the speed of our GA. On a grid of 40x40 (about 3 times larger than in the other papers), the time of computation was really short. After 30 generations of a population of 100 individuals we get approximatively our best result. This characteristic can maybe give some ideas for an on line planning.

6 What we could do if we had more time ...

This project and modelisation took quite a long time. And, before doing some funny stuff or improvement we preferred to validate our model on some problems as the two that we shown. By the way, we got a lot of ideas with this algorithm, this are several things that we thought about :

- As you can see on the last example, the diversity of the population is quite quickly lost. After only 30 generations, most of the population is reduced to 4 different individuals. A probably good improvement would be to use an heuristic to keep the diversity of the population without having a decrease of the performance of the genetic algorithms.
- Another improvement should be to add some others genetic operations in order to add more point into the chromosomes. We could use both operators randomly according to probability. Maybe with this kind of improvements, we could be able to find some results with really tricky problems (laces in a mountain).
- We could introduce a third criterion in order to design some specific problems: cost of digging a road on some special rocks, cost of cutting trees, cost of the fields ... With this stuff, we could test our algorithm on a real problem which is the road planning in a mountain.
- We could do a real time system. The algorithm is able to re-compute very quickly a new optimal pathway and could be able to react at a change of the landscape. For example, a robot that can't in fact follow the first computed path because of a weakness of one of his wheel or if the landscape change (sand dune in the desert).

7 Conclusions

This paper proposed two different ways to program path planning with genetic algorithms and their results. We saw a huge difference of results, the first model leads to poor results and the second to some good. After this experience, our belief is that when you try to model a problem for a genetic algorithm the leading idea must be the simplicity of the representation in order to stay in the well know behaviour of the GAs. The various length of our representations led the first one to a failure but not for the second one because we were able to maintain a reasonable number of points.

By doing this, we were able to use our GA with no X or Y monotone problems in opposition of the paper [2] in which one the writers preferred to use a fixed-length strings. So our GA is more able to find an optimal path offline and can also be use to find some motion planning but will be less efficient than the special design model.

The multi-criterion approach is good way to make our algorithm adaptif to a lot of different problems. This nonspecialization, according to our opinion, guarantees safety and leaves presager for many applications with such an algorithm.

References

[1] Juan Manuel Ahuactzin, El-Ghazali Talbi, Pierre Bessiere, and B. Mazer. Using genetic algorithms for robot motion planning. In *European Conference on Artificial Intelligence*, pages 671–675, 1992.

[2] K. Sugihara and J. Smith. Genetic algorithms for adaptive motion planning of an autonomous mobile robot, 1997.

Forking Particle Swarm Optimisation

Christian Gasser, Kasper S. Jensen, and Kari S. Schougaard

Abstract— **We have combined the forking GA model and the basic PSO model into a model which we have named the forking PSO model. So far our empirical tests show our model to be inferior to the basic PSO on most functions . This article will in detail explain what we have done so far and what needs to be done. It is still our firm belief that this model will beat both the forking GA and the PSO on complex functions.**

1 Introduction

In 1997 Tsutsui, S., Fujimoto, Y., and Ghosh, A. proposed the idea of a forking GA [4]. The basic search method of the forking GA is the standard GA. The standard EA will examine the search space as usual, but whenever it finds an interesting area in the search space and converges in the area, it forks a child-population. The child-population will now search the interesting area, while the parent-population will continue searching for other interesting areas. Child-populations themselves can fork and the nesting depth can be arbitrarily high.

Tsutsui et al. showed empirically that the forking GA is superior to the standard GA on complex function optimisation problems. The standard EA often converges on a local optimum. The genes of an individual sitting on a local hill will spread rapidly throughout the population and you will get a population with a very low diversity - poorly equipped to explore the search space for even better solutions. The forking algorithm keeps the diversity high throughout the run and is thus less prone to ending up in a local optimum. Another great advantage is the ease with which it can be distributed and run concurrently. Each subpopulation can be seen as an almost independent unit, which can run isolated with very little interaction with the other subpopulations. The disadvantage is of course the use of standard GA's in the populations. These require a relatively high number of individuals in each population and thus many fitness evaluations.

Another approach to finding good solutions for multi-modal problems is the Particle Swarm Optimisation technique. This model was introduced in 1995 by Kennedy [2]. The strength of the basic PSO is finding good solutions with very few fitness evaluations. This is achieved using a low number of particles, which fly around in the search space. Each particle remembers the best solution it has found so far. In addition it has knowledge of the best solution found by any particle. All the particles are then attracted to their own best solution and the global best solution. This gives them an acceleration- and a velocity-vector, which will make the particles move around the search landscape and converge on the best found solution. The basic PSO finds good solutions very fast, but the downside is that it can be trapped very easily on a local optimum.

Greatly inspired by Løvbjerg et al. [3] and their success in making a hybrid between a standard GA and a PSO we have tried to combine the good results from the forking GA with the low number of fitness evaluations from the basic PSO in a model we call Forking Particle Swarm Optimisation.

2 Forking PSO

In short our model works as a phenotypic forking GA, where the GA has been replaced by a basic PSO. On top of that we have introduced a queen, which spawns the initial swarm and which keeps control of how many resources are used in which areas. Our algorithm can be outlined as:

```
spawn(parents)
while(evaluations<maxEvaluations) {
  calculate which swarms is gonna fly
  for each swarm which is gonna fly {
    evaluate fitness
    if swarm has converged {
      spawn childswarm
      reinit parents outside this area
    }
    if childswarms has converged {
```

```
      merge chilswarms
    }
    calculate velocities
    move swarm
  }
  calculate new priorities
}
```

Where one loop in the whileloop is called a turn and there are many evaluations in one turn.

In the rest of this section we will describe our forking rules, boundary considerations and resource allocation.

2.1 Velocity update

Our model uses particles with certain positions in the search-space and a velocity-vector giving the particle's direction. As in the PSO model Kennedy presented in [1] the particles is inserted randomly in the search-space and with a random velocity-vector when the swarm is created.

The velocity of a particle is then calculated on basis of the current velocity with a stochastic correction in the direction of the best position this particle saw and the best position seen by any particle in the swarm:

$$\overrightarrow{v_i} \leftarrow \overrightarrow{v_i} + \varphi_1(\overrightarrow{p_i} - \overrightarrow{x_i}) + \varphi_2(\overrightarrow{p_g} - \overrightarrow{x_i})$$

where φ_1 and φ_2 are random numbers defined by their upper limit (usually 2.0). The index g is the index of the particle in the swarm with the best performance so far, so that p_g is the best vector found by any particle in the swarm.

After teh velocity has been calculated the position is updates as follows:

$$\overrightarrow{x_i} \leftarrow \overrightarrow{x_i} + \overrightarrow{v_i}$$

2.2 Forking rules

Tsutsui et al. [4] base their forking rules for the phenotypic forking algorithm upon what they call a hypercube neighbourhood. When a certain percentage of a population is within a hypercube centred around the position of the individual with the best fitness, forking will occur. The exploration of this neighbourhood will then be left to the child-population. In our terminology hypercube neighbourhoods are called areas. We examine the area around the best position a swarm ever saw, as this point is the one all particles in the swarm are attracted to. The current position of the particle that found the position can be somewhat different from this.

When to fork

Two criterias has to be met for a swarm to fork.

1. Over a certain treshold of the particles have to be inside a hypercube around the best position ever seen by the swarm. How the size of the hypercube is determined is explained in the next subsection.

2. The best value this swarm ever saw may not have been updated for some time - for example not for 20 times in a row.

Both of these criterias is used by Tsutsui et al. [4]. We use a larger value for the second criteria than proposed in the article. As a particle can stumble on a very good spot but have a good speed in some direction it will need some time to turn around and come back to look in this fine area. This means that often when the best value is changed, some time will elaps before we get a new change.

Threshold for forking and area-sizes

We define two parameters which controls the forking. The first parameter defines how many percent of a swarm has to be inside a given area before forking occur. That is $0.5 \leq K_{TS} \leq 1$ where K_{TS} is the threshold parameter. The second parameter defines the size of a child-area compared to the parent-area. You could for example define this parameter to be 16 which means the childareasize will be $1/16^{th}$ the size of the parent-area. If the children spawn some children themselves, the new area will be $1/256^{th}$ the size of the original parent-area and so on.

Keep child-areas inside boundary

When forking occurs there will be special cases which require special treatment. In particular

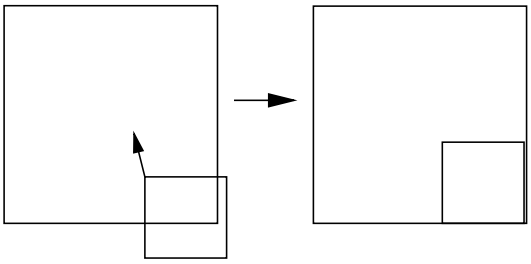


Figure 1: When creating a new area it can be out of bounds so it must be moved back in.

you will have to handle merging and overlapping. In the following we will explain how we handled these special cases.

Assume a swarm has converged near the border of its area. Now creating a child-area here will exceed the limits of its parent-area, (see figure 1). We have decided to move the borders of the child-area inside the parent-area. Alternatively you could just make the child-area smaller in which case you would have to consider how many resources you invest in that area.

Another special case you have to consider is when you want to create a new child-area, that will overlap with an existing child-area. Our solution allow areas to overlap. We have chosen this solution despite the obvious disadvantage of having two or more swarms searching the same search space. Firstly we give swarms a priority, which means they are allowed more evaluations if they have found an interesting area. Secondly if we have found a good area we don't mind using extra resources (two or more swarms) searching that area. This of course implies the necessity of a good priority-scheme. If two areas overlap a hypercube (of same size as the areas for the two swarms) which is placed with centre in the best particle in one of the swarms. If the hypercube contain more than the forking threshold of particles from both swarms the best of the particles in the swarms are transferred to a merged swarm in the new area (see fig. 2). Thus if a optimum is placed inside the areas of to different sibling swarms and both swarms converge on this optimum the swarms are merged.

2.3 Boundary rules

We think of child-areas as holes in the parent-areas. It is forbidden for parents to go into a child-area and a child has to stay in its own area. In the following we will explain what we do when a particle tries to go out of its boundaries.

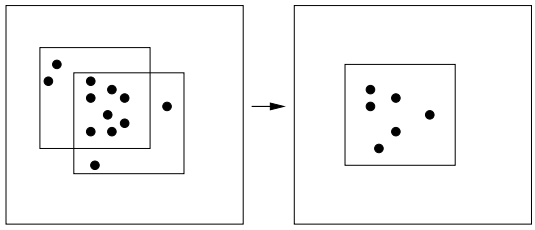


Figure 2: If two areas overlap and particles from both swarms converge. The existing areas are discarded and a new is created keeping the best of the particles from both swarms.

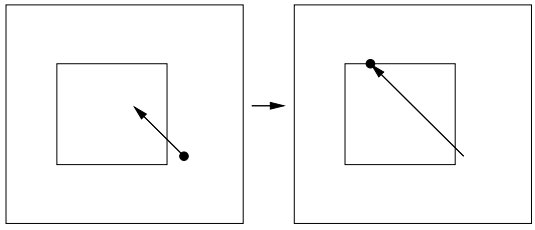


Figure 3: A particle cannot enter a forbidden area and if it tries it is transfered to the other side of the area.

When a swarm spawns a childswarm, the childswarm will be restricted to the area in which it was created and for the parents it will be a forbidden area. It can happen that a parent-particles velocity-vector will make it end up in a forbidden area. In this case we simply prolong the velocity-vector such that the particle end up on the border in that direction on the other side of the forbidden area, (see figure 3). This way we don't run the risk of trapping particles on a border of an area opposite of where it is attracted to, (see figure 4). When a particle tries to go beyond the boundaries of its area we move it to the nearest boundary and set it's velocity to zero. Now setting a particle's velocity to zero means that the next velocity it will receive will only be computed based on the solutions it is attracted to.

2.4 Resource allocation

The cornerstone in our model is the resource allocation. The goal is to obtain as good results as possible using the least possible fitness

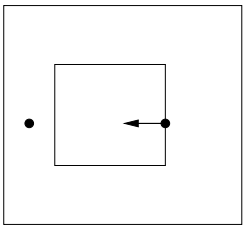


Figure 4: If the particle aren't transfered to the other side of the area it could still be attracted to a particle on the other side, thus getting trapped.

evaluations. To this end we work with suspension and priorities of swarms.

We have introduced a Queen to the particle model. The Queens task is to spawn the first swarm and keep an eye on how well each swarm is doing. If a swarm is doing well she should add more resources to that swarm. On the other hand, if a swarm is doing bad it should have less resources. There are two ways the resources can be controlled. An area can either get more particles or the particles in that area run more often. In this article we concentrate on the latter, which we call suspension.

To have a swarm evaluate more or less frequently than another swarm requires some kind of prioritising. Therefore we let the Queen control a priority-scheme. When a swarm forks, both the childswarm and the parent-swarm will get priority 0 meaning their particles are evaluated every turn. If a swarm is doing bad it will go down in the priority-scheme. The priorities correspond to exponentially decreasing number of evaluations such that priority i means evaluation every 2^i'th turn. Priority 8 is the lowest possible priority. This way we ensure that no area is completely discarded.

The best and worst values for all the swarms that flew in this turn is recorded. Now the queen prioritizes the swarms that flew in this turn: A swarm is evaluated after the following criterias. The middlevalue of the bestposition ever seen by this swarm and the best position obtained in this round by this swarm is compared to the best of the best and the worst af the best for all the swarms that flew in this turn. If the value is close to the best of the best the swarm gets into a better place in the queue, if it is close to the worst of the best the swarm is moved downwards in the queue.

3 Experiments

We started out with an implementation of a simple PSO that ran on four well known problems. The Sphere, Griewank, Rosenbrock and Rastrigin as described by Løvbjerg [3]. In order to compare the forking PSO to the basic model we ran our experiments on the same problems.

3.1 Tuning of parameters

The FPSO algorithm is dependent on a number of parameters, which greatly influence its behaviour. We have optimized these parameters using a standard GA. We found the following values to be optimal for each function:

Parameters	Sphere	Griewank	Rosenbrock	Rastrigin
Swarmsize	18	14-16	12-13	15-17
Stagnation	22	10-20	15	15-20
Phi	1.03	1.75-1.9	1.8-1.9	1.9-2.2
Weight	0.92	0.65-0.75	0.65-0.75	0.3/0.6-0.7
Max velocity	0.16	0.1/0.4	0.1-0.2	0.17/0.35
Threshold	0.95	0.75-0.95	0.75-0.9	0.9-1.0
Areasize	23	60	50-60	26/45-60

3.2 Testfunctions

To compare the models we focused on average best values measured at regular intervals. In contrast to the basic model and due to the priority-scheme the forking PSO lacks the notion of *generations*, instead we counted the *number* of fitness evaluations.

For some time we considered defining a diversity measure that could be compared, but gave up as we did not think it would be possible to devise a *fair* measure for the forking PSO - again due to the priority-scheme, where different swarms are run at differing intervals.

All our runs were conducted for ten dimensional problems, in each case the average is taken over fifty runs. As seen in the accompanying figures the fpso only performs better for the rosenbrock function.

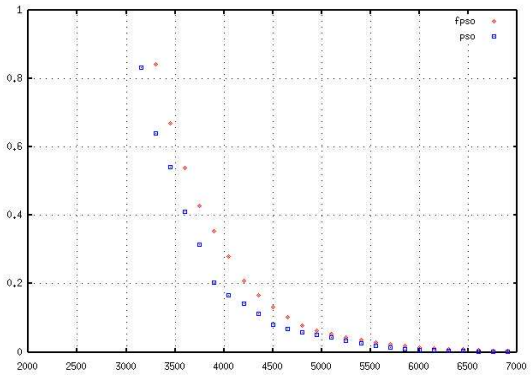


Figure 5: Results for the Sphere function

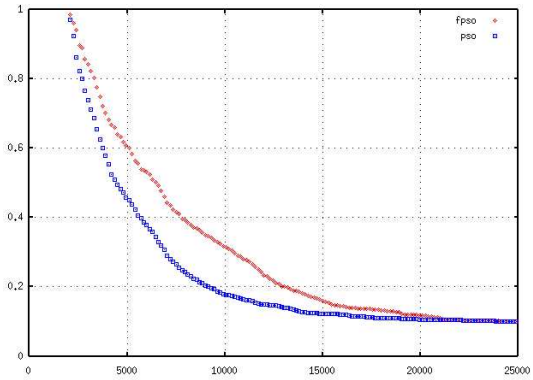


Figure 6: Results for the Griewank function

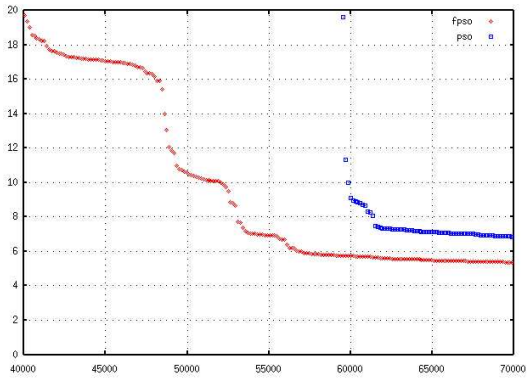


Figure 7: Results for the Rosenbrock function

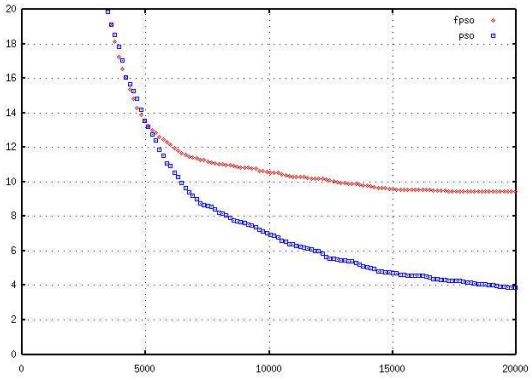


Figure 8: Results for the Rastrigin function

4 Results

4.1 Tuning of parameters

As can be seen from the results the same parameterset works well on all the testfunctions. We have found the following values: swarmsize 15, stagnation 15, phi 1.9, weight 0.7, max velocity 0.1, threshold 0.9 and areasize 60. Note, however, that small changes in the parameters can improve the results so for best results, please refer to 3.1

4.2 Testfunctions

As can be seen from the experiments the forking PSO converges more slowly than the basic PSO. In the case of the Rosenbrock function we see a remarkable staircase pattern. We haven't had time to explore this pattern further, but it is this staircase pattern we had hoped for on all the functions. In particular it could be interesting to see if the stairs begin whenever forking occur. The forking PSO run particularily bad on the Rastrigin function.

5 Conclusions

So far our algorithm are not doing well enough for most testfunctions. There are still alot of improvements which can be made and we think that the forking PSO will eventually be superior to the forking GA and the basic PSO.

6 Future Work

Since we did not beat the regular PSO with our implementation of a FPSO there is certainly room for improvement. The main difficulty in implementing a FPSO clearly lies in the distribution of resources, that is by *resources* we mean the number of all-in-all fitness evaluations.

Two different approaches come to mind, either one could improve the priority scheme used now or one could devise a new scheme that does not rule how often each swarm runs but instead distributes resources in a completely different way.

To improve the priority scheme different approaches are thinkable on how swarms get

their priorities increased or decreased. In our current model the priority for a particular swarm is altered based on how it performed compared to the other swarms in the same priority class. Instead or additionally we could base the decisions on whether the swarm has been stagnating for a long period or has subsequently been getting better for some time - experiments in that direction actually seemed promising.

When using the priority scheme it is of utmost importance to keep the number of swarms with high priorities low. Different measures to ensure this should be considered more thoroughly. In our FPSO model we only fork after a swarm has been stagnating, merge swarms if they search the nearly the same area and limit the number of forking levels - other rules are thinkable.

Since the priority scheme approach has proven to be hard to control we could try to implement other schemes. One idea focuses on limiting the number of particles, that is when a swarm wants to fork off a new swarm it requests particles from a controller which in return takes particles from the swarms doing worst - in this case a swarm would be abandoned when it contains less than some number of particles.

Another idea we had limits the number of swarms - and thus also particles. In this scheme forking is only possible if the limit of swarms has not been reached or when another swarm is destroyed at the same time.

In this scheme like in all the others there is some form of controller - we believe that the main difficulty in implementing a FPSO is to find suitable rules for these controllers.

7 Conclusions

8 Future work

References

[1] 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.

[2] 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.

[3] 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.

[4] Shigeyoshi Tsutsui, Yoshiji Fujimoto, and Ashish Ghosh. Forking genetic algorithms: Gas with search space division schemes. *Evolutionary Computation*, 5:61–80, 1997.

Evolving a mill player

Simon Nejmann, Kasper Fauerby, Mads Olesen, and Carsten Kjær

Abstract— **In this paper we describe our attempts to evolve a mill player, using an EA and a hill climber to train the weights of a neural network. The neural network is used to assign a quality measure to the different board states and thus allow the computer to decide on a move to make from a given state. The result is a fairly good mill player but unfortunately we experience a stagnation at the trivial all-ones network. Finally possible reasons for this stagnation are discussed.**

1 Introduction

A problem with having computers play turn-based games like chess, checkers, mill etc. is that the games are so complex that it is not possible to calculate a complete tree of all possible game states. If we could do that we could always deterministically look up the best possible move for the computer, given any game situation, which would lead it towards a win situation. Since this is impossible, an approach is to build a part of such a tree, looking a number of steps ahead from a given state. For the computer to be able to choose which move to make it must be able to assign a quality to each possible game state. This quality measure can either be calculated using certain hard-coded heuristics about how the game is best played - or we can try to let the computer figure out those heuristics by itself.

Inspired by the work in [2] and [1] in which they used an EA for finding these heuristics for the game checkers we wanted to test if a similar approach could be used for playing mill.

2 Abstract model

In this section we introduce the game of mill and give a short high-level description of our implementation of the game mechanics. We then move on to describe the way we have made a parameterized computer-player and finally we describe how we use an EA to try to find a good parameter setting for such a player.

2.1 The game of mill

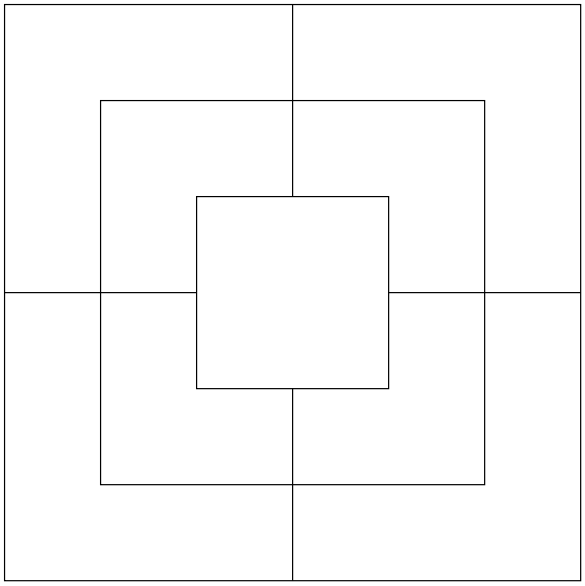


Figure 1: The mill board.

In 1 the mill board is depicted - the 24 places where lines meet is where pieces can be placed. Each player starts with 9 pieces and the game proceeds turnwise in 3 stages:

1. Insertion of pieces. The players selects any empty place on the board and insert one of their remaining pieces. Pieces already on the board cannot be moved.

2. Moving of pieces. The players select one of their own pieces and moves it to a non-occupied neighbour place along the lines of the board.

3. Jumping of pieces. Once a player has only 3 pieces left he can move them not only to neighbour places but to any available place on the board.

During any of the stages a player gets a mill if his pieces occupy all three places of a line. When a player gets a mill he is allowed to remove any one of the opponents pieces from the board. The game ends when a player is unable

to make a mill i.e. when he has less than three pieces left. A player can also lose if he cannot make a move.

We have made an implementation of these rules where we use an array of size 24 to represent the board-state. An entry with -1 in this array represents a place occupied by an enemy piece, whereas 0 means empty and +1 means occupied by one of our pieces. We then have a method that given a board-state A calculates the set of board-states that can follow A after a single legal move. When the computer has to move it will first calculate this set and then select the best move from it.

2.2 A computerized player

We used a standard feedforward fully-connected neural network to evaluate a boardstate. In our first implementation the input layer had 24 neurons connected directly to the array representing the board (as described above). The output-layer always has a single neuron and we use its output directly as the quality of the boardstate. It is possible to use such a neural network to decide between all the legal moves from the current state. However this would require the neural network to be very good at judging the quality of a board and therefore we help it by looking a certain number of moves ahead. This is done by generating all the possible states that can be reached from the current in *lookahead depth* number of steps. We can then use a standard *mini-max* algorithm [3](pp. 197-202) to find the best move.

We tried this implementation with a simple handcoded neural network (*allOnes* - described below) and found that it was very time-consuming even with a lookahead depth of only 5. Instead of the minimax algorithm we then implemented an *alpha-beta* algorithm [3](pp. 202-208). This algorithm is a lot faster and we can now use a lookahead depth of 7 when playing against it (when evolving we typically used a value of 5). There are a lot of even better algorithms but implementing these is future work.

2.3 Finding a good player

The player described above is parameterized by the weights of its neural network. We can therefore see such a player as an individual where the weights is the genes. We have constructed a standard evolutionary algorithm with these configurations:

- Crossover is random weighted arithmetic crossover between two parents.
- Mutation is a run over all the weights where there is a probability p that we add a number. This number is sampled every time from a random gaussian distribution with mean 0 and deviation d. p and d are parameters to the EA.
- Selection is tournament selection where we have tried both 2 and 4 competitors.

We have also constructed a simple hill-climber algorithm as described below.

Since there is no rating scheme in mill (like in chess f.ex.) we were unable to come up with anything that could describe the fitness of an individual. The only thing we can do is to compare the fitness of two individuals by letting them play against each other. This introduces some problems in our search for a good player. First it makes it difficult to tell if a search is making progress at all and secondly the seemingly simple task of finding the best individual in a generation requires that we let all players play against all the other players. Since this would require playing $n * (n - 1)$ games (where n is the population size) it is not feasible. We therefore play a tournament where we in each iteration half the number of contestants by letting number one and number two play and letting the winner proceed to next iteration, then number three and number four and so on. When there is only one player left we stop the iteration and selects this player as the best individual. This requires only $n * log_2(n)$ games and ensures that the winner can beat all others through a string of length $log_2(n)$.

2.4 GUI

To easily see how a network performs we made a program that allowed us to play against any of the evolved players. It has a graphical display and mouse input as seen in figure 2. This program is written in Java and can be downloaded from:
`www.daimi.au.dk/~carsten/Mill.tar.gz`.

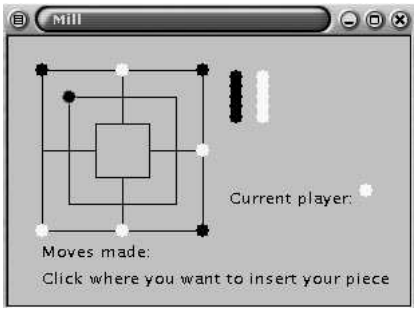


Figure 2: GUI.

3 Experimental setup

In our experiments we had problems with not being able to see in reasonable time if our changes had any effect, and if they were bugfree, thus we were forced to make a compromise between lookahead and population size. We judged that a high lookahead was more important and thus we chose to have the small population size of only 25 individuals and maintain it at that level, while using all gains from speed optimizations to increase the lookahead size.

We have mostly had the same crossover and mutation rate all the way through the experiments, while we have tried a few different ways of controlling mutation power - both with a fixed power and a decreasing power of the type
$initialpower/generation$. In the hillclimber discussed below we even tried to have a fixed power for x generations and then slowly decrease it like described above.

As written above, we did our evolution with a population size of 25 but we also tried to make a kind of hillclimber by reducing the population to just one individual, stopping crossover (dosn't make sense with just one individual) and instead we spawn a copy, mutate it, let copy and original compete and keep winner. The idea is that a mutation will give us a neighbour to the current player and by letting the two compete we can decide if we are approaching a local optima or looking in the wrong direction.

We also tried to use a few different neural networks for both the evolution and the hillclimber:

- The first neural network we made had three layers of size 24-8-1, and just used the 24 different board positions as inputs. After tests we came to believe that this was too simple though.

- To compensate for this we made the second network with four layers (size 24-24-10-1). But when this gave us much slower computations, the net were too large, and not better results we thought about using a different strategy.

- We did stay with four layers (size 36-24-10-1) in our third neural network and we even added 12 extra inputs (sums of the 12 possible millpositions on the board) in the hope that this could give the extra it needed to develope a good player. However computation time went up with the extra inputs and the evaluations became unacceptable slow.

- Thus in our fourth network we retained the extra input but cut the network down to three layers again (size 36-36-1) hoping that the reduced size would give a boost to speed while still retaining the complexity to express how to evaluate a mill board.

As a final note: We wrote the best player to disk every fifth generation when evolving while the hillclimber wrote the survivor to disk every generation. This allowed us to keep a history of how the networks evolved, and compare them over time to see if they got better.

4 Results

Through the numerous evolutionary and hill climbing runs, the results have not been satisfying. Firstly the players do not get strictly better. Usually, if you let subsequent generations compete against each other, the results do not show the later generations strictly better than the earlier. It can even appear to be random at times. This was what led to our implementation of tournement selection of size 4, in an attempt to increase the selection pressure. The increased selection pressure did not make a difference though, and we suspect that it is our "findBest" function, which tries to find

and save the best network in the population, that is inadequate, since this would show the same kind of behavior.

Secondly the evolved networks have never really been able to match the trivial neural network with all weights set to one. This might seem fatal, but one must keep in mind that this trivial network, based on our specific implementation of the neural network, in essence evaluates a board from the number of pieces on each side only. Thus the trivial network simply tries to take out as many opponent pieces as possible, while retaining as many of its own pieces as possible [15].

This allOnesPlayer with a decent ply is in fact not a bad player. The authors of this article had quite a difficult time playing against it, except in the case where the allOnes is put into a position where it can no longer move very early in the game. But how well would it fare against more experienced players? Although there might be some online gaming site running Mill, the allOnes has some fatal weaknesses. These include the before mentioned early strangulation, but minimax and alpha-beta also only evaluates the leaf board states in the search tree, and thus cannot differentiate between quick and slow improvements within the ply. This actually applies to any network running on our system, and not just allOnes. A human player is quick to seize the opportunities that arise from the mentioned weaknesses, making it hard to see what the network is good at.

The fact of the matter is that allOnesPlayer plays well, but there simply must be ways to improve the network to avoid the network's weaknesses, simply by altering the network weights. Some improvements one could imagine would be:

- Reckognition of "near-mill" constellations.
- Penalizing low movability.

Why did our evolution and hill climbing have such a hard time competing with this allOnesPlayer? It ought to be able to evolve the allOnes behavior, and possibly improve on it. It is notable that in the case where we evolved from an initial population of allOnesPlayers, the evolution had a very hard time improving the networks. We never really achieved a better player, only players able to do slightly better than allOnes when it was making the initial move.

Although our evolved networks could not really compete with allOnes, we tried playing them ourselves. They did play fairly well in the sense that they often blocked our attempts to make mills, and were able to make mills themselves. It was however clear that the networks had many "blind spots" on the board, where they were unable to see an upcoming opponent mill or make one themselves. These networks were around generation 250 evolved from an initally random population with the simplest of the neural networks. We are confident that they could play even better and approach allOnes if they were given more time to evolve. The question is if they could have evolved *past* allOnes. This is not so certain since an evolution initialized with allOnes players did not fare notably better.

Unfortunately we never really settled on an implementation and configuration of the system, and bugs in the code required restarting the evolution many times. Because of this coupled with a shortage of time, we never had an evolution run for several weeks as Fogel did (see [1] and [2]).

Also the implementation of the alpha-beta algorithm came too late in the project to really impact the end result.

As mentioned in section 3, we have tried giving the network some spatial information by including the sum of pieces for each line of mill, but without notable success. With the current implementation of the neural networks, this spatial information can be described in the network itself, without the need for extra input. We had hoped however, that it would speed the progress. It mostly just seemed to slow the evolution though because of the increased network complexity.

We also would have liked to try out other kinds of neural network implementations, like a sigmoid neuron behavior and more complex

[15] This is why we have also implemented what we call the simplePlayer, which is simply a hand coded version of the allOnesPlayer running a lot faster because it doesn't need to evaluate a neural network.

layer connections, but did not have the time.

5 Conclusions

When we started this project we hoped to end up with an evolved network which could play mill on an expert level. What we eventually ended up with was a program which actually plays fairly well - but it does not beat the simple network with all weights initialized to 1. This is of course not a very satisfying result but it raises the question about how much better can a neural network give a quality to a board than simply following the 'greedy' strategy of the all-ones player - namely to always follow the path which leads to the removal of one of the opponents pieces. There certainly are other aspects to playing mill and this shows when a human player plays against the network. The computer could for example benefit from being able to plan for a strategy and follow it for even more moves that it uses in its look-ahead. Also certain board states that the all-ones network judge to be of equal quality because the number of pieces on them is the same has other more subtle aspects to them which makes a human player judge them differently. Unfortunately none of these enhancements were found in our evolution and it remains an open question if they can be represented by a neural network at all.

It is possible that a better mill player can be evolved using Fogels method with a different setup than the one we used but we think we did try out many different ideas and none of them gave any results at all.

References

[1] Kumar Chellapilla and David B. Fogel. Anaconda defeats hoyle 6-0: A case study competing an evolved checkers program against commercially available software. In *Proc. of the 2000 Congress on Evolutionary Computation*, pages 857–863, Piscataway, NJ, 2000. IEEE Service Center.

[2] David B. Fogel. Evolving a checkers player without relying on human expertise. *Intellignece*, 11(2):20–27, 2000. ACM Press.

[3] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers, Inc., San Fransisco, California, 1998.

Effect of dimensionality in the Diffusion model

Niels C. Bach and Roar Kjær-Larsen

Abstract— **Abstract. A common method of improving evolutionary algorithms involves a sort of structure, in which the position of the individuals restrict their free interaction with other individuals. The diffusion model is such an example with some fixed interaction range in the structure. Very often this structure or landscape is two dimensional. We question the rationality in this choice. A diffusion model is developed to handle any dimension of the involved landscape. Extensive test have been performed comparing the traditional 2D model with a 1D model. The full functionality seems to be preserved with this simpler approach. Test have been performed to examine the effects of increasing the number of dimensions. No special effects are observed, only a gradual transition into a standard EA. Test involving changing the interaction distance in the 1D model has been performed. The same gradual change is observed here. A measure of the "complexity" of a given structure is developed and shown to facilitate comparison between different models. We conclude that for the used benchmark problems the 1D model is equivalent with the other topologies and offer a much simpler algorithm..**

1 Introduction

Several techniques for improving the standard EA have something in common; they introduce a sort of landscape in which the individuals of the EA live. The goal of this is to eliminate premature convergence and maintaining diversity.

The diffusion model for instance, though inspired by the structure of parallel computers, resembles animals living over a wide area. Close proximity is required to be able to interact directly - without actually having more than one population.

Most of these examples feature two dimensional landscapes without any explanation for this choice.

Choosing the diffusion model as example we aim to implement this EA with its traditional square grid as landscape. We will try to adopt some method for generalising the EA to use any number of dimension for the grid.

This should allow us to examine several interesting points. First of all, is it really worth going through all the trouble of creating a 2 dimensional structure at all? Can a much simpler one dimensional model compete with the original? Next, what possible benefits can come from adding more dimensions to the landscape.

Further it would be interesting to possibly watch the diffusion model gradually degenerate into a standard EA when the dimension of the landscape gets high enough and all individuals are direct neighbours.

2 Model

We have, for a start, based our algorithm on a standard EA; thus repeating the cycle of evaluation, selection, crossover and mutation. Representation is by real value encoding, and initialisation with uniformly distributed random vectors.

The genome of the best individual for any generation is saved and inserted into the next generation at the same position. This elitism ensures that the best solution is not lost.

```
N-dim Diffusion {
  t = 0
  initialise population P(0)
  while (not termination-condition) {
    evaluate population P(t)
    save best individual in P(t)
    select P'(t) from P(t)
           through tournament
    create P"(t) from P'(t) with
           arithmetic crossover
    create P(t+1) from P"(t) with
           Gaussian annealing mutation
    insert saved individual in P(t+1)
    t = t +1
  }
}
```

The probability p(t) decides whether there will be selection (or plain copying) for any given position. Selection is by tournament between two random individuals and the "source" population is kept fixed by copying the results into a separate array.

```
Tournament {
  for all individuals I in P(t) {
    with probability p(t) {
      pick two individuals from I's
            neighbourhood in P(t)
      compare fitness of these two
            individuals
      insert copy of fitter individual
            in P'(t)
    } else {
      copy I from P(t) to P'(t)
    }
  }
}
```

The probability p(c) decides whether there will be crossover (or plain copying) for any given genome. Crossover is arithmetic between two random individuals with a separate uniformly distributed weight for each genome.The "source" population is kept fixed and separate from the newly calculated individuals.

```
Crossover {
  for all individuals I in P'(t) {
    pick two parents from I's
            neighbourhood in P'(t)
    for all genomes G in I {
      with probability p(c) {
        create randomly weighted mean
            from parents G's
        insert mean as genome in I in
            P"(t)
      } else {
        copy G from P'(t) to P"(t)
      }
    }
  }
}
```

The probability p(m) decides whether there will be mutation (or plain copying) for any given genome. Mutation is by addition of a random normal distributed number. The variance of the normal distribution starts out with the value σ^2 and is multiplied with the factor $1/(t+1)$, thus decreasing for each generation. (annealing)

```
Mutation {
  scale mutation variance W with 1/(t+1)
  for all individuals I in P"(t) {
    for all genomes G in I {
      with probability p(m) {
        add random number from
            N(0,W) to G
      }
      insert G as genome in I in P(t+1)
    }
  }
}
```

The algorithm has after this been modified into a diffusion model which pictures its individuals arranged in a rectangular two dimensional grid. To avoid anomalies the grid wraps around in all four directions being equivalent to a torus.

```
  wrap
^ ^ ^ ^ ^
| | | | |
O-O-O-O-O->
| | | | |
O-O-O-O-O->
| | | | |
O-O-O-O-O-> wrap
| | | | |
O-O-O-O-O->
| | | | |
O-O-O-O-O->
```

The neighbourhood of an individual is defined as those individuals within a specified range in the grid. If the range is 1 the neighbourhood consist of the individual itself (I) and the eight adjacent individuals (N).

```
O-O-O-O-O
| | | | |
O-N-N-N-O
| | | | |
O-N-I-N-O
| | | | |
O-N-N-N-O
| | | | |
O-O-O-O-O
```

Any individual is restricted to interaction with individuals in it's own neighbourhood. Actually we have restricted tournament and parent selection for a given position in the grid to the neighbourhood of this position.

This model can be generalised to one, three or more dimensions of the grid. One dimension yields a ring structure and three or more a hyper-torus. The neighbourhood in d dimensions is calculated as:

```
(x'(1),x'(2)...x'(d)) = (x(0)+r(0),x(1)
                  +r(1)...x(d)+r(d))
```

where r(i) is discrete uniformly distributed on [-R:R] and R is the interaction range. Wrap around by restricting each parameter x'(i) to the grid size by modulus.

If we assume this d-dimensional grid is of the same size in all directions and we picture its individuals as one linear adressed array the index I' of the neighbourhood of I is:

```
standard EA:    I'=[0:N-1]
1-D diffusion:  I'=(I+r(0)) mod N
2-D diffusion:  I'= (I mod sqrt(N)
                    +r(0))mod sqrt(N) +
((I div sqrt(N)+r(1))mod sqrt(N))*sqrt(N)
3-D diffusion:  I'= ... arrgh!
```

where N is the population-size and mod & div the modulus and integer division operators. Not very nice.

Picture the 2D situation with N=9

```
1-2-3
| | |
4-5-6
| | |
7-8-9
```

wrap around means horizontally that for instance 6 and 4 connect. If we instead connect 6 to 7 and likewise for the rest, things become easier, corresponding to a torus twisted slightly at its "seam".

```
standard EA:    I'=[0:N-1]
1-D diffusion:  I'=(I+r(0)) mod N
2-D diffusion:  I'=(I+r(0)+r(1)*sqrt(N))
                    mod N
3-D diffusion:  I'=(I+r(0)+r(1)*qbrt(N)
                  +r(2)*qbrt(N)^2) mod N
.
.
d-D diffusion:  I'=(I+SUM(i=0 to d-1)
                   r(i)*N^(i/d)) mod N
```

This last formulae works even for non-hypercubic population sizes:

$N \neq k^d$

where k is some integer and allows to compare a fixed populationsize under several topologies with different dimensions. Otherwise N would have to be a "magic number" like 2^6, 2^{12}, 2^{60} for $1-3$, $1-4$ and $1-5$ dimension respectively.

This slight change of the true hyper-torus is assumed to have no side-effect.

```
    25-26-27    Popsize 3^3 = 27
   /  /  /
  22-23-24      from center (14)
 /  /  /
19-20-21        - right: +1 = +27^(0/3)
                - left:  -1 = -27^(0/3)
    16-17-18
   /  /  /      - in:    +3 = +27^(1/3)
  13-14-15      - out:   -3 = -27^(1/3)
 /  /  /
10-11-12        - up:    +9 = +27^(2/3)
                - down:  -9 = -27^(2/3)
    07-08-09
   /  /  /                     N^(i/d)
  04-05-06
 /  /  /        mod 27 =>
01-02-03        twisted wrap left/right
```

3 Implementation

Through out the whole project we have maintained two different implementations. One in JAVA reflecting a true representation of the multidimensional diffusion model grid and one in C with a one dimensional array and a neighbourhood function. Comparing the outputs from both models has helped greatly to identify errors in the implementation. In despite of their very different nature they have been able to produce the same results within statistical errors.

We have performed other extensive test of the two systems to weed out mainly conceptual error possibly common for both implementations. The JAVA implementation was especially easy to test component after component while building it. Most important have been comparing performance for the 2d and std. EA settings with known good results, from especially the RBEA paper. It has been very useful to manipulate the standard problems to attain their minimums somewhere outside the classic value at origo.

For the C implementation we have addressed a widespread problem. The random number generator guaranteed by ANSI C has several flaws. Especially important in our case is the correlation in k-space (important when initialising a multidimensional problem) and non-random behaviour of the low order bits (using mod to get small integers). An alternative random number generator and conversion to Gaussian distribution was copied directly in C from chapter 7 of "Numerical recipes in C". The code is based on a multiplicative congruential generator:

```
I(j+1) = (16807*I(j) mod 2147483647)
```

with Schrage's algorithm for avoiding 64 bit work. A shuffling procedure is added afterwards. Except for ran1 & expdev the code is our own.

4 Experiments

As mentioned earlier we have assumed that the trick of using a modified wrap around system for the grid wouldn't influence the results. By direct comparison of results from the JAVA and C implementations this has been verified. The C implementation uses the twisted torus and the JAVA implementation the real torus due to its real multidimensional structure.

4.1 Setup

The presented graphs show fitness of best individual in each generation and some the mean fitness of the whole population in each generation. These figures are averages calculated from 30 consecutive or simultaneous runs. All runs have been executed with a population size of 400 individuals.

Each graph represents a hole series of experiments. We have started each with an automated search of the parameter space to obtain an overview of the feasible combinations of p(t), p(c), p(m) and sigma. This has been followed by an extensive hand optimisation where we have searched mostly for typical instead of optimal results.

The used parameters are listed with each graph together with the type of algorithms used, their dimensionality and neighbourhood size. Where this size is omitted a default of 9 has been used.

4.2 Test functions

Schaffer F6:
$f(x,y) = \frac{sin^2(\sqrt{x^2+y^2})-.5}{(1+0.001(x^2+y^2))^2}$
$-100 <= x,y <= 100$

Rastrigin F1 (20d):
$f(x') = \sum_{i=1}^{20} {x_i}^2 - 10cos(2\pi x_i)$
$-5.12 <= x_i <= 5.12$
Griewank's function:
$f(x') = \frac{1}{4000}\sum_{i=1}^{10}(x_i - 100)^2 - \prod_{i=1}^{10} cos(\frac{x_i - 100}{\sqrt{i}})$
$-600 <= x_i <= 600$

De Jongs F2:
$f(x,y) = 100(x^2 - y)^2 + (1-x)^2$
$-2.048 <= x,y <= 2.048$

De Jongs F4:
$f(x') = \sum_{i=1}^{30} {x_i}^4$
$-1.28 <= x_i <= 1.28$

4.3 Standard EA, 1D diffusion and 2D diffusion

The first series of experiments are supposed to analyse whether a 1 dimensional diffusion model with a neighbourhood of 9 is capable of producing results similar to the 2 dimensional diffusion model with the same neighbourhood size. The standard EA is displayed alongside to indicate whether the special abilities of the diffusion models are actually used. Rastrigin F1 with high selection, huge mutation and no mating is an example of unwanted settings indicated by the superiority of the standard EA (random search).

Rastrigin F1, De Jong F4 and Griewank show similar pictures; the standard EA is much quicker than the others but looses diversity and stagnates after relatively few generations. The 2D does better but still stagnates in a similar way where as the 1D retains more diversity and does not stagnate or does so on a very low level.

In both Schaeffer F6 and De Jong F2 the curves of the mean are off scale. But even with enough diversity for everyone the 1D outperforms the two others.

At least for these test functions it is clear that the 1D diffusion model can perform just as well as the 2D can. It is also seen that the same neighbourhood size for the two different topologies does not mean that they perform alike.

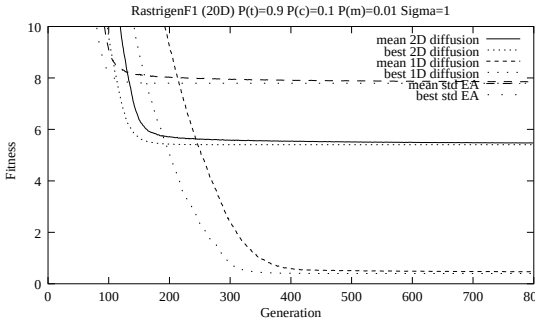


Figure 1:

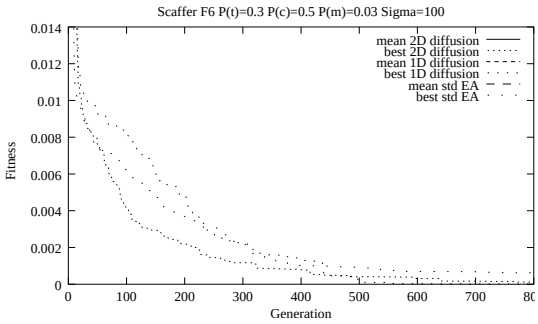


Figure 2:

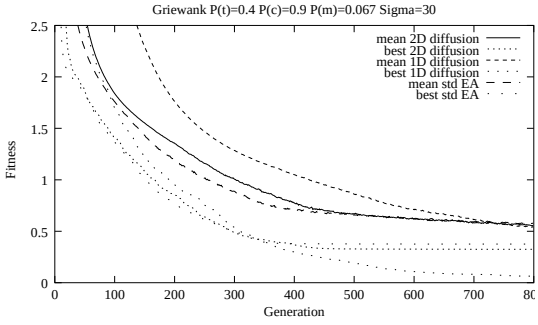


Figure 3:

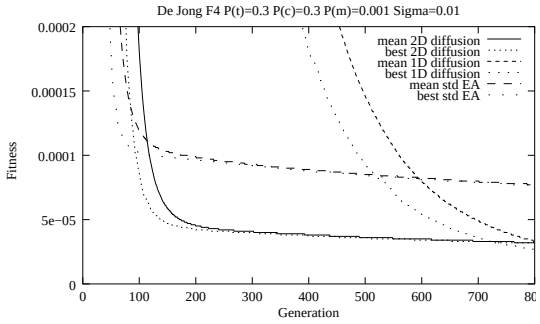


Figure 4:

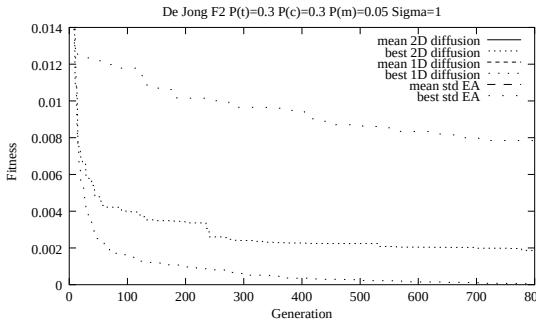


Figure 5:

4.4 Diffusion model with different dimension

In this experiment we increase the number of dimensions of the diffusion model while keeping a interaction range of 1. This means that the neighbourhood size increases as 3^d. When six dimensions are reached the neighbourhood size calculates to be bigger than the populationsize. This means that all individuals are neighbours and that there are several ways to reach some of the other individuals resulting in non-uniform selection. The 6D ought to perform like an standard EA.

Rastrigin F1 shows that 1D performs OK and 2D stagnates a bit below 6. This last result is identical to the one one Figure 1. Comparing the same graphs for the 1D shows a difference because of the different neighbourhood size (3 and 9). 3D to 6D all stagnate around 8 after approx 100 generations. This is identical to the performance of the std EA on figure 1.

De Jong F2 shows a similar picture. 1D performs excellent, even better than the 9 neighbourhood 1D from figure 5. The 2D does well too but surprisingly a bit worse than the identical 2D from figure 5. The algorithm is the same but the implementations differ widely, so we have performed several comparison test. The difference stems from a high variation in the performance even when averaged over 30 runs. The 3D to 6D show performance close to the std EA on figure 5.

The general picture from the two graphs above repeats itself with the Schaffer F6. The curves correspond with those on figure 2. almost as clear as before. The 1D seems to benefit more with the 9 neighbourhood size here than with the 3. This is problem dependent at least.

It is clear that the different implementations of the 2D model act the same, just as expected. The gradual change from restrictive 1D model to 6D equivalent to the std EA is very easy to observe. Whether there is no real difference between the 4-5-6D models or it is a question of wrong test functions is not yet known.

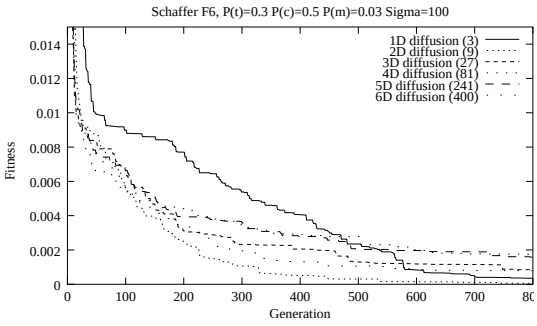


Figure 6:

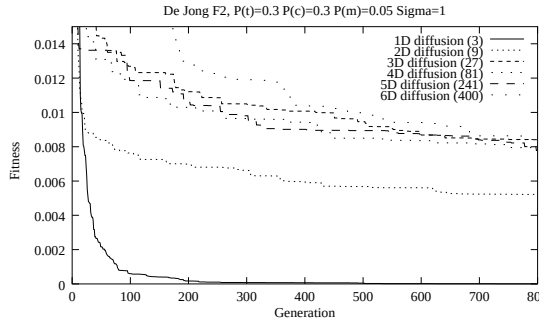


Figure 7:

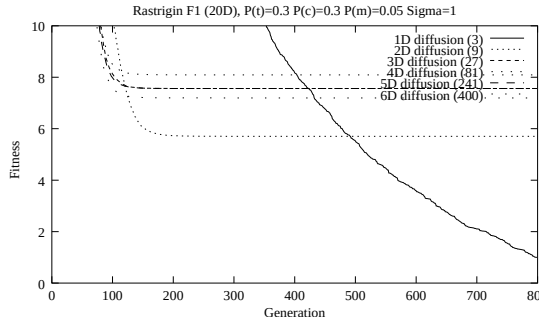


Figure 8:

4.5 Neighborhood size

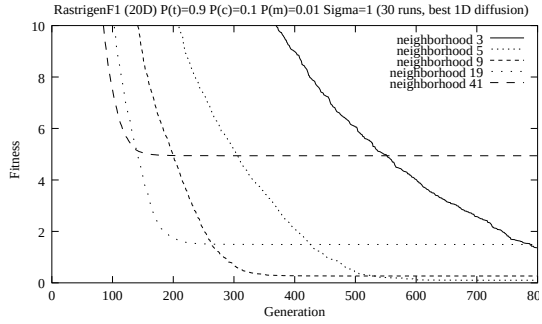


Figure 9:

The last experiment is similar to the one before. Again we want to change the diffusion model gradually from most restrictive mode to a mode similar to the standard EA. As we have not seen any special improvement of the diffusion model by adding more dimension to the grid it is tempting to ask whether the 1D model may not replace all these multi dimensional topologies altogether.

We can use the interaction range in the 1D diffusion model as our parameter and thus regulating the neighbourhood size. By watching the very gradually changing curves carefully and picking a few interesting values of the interaction range we got figure 9. Starting with a neighbourhood of 3 we see that it is identical to the one in figure 8 which is not surprising. A neighbourhood of 5 seems close to the optimal although it does not reach 0 every time. The neighbourhood of 9 is of course identical to the result for 1D on figure 1. At 19 the results start to get unacceptable. The algorithm is fooled by the local minima at 1 too often resulting in the high average stagnation value. At very high values we again get the std EA from figure 1. (last not on graph)

The 41 neighbourhood size line is quite interesting too. It corresponds nicely to the 2D with 9 neighbourhood size. If we calculate the minimal number of "jumps" (along allowed connections) we will need to get from one individual to any other individual we get something interesting. This is the minimal number of generations for some information to spread from one point to the rest of the grid. Let us call this the "complexity" of the topology.

The dimension, neighbourhood and complexity for the topologies with graphs for Rastrigin F1 are shown in table 1.

Notice that this "complexity" corresponds closely to the performance of the topology. The two identical graphs for 2D-9N and 1D-41N mentioned above have the same "complexity". Further test have confirmed these results.

We can conclude that the 1D diffusion model with variable interaction distance is more than sufficient to produce all the seen results. It is even possible to regulate the restrictiveness of the topology finer than the multidimensional model allows. We also have a nice measure for the "complexity" of the topology.

An interesting idea is to incorporate the inter-

Dimension	Neighbourhood	complexity
1D	9N	50
2D	9N	10
1D	3N	200
1D	5N	100
1D	19N	20
1D	41N	10
3D	27N	7
4D	81N	4
5D	243N	2
6D	400N	1
std EA		1

Table 1:

action range into the genome of the individuals and have this setting regulated by the ea. The fact that an individual can only increase its influx of foreign genome material with this parameter and NOT to export it's own genome more effectively with it should prevent a unstable evolutionary behaviour of the individuals.

5 Discussion

It is evident from the listed results that neither the dimensionality nor the neighbourhood size alone is enough to categorise the effect of the chosen topology. If we define the minimal number of "steps", we have to trace trough the topology to be able to reach any one point from any other, as the "complexity" of the topology, we can see a distinct correlation with the effect on the algorithms behaviour.

The higher the "complexity" number the more it tends to preserve diversity, the closer to 1 the more it resembles a standard EA. Two widely different topologies with same "complexity" number are shown to have identical behaviour.

We have until now focused on the direct dimensionality of the diffusion model. A bit more general we can say that we have N individuals that can communicate freely in the standard EA and that we restrict this by only allowing certain connections in the diffusion model. The topology of these connections can be whatever we can imagine, not just those motivated by the picture of some hyper-torus.

If a simple topology can achieve the same results as a more complicated one the simple is of course the one too choose. It's hard to imagine a simpler one than the one dimensional array connected as a circle. Either with just connections between immediate neighbours or with some interaction distance.

```
range=1:-O-O-O-O-O-O-
```

```
          _   _   _
         / \ / \ / \
range=2:-O-O-O-O-O-O-O-
        _/ \_/ \_/
```

We have not been able to document any beneficial effect of the more complicated models - which doesn't mean that there is none or that some other complicated scheme might work. And we have seen the simple circular model to produce just as nice results as the others. This together with the easy and precise regulation of the topology's so called complexity makes us confident that it's a wise choice.

6 Problems

In both implementations of the crossover operator there is a small bug. Two random parents are chosen from the neighbourhood. Each genome is replaced with either a weighted mean between the two parents or the genome of the individual in the SAME position in the previous generation. Thus it is likely that the new genome is a hybrid mix of THREE other individuals. Not what we had in mind. We should have used one of the parents for the copying part. Unfortunately this changes the behaviour of the hole algorithm for a lot of the already finished experiments and graphs. We choose not to correct the error.

Evolutionary Speaker Classification

Troels Grosbøll-Poulsen, Mads U. Kristoffersen, and Morten Bek

Abstract— **An evolutionary model for classification of utterances is proposed. In this model feature vectors are extracted from the speech signal. A set of proposed measures are calculated from the feature vectors and these are used as input for training a Kohonen network. The task of the evolutionary algorithm is to optimise a set of weights for the distance measure used in the Kohonen network. The network is evaluated on it's ability to distinguish sets of training data. The set of weights that gives the best fitness are expected to show the relative importance of the measures with regard to distinguishing the different types of input. Having performed several experiments for fine tuning the EA and testing the quality of the results, we conclude that although the EA is able to find extremely good separations of the input, this might be due to the small amount of input data compared to our chromosome length and the size of the Kohonen network. On the other hand the results also show fast convergence and a stable result in the less involved experiments. We conclude that coupling EAs with Kohonen networks for automatic optimising of data classification problems is indeed a useful tool that however needs some refinement.**

1 Introduction

The project described in this paper attacks the problem of partitioning a set of human utterances according to one or more predefined criteria. The perhaps biggest problem in obtaining this goal, is that quite often one has no a priori knowledge of a set of parameters discriminating one utterance from another according to a given criterion. For example, in determining whether an utterance originates from a male or a female speaker one intuitively suspects the pitch to be of significant importance. In the general problem of discriminating utterances of one voice from another, however, it is hard to estimate which features are of importance. Furthermore, the weighting of one feature as opposed to another is not evident either.

Historically, much effort has been put into the task of speaker identification, a special case of the general problem formulated above, mainly based traditional techniques such as Hidden Markov Models, template matching based on Dynamic Time Warping, neural networks, etc. (see [1], [6])

In this paper we investigate another approach, describing a system which uses an evolutionary algorithm to optimise an "utterance distance measure", a weighted euclidian distance measure operating on vectors of so called feature measures that are calculated on sets of feature vectors extracted from a set of training utterances. The weighting is evolved according to the set of criteria one wishes to classify according to and can afterwards be used in building a Kohonen Network (as described in [4] and below), a self organising map, that performs the actual classification on arbitrary utterances.

In the sections below, we describe the model underlying our system, the experiments conducted and the results obtained.

2 System Description

An overall view of the system can be seen in figure 1. The input to the system consists of the following:

- A set of training utterances. These are sampled waveforms of speech[16] on which no particular restrictions are imposed, apart from the fact that they should represent as great a diversity as possible with respect to the criteria one wishes to classify according to. In the experiments described later, however, some preparatory work was done in order to enhance the performance of the system (see section 3.2).

[16] From now on, a sampled utterance is referred to as a speech signal or simply a signal.

- The criteria one wishes to classify according to, along with the classification of each training utterance. For example, if one wishes to partition male and female speakers, the system must be supplied with the criterion "male/female" and each training utterance must be tagged as originating from either a male or a female speaker.

- A set of feature measures each of which expresses some facet of an utterance. Feature measures are described in detail below.

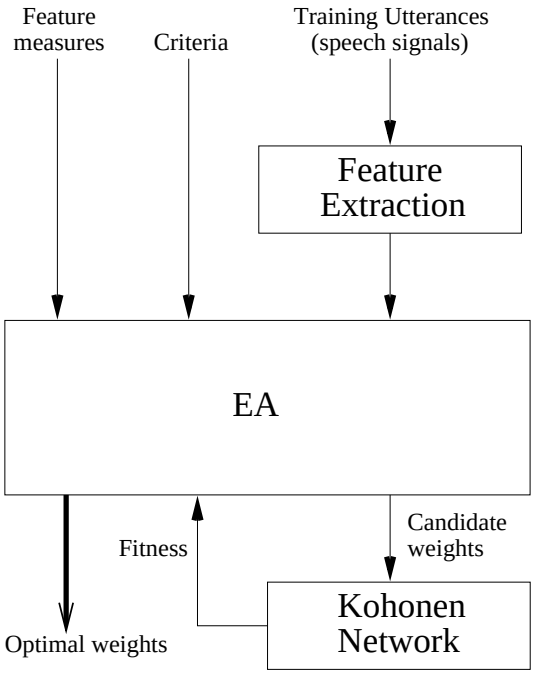


Figure 1: System overview

The overall goal of the system is to evolve a vector w of weights such that the *Classification Measure*, defined as:

$$CM(S_i, S_j) = dist(w \times FM(S_i), w \times FM(S_j)),$$

can be used as a distance measure in training a Kohonen Network, such that the classification thus performed is optimal with respect to the given criteria.

In the formula above, FM designates a set of *feature measures*, each representing some property of an utterance that can be calculated from the speech signal, or a transformation thereof (i.e. mean pitch, energy variance etc.), thus $FM(S_i)$ refers to a vector where each entry contains the result of evaluating a feature measure on signal S_i. dist denotes euclidian distance, while $\times$ denotes pairwise multiplication of vector entries.

The task of finding w is solved by an EA, where each individual represents a candidate vector. To evaluate a candidate, a Kohonen Network using that candidate in its distance measure is trained with the training utterances. From this process a fitness can then be calculated. In the end, the candidate receiving the best fitness constitutes the output of the system.

Below, the key components of the system are described.

2.1 Feature extraction

Since a speech signal consists of an impractically large amount of data with the characteristics of the speaker not directly "visible", some means of extracting this information must be chosen. One possibility is to calculate the power spectrum [3](or perhaps the slightly more complicated *cepstrum*) of the signal, using the Fast Fourier Transform algorithm. While this certainly reveals a great deal of information about the speech signal, it does nothing to reduce the amount of data to be handled, since the size of the power spectrum for a signal of length N is itself N. Instead we employ a different technique (a quite common approach), using a speech signal compression algorithm to reduce the amount data. A "fortunate" by-product of this is that with the algorithm chosen, the compressed data directly expresses a set of features of the particular signal. Applying a number of feature measures to such a feature set, we then (can hope to) obtain a set of numbers characterising the speaker of each utterance.

The actual algorithm chosen is Linear Predictive Coding and in particular the US Department of Defence version (see [5]) called LPC10-E. We briefly describe the concepts of this algorithm here. A more comprehensive description can be found in [2].

The general objective in (this form of) linear prediction is to express the i'th sample point of a signal as a linear combination of the M previous sample points and a residue called the *error*:

$$S(i) = \sum_{k=1}^{M} a_k S(i - (M - k)) + e(i)$$

The task is thus to determine the coefficients a_k such that the error is minimised. At the bottom line, this amounts to solving a special system of linear equations, but we shall not delve into the details of that here - the interested reader is referred to e.g. [2].

A common approach used when modelling the human sound production system (see [6] and [1]), is to regard the vocal tract as a series of uniform cylinders with varying cross-sectional areas. This is called the acoustic tube model. An interesting property of LPC is that the M coefficients found above can be shown to be equivalent to the so-called reflection coefficients which in turn define the area ratios of the cylinders in the acoustic tube model. The coefficients thus represents a description of the state of the vocal tract at a particular point in the signal.

In the case of LPC10-E, a signal is processed in chunks (*frames*) of length about 22ms (this number is chosen based on the rate of change of the vocal tract during speech). For each frame, 10 LPC coefficients are calculated (i.e. $M = 10$ in the formula above). On top of this, LPC10-E also provides estimates of the voicing, pitch (if applicable) and the energy level of the frame as a whole.

Altogether, a feature set for a particular utterance in our system can be described as a set of 14-dimensional feature vectors containing the energy, pitch and 10 numbers describing the shape of the vocal tract.

2.2 Feature measures used

As mentioned above, the feature measures are the components that make up the CM. The feature measures naturally plays a great role with respect to what can actually be classified with the system and the search for good feature measures is a quest in itself. We have chosen to equip the system with a set of standard measures and our hypothesis is that a weighted sum of these measures are adequate for a large number of classification tasks.

The feature measures used in our system consists of calculating the mean, variance, minimum, maximum and difference between maximum and minimum for each entry in the feature vectors of an utterance. This leads to a total of 60 feature measures.

2.3 EA configuration and EA operators

The EA used is based on a standard GA using real encoding. This was chosen for convenience, since we are optimising a real valued vector. As mentioned before, each individual represents a candidate for the weight vector to be determined. Specifically, an individual consists of one chromosome which is a vector of the weights to be applied to the feature measures.

The selection scheme employed is a combination of elitism and tournament selection with a tournament size of two. During the selection phase, an elite consisting of the k fittest individuals, where k is a constant fraction of the population size, are automatically selected. Next, individuals are selected using tournament selection until a total number of individuals equal to the population size has been selected. Finally, the selected individuals are allowed to mate at random, creating offspring which is subject to crossover and mutation with constant probabilities. Notice that the elite is copied directly to the next generation and is *not* subject to mutation.

The crossover operator used is a simple 2-point crossover, with the 2 points selected at random for each crossover made. This type of crossover was selected because initial experiments showed that with a standard arithmetic crossover the EA had difficulties pulling irrelevant genes towards zero (an important property of our system — we want the system to "pick" the feature measures needed to perform the classification task). Moreover, in this way we also hope to avoid stagnation due to the "averaging" effect which results from using e.g. the standard arithmetic crossover.

With respect to mutation, the following scheme is used: Mutation of an individual occurs with a constant probability throughout the entire run of the EA. The type of mutation performed differs, however. Rather than adding a small amount to each gene, as done by Gaussian mutation, only a fraction of the genes are affected. The type of mutation (of which there are three) is randomly decided.

Each individual has a probability of 85 % for each of it's genes to be either normally or highly mutated. The number of genes mutated is decided by a parameter P_d. Whether a gene is normally or highly mutated is decided by a parameter P_h.

Let n be a normally distributed number with mean 0 and variance 1 and $Lim1$ be the lower border of the gene and $Lim2$ the upper. Then high mutation is calculated as follows:

$$Newgene = Oldgene + (Lim2 - Lim1) * n/3$$

and normal mutation as following:

$$Newgene = Oldgene * P_a^n$$

where P_a is a parameter called mutation altering.

There is a 10 % probability of an individual having all of it's genes mutated in the following way:

$$Newgene = Oldgene * P_a^n$$

Finally there is a 5 % probability of an individual having all of it's genes mutated in the following way:

$$Newgene = Oldgene - 1$$

The reason for using the term P_a^n is, that we would like to alter the genes relative to how big the number represented by the gene is, but occasionally in a more drastic way. The high mutation type is included to give "new blood" to the population. Sometimes we would like to "stretch" all the genes in an individual to cope with the problem that genes are generally too small or too big. Finally the last mutation is done to give the EA an easy way to eliminate useless genes close to 0, as we we hope that a few feature measures can partition the data.

2.4 Kohonen networks

Kohonen networks are an example of self organising maps. The purpose of these classification networks is given a set of training vectors, to form two or more groups depending on the contents of the vectors. Thus one can view the networks as being a kind of intelligent projection from a space of high dimension (that of the input vectors) to a space of low dimension that is easier to interpret (usually one or two dimensional networks are used). Having trained a Kohonen network one should be able to present to it a new vector and have this classified. The basic idea of the Kohonen network is fairly simple:

- Build a two dimensional array of small size (10x10-30x30 seems appropriate in most situations), containing vectors of the same length as the input.
- Initialise the vectors in the network with random data.
- For each vector, v, in the training set find the vector in the net with the closest distance (euclidian or other) to v.
- In the neighbourhood around the closest vector, pull the network vectors toward the input vector.

The two latter steps (depicted in figure 2) are iterated.

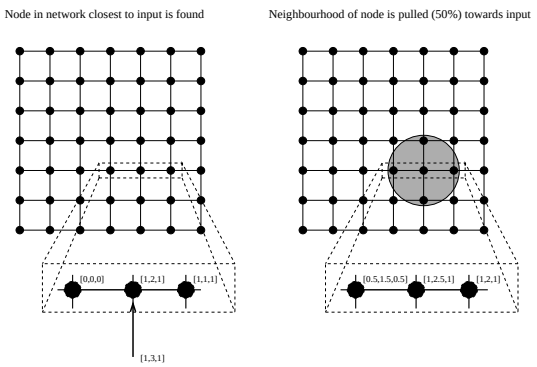


Figure 2: The workings of a Kohonen network

Apart from these basics of the network there are many subtleties and many choices to be made about the topology of the network. The

rate of falloff for the size of the neighbourhood to be pulled towards the input vector, the shape (e.g. rectangular or hexagonal) and the attraction distance are examples of these.

The distance measure for the input vectors is very important for the success of the network, as this is a vital part of the projection mechanism. Often a weighted euclidian measure is used, and thus it becomes an important task to find the best weights. As the resulting network is two dimensional, one cannot have significant noise in any highly weighted dimension as there thus wouldn't be enough *space* in the network to contain the projected data.

If the weighted input on the other hand represents data that in some sense is connected through an arbitrary shape in the input vector space, the trained Kohonen network will replicate this shape in two dimensions (possibly warped and twisted).

2.5 EA fitness evaluation

As mentioned above, the genes in the evolved chromosome is used as weights for a Kohonen network. We have fixed a size and a topology for the networks used in our system (the size is 15x15 and the topology is rectangular). For each individual we train a Kohonen network with the weights represented by the individual, in the distance measure. After training the network we need to evaluate it's fitness. This, of course, is quite a task since there is no obvious way to rate a trained network, and furthermore we want the fitness function to constrain the number of weights different from 0.0 (as we want the result to be a number of weights that can distinguish categories of input we want this number to be as small as possible in order to remove any feature measures that do not directly contribute to separating the data).

Our first fitness measure was to check where the input data was placed in a trained network. We found the centre of mass of each type of data (that is data matching a particular input criterion) in the network and partitioned the network according to these centres (all points closest to mass centre of type i was said to belong to group i – in other words the partitioning is the Voronoi diagram [7] of the mass centres for the different types of input data). We then counted the correctly placed input vectors and let the fitness be a percentage of the total. This worked very well for our first (and not extremely ambitious) experiment: Partitioning male and female speakers. When we moved on to other experiments we found that our fitness measure fell short. When trying to determine one speaker from the rest we found that dividing the network in two halves presented a problem. Even if the speaker had a unique range for one feature measure, this would most likely not be detected if there were entries both larger and smaller among the rest of the data (even if a trained Kohonen network placed one speaker inside a circle and everything else outside, it is not possible to place a line that separates the two sets of data).

Our second approach seems much better. For each node in the trained Kohonen network, we find the distance to all of the input vectors. The type of the closest input vector is assigned to the node. This colours the network according to the types of input. Afterwards we see where the input data is placed and count the hits and misses and weigh this according to the number of input vectors of that type. This hit-rate percentage is the primary part of the fitness function. Secondarily, we count the number of connected components in the coloured network. We would like all of the input data of a certain type to be grouped together (as we imagine that they in some sense are grouped together in their original representation), but we assume that this is not of primary importance as we care more about being able to discriminate the input so that the network is actually usable. As our test results show, this assumption does not hold completely.

In this approach we would still like the weights to be 0.0 on as many of the entries as possible, so we punish the algorithm for creating individuals having many non-zero entries.

Finally we would like to constrain the weights to some degree so that genes from different experiments can be more easily compared. First of all we reasoned that the comparative size of the weights was the most important, so we made the genes take values from 0.0 to 100.0. Secondly, we punish the algorithm a bit for having the sum of weights being far from 100.0

(it should not at all matter what this number is chosen to be) – thus if it is important for the algorithm to make one weight larger it must push some of the others down.

This is a rather loose constraint that keeps the weights close to representing percentages of importance for the final classification. Another idea would be to keep the sum of all weights constant at all times. This would, however, require quite atypical mutation and crossover rules as these would have to scale the weights of the new chromosomes. It becomes rather unclear what a crossover in this case represents and therefore we have chosen the looser constraint of making this part of the fitness function.

Altogether the parts of the fitness function are:

- The weighted sum of correctly placed vectors of type t, $w(t)$ (a percentage)
- The number of connected components, cc (an integer in the interval $[2; \text{inf}]$)
- The weighted number of non-zero genes, $w(n_z)$ (a percentage)
- The weighted distance of the sum of weights from a constant (100.0), $w(k)$ (a percentage)

We have weighted the different parts of the fitness as follows:

$$fitness = 10 * \sum_{t}^{N} \frac{w(t)}{N} + cc^{-1} - w(k) - \frac{w(n_z)}{10}$$

There is even a bit of reason behind the choices shown in the above equation: First of all the percentage of correctly placed genes is by far the primary goal. We are also happy whenever another zero weight is introduced, but almost never at the cost of the percentage of correctly placed genes going down. In earlier models this had a higher priority with the result that we could go from a state of no errors but with many nonzero weights to a state with up to a five percent error rate but a lot of zero weights. As can be seen in the fitness function formula all weights now only count as much as one percent of the correctly placed vectors. This effectively means that it has secondary importance and weights are only set to zero if they do not give rise to a worse partition of the input. The $w(k)$ element is rather small and only punishes the algorithm when a gene is set to 100.0 without resulting in a strictly better partitioning. The $cc^{-}1$ element is just chosen to be *not too big* but still worth improving on (going from three to two connected components counts as much as two and a half percents of the correctly placed).

Finally all of these values have been somewhat determined through trial and error – the final choice being satisfying for us as we have not seen anything which *we would consider* as bad behaviour when comparing chromosomes.

3 Experiments

Due to the fact that the running time of our implementation of the system is quite long, we can only perform a limited number of tests. However we have tried to cover as much as we can. The following sections describes what tests we have performed.

Our test data set consists of 12 persons uttering 10 words each. This yields 120 test samples. Of the 12 persons 6 of them are female and 6 of them are male. All of the utterances were recorded using the same equipment and was afterwards normalised in order to get an equal maximum amplitude.

Although our model supports partitioning according to several criteria, our current implementation only supports one.

3.1 Testing the Evolutionary Algorithm

Initially we will test some different parameters of the evolutionary algorithm in order to fine tune it. There are quite a number of parameters in the present implementation, as summarised in the table below.

Even for a limited number of values this gives a very high number of tests, which we can't possibly perform for all combinations of the parameter values. Therefore we will cut some corners doing the tests by only testing some of the parameters and only alter one at the time, using standard values for the other

Param. name	Std. value	Description
EliteShare	0.05	Part of the best Individuals to be held
Pc	0.7	The crossover rate
Pm	0.7	The mutation rate (Part of individuals to be mutated)
Pd	0.05	The mutation degree (Part of genes in an individual to be mutated)
Ph	0.01	The high mutation degree (Part of genes in an individual to be highly mutated)
Pa	2	The mutation altering (Factor of mutation to be raised to a normally distributed number when altering every gene in an individual.)
popsize	40	The population size
TermCond	100	The termination conditon

parameters. Apart from the standard values we will test the following parameter values:

- Pc: 0.3, 0.5, 1 and 0
- Pm: 0.3, 0.5, 1 and 0
- Pa: 1.5 and 2.5
- Ph: 0, 0.05 and 0.1

This yields 13 different tests – as we will perform each test thrice (once on each experiment type described below) we have 39 tests to perform.

3.2 Testing the Speaker Classification

We have made three overall tests with all of the 120 samples. We also planned to make the tests with the following subsets of the 120 samples:

- All persons speaking a particular word.
- 10 different persons each speaking a different word.

This was omitted, because of insufficient test material (less than 12 samples for the two subsets).

In the tests we have tried to classify the test data according to the following criteria:

- Male / Female (1 test)
- Single person / Anyone else (12 tests)
- Single word / Any different word (10 tests)

As for the last criteria, we do not expect that our system in its current form will actually be capable of *speech* classification, since none of the used feature measures take the contents of the speech signals into account.

Apart from testing whether the resulting network is able to partition the data correctly, we would would also like to make sure that it does not happen by a mere coincidence. We want to make sure that it is not always possible to create a good partition of random input vectors due to the relatively small amount of test data (we have 120 training vectors and compared to the size of each vector, 60, this is rather few). This was done by replacing some or all of the feature measures with functions outputting random data. Three test were then carried out. In the first case we tried completely random data, hoping for a bad classification, since this would indicate our system is not just good at finding a very complex ordering of random data. In the second test all but one weight are random (this weight partitions the set in two). In the third all but two values are random so (so that these two weights together, but not separately, partitions the set).

All together we performed 65 tests consisting of 39 used for tuning the evolutionary algorithm, 23 for speaker/speech classification and 3 for the random data experiments.

4 Results

Each of the test results presented in this section is the average of 5 actual test runs. The results for the experiments involving the mutation and crossover parameters of the EA are shown in figure 3 and 4. Note that in both cases the elite is not affected (even when the rate is 100%). The altering and high mutation parameters also showed slight variance in convergence points.

To show that the speaker/speech classification tests actually worked out as expected we have calculated the average ability of the system to correctly classify the speakers/speech. The results can be seen in table 1.

In the table Avg(cc) is the average of the number of connected components. Avg(cpX) is the average percentage of correctly placed vectors of type X. The reason that type 1 is

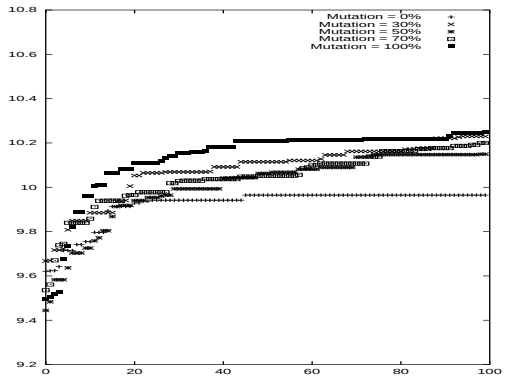


Figure 3: Average best fitness plotted against number of generations.

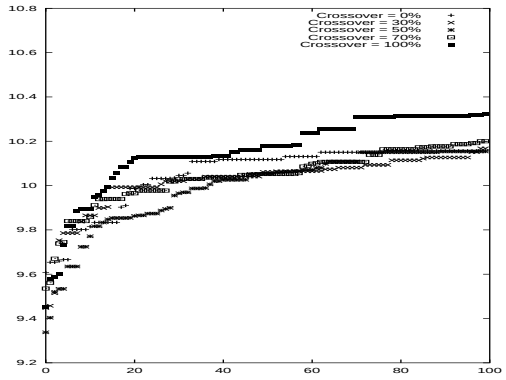


Figure 4: Average best fitness plotted against number of generations.

Experiment	Avg(cc)	Avg(cp0)	Avg(cp1)
Male/female	2.0	100%	100%
Person 1.	4.2	98.73%	100%
Person 2.	2.8	99.64%	100%
Person 3.	8.8	99.27%	100%
Person 4.	6.2	99.64%	100%
Person 5.	6.4	99.64%	100%
Person 6.	4.2	99.82%	100%
Person 7.	3.6	98.18%	100%
Person 8.	5.8	99.09%	100%
Person 9.	3.6	98.73%	100%
Person 10.	3.8	99.82%	100%
Person 11.	2.0	99.64%	100%
Person 12.	6.8	99.64%	100%
Average	4.93	99.32%	100%
Auditorium	5.8	98.52%	100%
Båd	2.0	100%	100%
Feta	4.2	99.44%	100%
Hæsligt	2.0	97.59%	100%
Julemand	3.2	99.07%	100%
Kaffemaskine	2.6	97.59%	100%
Rugklapper	2.0	100%	100%
Spæk	2.0	100%	100%
Talegenkendelse	5.4	98.33%	100%
Vinterbyøster	4.2	98.52%	100%
Average	3.34	98.91%	100%

Table 1: Results of the classification tests.

always better is that this is the smaller group in our tests, and a single wrongly placed vector of this type is therefore punished accordingly more.

For comparison the random data experiment gave the following end result: Avg(cc) 18.8, Avg(pc0) 90.67%, Avg(pc1): 95.67%

The two other random examples (noise in all but a few genes) were 100% correctly separated.

5 Conclusions

There are several observations to be made on the results gathered. First of all the different test runs on the same test case finds different solutions. As this is not improved by letting the EA run much longer we assume our problems to have many local optima.

A worse problem, however, is that the experiment with random data has shown us that it is quite possible to get a good classification of even completely random data. This could indicate several things. Firstly the size of network that we used might be too big for the small amount of data that we have – i.e. each input vector is likely to find it's own spot in the network. This hypothesis is strengthened by the fact that in the random data case we had a lot of connected components. Moreover the number of genes is rather high compared to the amount of test data, which means that a random training set is more likely to eventually result in a chromosome with a good fitness. Altogether the above means that it is difficult to conclude anything about the relevance of each of the feature measures.

The problems can thus be summarised as:

- The EA tends to get stuck in local optima.
- Too little training data was used.
- The amount of correctly placed training data is not necessarily an indicator of good fitness.

A solution to the local optima problem could be to investigate another type of EA, for example the Island Model. On the basis of the results of our experiments it seems that the

connected component count should contribute more to the fitness.

The problem of the amount of training data might actually be improved by trying to partition on more than one criterion. In this case we expect that the random data will prove harder to classify correctly and thus that a good partitioning has more validity.

Despite the problems there are also indications that it is worth continuing down this path, regarding the problem of speaker classification. In the experiments partitioning male and female speakers as well as the random sets with one or two distinguishing parameters we found the following qualities. First of all the EA is extremely quick at finding an optimal solution and secondly this solution was robust as well, in the sense that it was the same genes that were given the highest weights.

Another remark to be made is that the feature measures used in our current model are fairly simple. We compress an utterance of variable length into a fixed set of measures and the time dimension is dropped. If these measures indeed are not good enough for distinguishing data in more advanced experiments one might consider the following alternatives:

- LPC spectrum measures can be used to find formants in speech. As formants for a set of vowels to a large extent identify a person we expect that it would be benefitting to have characteristics of these as a part of the feature measures as well.

- After finding a set of features, the feature measures could be co-evolved using evolutionary techniques, so that we do not limit ourselves to predefined functions of the feature data.

Since we are able to find the important weights in the simple cases we still believe that EAs coupled with Kohonen networks are good at finding important characteristics about speakers, given larger training sets and some model improvements.

References

[1] William A. Ainsworth. *Speech recognition by Machine.* Peter Peregrinus Ltd., 1988.

[2] A. H. Grey Jr. J. D. Markel. *Linear Prediction of Speech.* Springer Verlag, 1976.

[3] Dimitris G. Manolakis John G. Proakis. *Digital Signal Processing.* Prentice Hall, 1996.

[4] Teuvo Kohonen. *Self-Orgaization and Associative Memory.* Springer Verlag, 1988.

[5] US Department of Defense. *US Department of Defense, Federal Standard 1015.* 1984.

[6] Thomas Rohde Ole Bystrup Jensen, Martin Olsen. *Automatic Speech Recognition & Neural Networks.* DAIMI, 1991.

[7] Schwarzkopf van Kreveld de Berg, Overmars. *Computational Geometry - Algorithms and Applications.* Springer Verlag, 1997.

Maintaining Diversity through Triggerable Inheritance

Henrik Sørensen and Jacob F. Jacobsen

Abstract—**This paper presents a technique for maintaining diversity in evolutionary algorithms through storing genetic material as inactive genes. We discuss and contrast the described technique w.r.t. previous work in this field. We examine various aspects of proposed techniques for maintaining and measuring diversity in evolutionary algorithms. Finaly we present preliminary results from using our proposed technique for solving dynamic problems.**

1 Introduction

One of the major shortcomings of standard evolutionary algorithms is their inability to maintain diversity in the population. This lack of diversity can lead to a number of problems such as converging to a *non-global* optima or not being able to react to changes in the environment. The lack of diversity is especially evident when dealing with multimodal problems or when using evolutionary algorithms to solve *dynamic* problems.

One approach to this problem is to maintain a memory structure, enabling the population/the individuals to remember useful genetic material from past generations. In evolutionary algorithms, techniques using *memory* can be classified as using either an explicit or an implicit memory structure. The former is often implemented as a globally available memory bank together with explicitly formulated strategies to store and retrieve information from it [5].An implicit memory structure is incorporated in the algorithm itself, by using redundant representation of the genomes, and leaving control of the storage/retrieval-operations to the algorithm. We have taken this approach, using a diploid representation of the individual's chromosomes. This way each individual carries genetic material which is not expressed in the phenotype. Thus, this inactive genetic information is shielded in the selection process (which is always based on phenotypes) and for that reason it can be propagated to future generations. In this way we reduce the risk of "selecting out" genetic material that might prove valueable in the future. This is especially useful when conditions in the environment are changing.

The soundness of using diploid chromosomes in a changing environment can be supported by more formal arguments. A diploid chromosome stores two alleles per gene, each being either dominant or recessive. A *dominance map* determines which gene is to be expressed, i.e. a given schema H has an expressed schema H_e(H). It is expected (though it depends on the dominance function) that the average fitness the expressed schema H_e(H) is greater than or equal to the average fitness of the base-schema:

$$f(H_e(H)) \geq f(H)$$

Thus, if a schema H is dominated (as in this situation), it is not as likely to be selected out of the population at an early stage, as would be the case if we had a standard haploid representation. This is due to the fact that the selection is based on the (in this case) higher fitness of the expressed schema.

The ideas presented above raise many technical issues: What kind of diversity measure should we choose (basing it on fitness or on individuals (genes), global or local measures)? How should the inactive genetic material be represented, and how should it be combined with other individuals' genes when mating? Should we able to guide the storage/retrieval of inactive genetic information (e.g by means of parameters). Will it be advantageous to combine this idea with other techniques, such as imposing a structure on the population (e.g. an island or a patchwork model)?

Most of these questions are best answered by experimenting with a concrete implementation, for which reason we defer this discussion to the *Implementation* and *Experiments* sections below.

2 Previous Work

Much work has gone into maintaining diversity within a population and quite a few well-known methods have been developed. Among these are *sharing* and *(deterministic) crowding* [4], which work with the idea of similarity between individuals, thus requiring a consistent distance measure in the population. Other approaches, such as *the Shifting Balance GA*, *Multinational GA*, and *Religion-Based EA* use subpopulations. However, diploid structures as a mean of enforcing diversity have apparently been studied very little. One such study can be found in [1] where D. Goldberg examines how a diploid representation combined with a dominance map can outperform a standard evolutionary algorithm on certain dynamic problems. More specifically, a genetic dominance function is used on each gene, determining which of the two stored values is to be expressed in the next generation. An oscillating 0-1 Knapsack Problem (varying weight constraint) is used as benchmark, and on this problem a diploid representation clearly beats a standard haploid representation – the latter is simply unable to track changes in the objective function. The difference is even more striking when using an evolving dominance function.

Based on his experimental results, Goldberg concludes – not surprisingly – that a diploid representation with (an evolving) dominance map is superior to standard methods as regards dynamic optimisation problems. What might come as a surprise is that the theory of diploidy and dominance has not been studied and implemented to a further extent, considering the ever-growing need for solving dynamic problems. One reason for this might be the lack of methods and theoretical guidance – this is indeed an experimental field with no guarantee of success. [5]

3 Implementation

Our implementation is made as an extension to the *Patchwork Model*. We expect our technique to be applicable in a wide range of evolutionary algorithms but the Patchwork Model, being a *spatial* model, has an interesting impact on the measurement of diversity in the sense that it allows us to measure this diversity *locally* and thereby allowing recall of genetic material to have a local cause and effect.

The Patchwork Model was introduced in [2] and has previously been used as a base for allowing self-adaption of the population [3] eliminating the need for tedious parameter tuning. In our work we have focused on techniques for maintaining diversity and for that reason, our implementation of the patchwork model does not include such self-adaption of the individuals. In the patchwork model the world can be seen as a two dimensional grid of fields where each field can contain a fixed number of individuals and where the grid borders are connected effectively transforming the grid into a torus. The individuals in the Patchwork Model are, in contrast to the traditional Cellular Genetic Algorithm (CGA) [6], allowed to move around in the world based on their autonomous measure of motivation. In our implementation the individuals are motivated for grouping up with the fittest individual in their field of view; the individuals have a fixed *view range* of one, meaning that each individual can see individuals on its own field and on the eight neighbouring fields, as visualised in figure 1. If the desired field is already fully occupied or there is no other individuals in the field of view, the motivated movement is chosen by random.

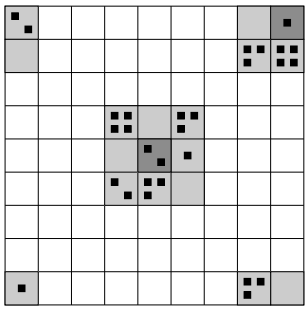


Figure 1: Spatial structure of the Patchwork Model

In the mating phase an individual can mate with any other individual in its field of view and the partner is randomly selected by those.

The mating process first creates two new individuals using a 1-point crossover operator [4] and the parent that initiated the mating process is then compared against the two new individuals; the fittest of those three individuals will survive and is inserted at the original position. Our implementation uses *diploid chromosomes* with real value encoded genes and during the mating process only the *active genes* (the phenotype) is subject to the crossover; the inactive genes are passed down from the individual that initiated the mating. After the mating process the individuals are subject to mutation using a Gaussian mutation operator with a decreasing factor defined by

$$aT = \frac{1}{1 + \sqrt{\text{generation}}}$$

After this point we calculate the fitness for each individual, move these according to their motivation after which we calculate and store the *standard deviation* of the specified measure (fitness or genome). This is done for each field and the deviation being stored is calculated by considering all individuals in the field of view. After this we are able to determine whether to store or recall genetic material for each of the individuals by considering the four bounds for store/recall of genetic material as shown in figure 2. The values of those bounds are of course problem-dependent and so are the settings for the store/recall pressure; *ps* and *pu*. We have implemented two different diversity measures: one based on the fitness and one based on the active genes. We defined the latter measure by separately calculating the standard deviation for each gene and then taking the average of these standard deviations.This measure, however, proved to be of limited value, probably due to the fact that many different gene combinations may result in identical diversity values.

Our implementation of the individuals *genotype*, active and inactive genes, uses real value encoding and we only keep one set of inactive genes but there is no obvious reason not to try keeping several inactive genes in further experiments. Keeping several inactive sets of genes for each individual would be more faithful to the genotypes found in the real world where the active genes are only a small fraction of the overall genetic material, but the effect of this is not obvious and should be studied in further work.

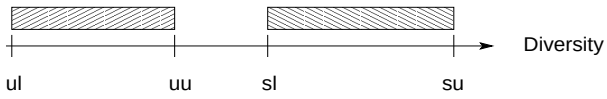


Figure 2: The four bounds for store/recall of genetic material

The pseudo code for performing the above mentioned steps are as follows:

initialise world population
for each generation
  calculate fitness
  move individual
  calculate deviation
  store material
  recall material
  produce offspring
  mutate individual

The initialisation of the world boils down to generating a number of individuals and inserting them in a randomly chosen field and obeying the restrictions on the field capacity.

4 Experiments

Our primary objective of the experiments was to test how our enhancement of the Patchwork Model has effect on dynamic problems. The experiments were made with focus on the effect on diversity, average fitness and best fitness both with and without our enhancement enabled. The two tests shown in this section have been performed using the same parameters. When testing performance of the "pure" Patchwork Model the probability for store/recall of genetic material has been set to 0, but when testing with the extension (marked by "trigger") we have tuned the 6 extra parameters for optimal performance as seen in figure 3. .

4.1 Test functions

We have experimented with two dynamic test functions: a dynamic version of DeJongsF1

Function no.	uu	ul	su	sl	pu	ps
1	1.0	0.0	5.0	2.0	0.1	0.6
2	1.0	0.0	4.0	2.0	0.2	0.5

Figure 3: Settings for store/recall

and a dynamic version of UrsemMultimodal1. The reason for trying our technique on a dynamic, multi modal problem is that we would like to see if the same problems occurred with our deviation measure based on fitness as the problems with the gene based diversity measure. When applying the fitness based diversity measure to a multi modal problem, several different configurations may yield the same fitness value, effectively decreasing the given measure of diversity and this problem resembles the previously discussed problem with our gene based diversity measure.

Dynamic problems can be categorised as either slowly changing, cyclic or abrupt changing. The experiments we have conducted has solely been on dynamic problems that are both cyclic and abrupt changing mainly because we find that these problems present a bigger challenge than, at least, slowly changing problems. Our tests function called *DynamicDeJongsF1* is defined as follows

$$f(\overline{x}) = \begin{cases} \sum_{i=1}^{3} x_i^2 & \text{if generation\%300} < 150 \\ \sum_{i=1}^{3} (x_i + 2)^2 & \text{otherwise} \end{cases}$$

and the dynamic multi modal test function is made with basis in the function named *Ursem-Multimodal1*

$$f(x, y) = \sin(2x - 0.5\pi) + 3\cos(y + 0.5x) + 0.5x$$

where we again just add the number 2 to x and y if generation%300 $\geq$ 150.

The settings for the store/recall of genetic material is summarised in table 3, and these settings has been used for 25 consecutive runs of the algorithm using a 15x15 world grid with a population size of 150. The tuning of the parameters for our extension can be guided by studying the diversity vs generation graph from running an experiment *without* our extension enabled. This graph gives a rough indication of proper settings for the four bounds.

4.2 Results

The results from the test of our technique on the first function is found in figure 4-6. When looking at the plot of the average fitness (figure 4) we observe that our technique during the first 150 generations, i.e. before the first change in the environment, performs a little worse than the "pure" algorithm but when later changes occur our technique is more reactive to this change. However, after around 150 generations our average fitness is more or less the same as for the pure algorithm. This should come as no surprise as our technique could be said to trade average fitness for diversity.

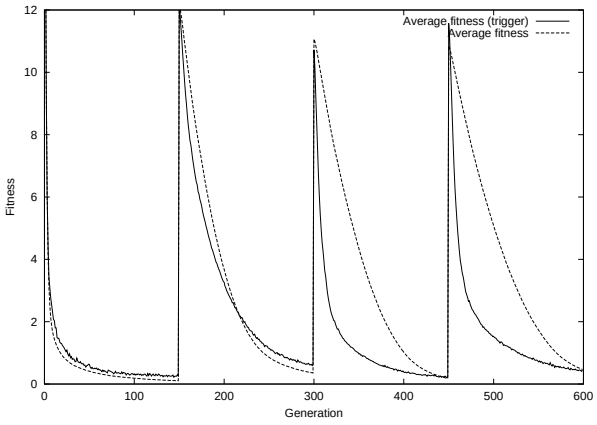


Figure 4: Average fitness

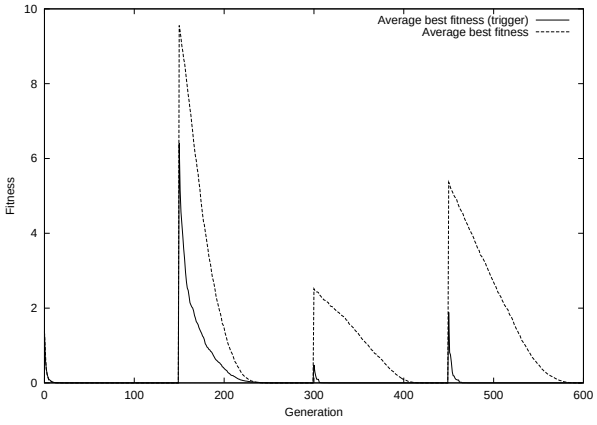


Figure 5: Average best fitness

The measure of *average fitness* shouldn't really be the only measure by which techniques

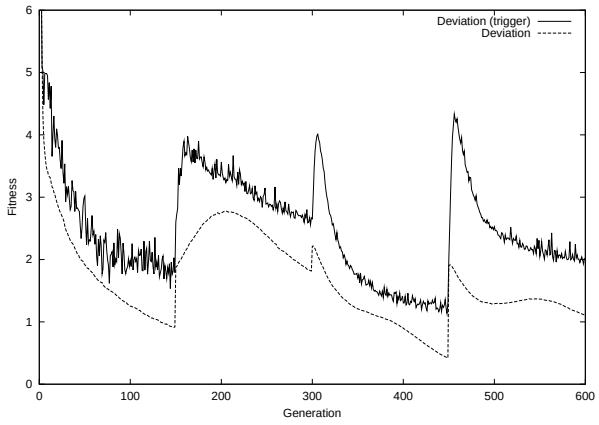


Figure 6: Deviation of the average fitness

for solving dynamic problems using evolutionary algorithms are compared; when using evolutionary algorithms the measure of *average best fitness* tells how good the algorithm is at finding optimal solutions and here the increase in diversity (see figure 6) has a positive effect as clearly seen in figure 5. Furthermore, figure 6 clearly shows that our technique *indeed* increases the deviation of the average fitness and roughly follows the *pattern* of the deviation of the pure algorithm.

Our second experiment using a dynamic, multi modal function gives the same results as those for the dynamic version of DeJongsF1. These results are visualised in figure 7-9 and shows that the diversity measure based on the fitness value performs quite well on this dynamic, multi modal problem and we observe that our small extension to the patchwork model enables us to correctly determine the global optima for the test function where the pure algorithm fails. Furthermore, the reader may have noticed that after the first change of environment using the function DynamicDeJongsF1 our technique performs worse than after further changes which could lead to the conclusion that our technique during the first two "environments" collects genetic material which can be used with advantage in later generations when the environment reverts to something encountered before. However, when looking at the average best fitness for the second test function, we observe that the technique is able to swiftly evolve towards the optimum after the first change in the environment. Why these contrasts exists is as yet not fully known but better parameter tuning may solve the problem for the first test function.

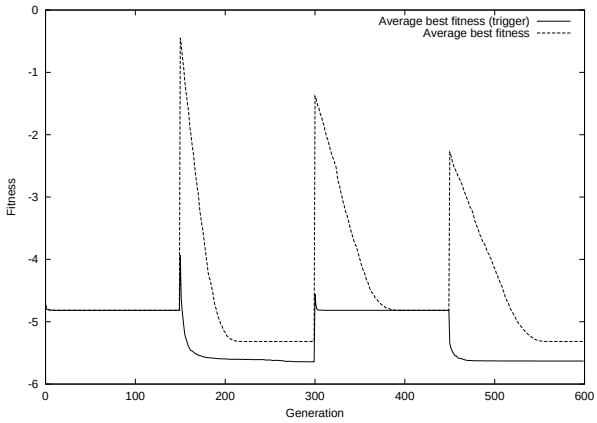


Figure 7: Average best fitness

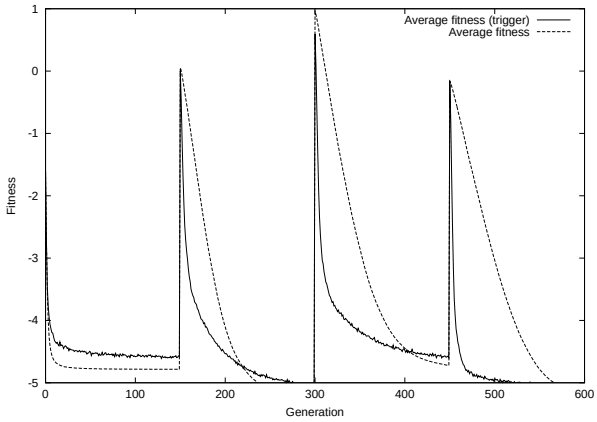


Figure 8: Average fitness

5 Resources

The work presented in this paper is available online at
http://www.daimi.au.dk/~ maniac/ToEC/triggerEA.tgz
and an online Postscript document of this paper is available at
http://www.daimi.au.dk/~ maniac/ToEC/triggerEA.ps

To run the program with our extension enabled you should type something similar to

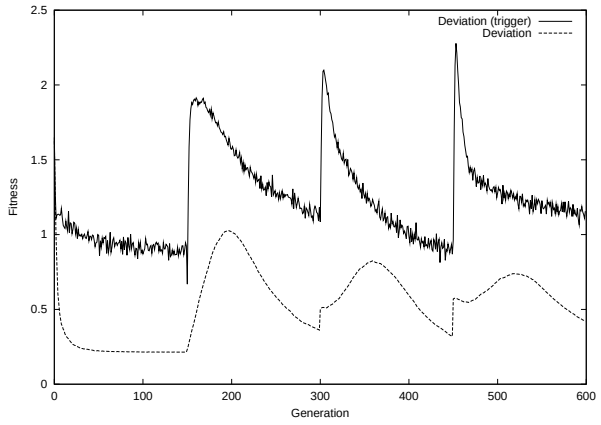


Figure 9: Deviation of the average fitness

```
java -Xmx128m PMmain -benchmark DynamicDeJongsF1
-diversity fitness -uu 1.0 -ul 0.0 -su 5.0
-sl 2.0 -pu 0.1 -ps 0.6 -popsize 150 -width 15
-height 15 -generations 600 -iterations 25
```

6 Conclusions

In this paper we have presented a technique for maintaining diversity in evolutionary algorithms. The main reason for the loss of diversity can be identified as side effect of the high selection pressure which is needed due to the fact that evolutionary algorithms have to evolve good solutions within an acceptable time frame.

We think that the results presented in this paper motivate for the usage of diploid chromosomes as a way to overcome the lack of diversity in evolutionary algorithms in general. To prove this point we should study the effect of using diploid chromosomes in a wide variety of evolutionary algorithms.

The functions we have used as a basis for our experiments were all cyclic and abruptly changing and we find it interesting to study how the described technique would perform on non-cyclic problems as well.

Finally, we find that there need to be done further investigations into the field of how best to measure diversity for evolutionary algorithms. The diversity measure based on gene values briefly discussed in this paper did not work well and we had to fall back to the well known measure for diversity based on fitness using the statistical tool of standard deviation. Perhaps deeper study into the field of biology would point in a useful direction.

References

[1] David Goldberg. Nonstationary function opimization using genetic algorithms with dominance and diploidy. In John J. Grefenstette, editor, *Genetic algorithms and their applications*, pages 59–68, Hillsdale, New Jersey, 1987. Lawrence Erlbaum Associates.

[2] Thiemo Krink, Brian H. Mayoh, and Zbigniew Michalewicz. A PATCHWORK model for evolutionary algorithms with structured and variable size populations. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99)*, volume 2, pages 1321–1328, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[3] Thiemo Krink and Rasmus K. Ursem. Parameter control using the agent based patchwork model. In *Proceedings of the Second Congress on Evolutionary Computation (CEC-2000)*, volume 1, pages 77–83, San Diego, CA, USA, 2000.

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

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

[6] Darrell Whitley. Cellular genetic algorithms. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, page 658, San Mateo, CA, 1993. Morgan Kaufman.

Tuning tournament selection for Evolutionary Algorithms

Uffe Bolsing

Abstract— **It can be expected, that a minor variation in the implementation of the tournement selection operator, can have significant effect, on the performance of a genetical algorithm. This paper describes, a emperically examination of 3 different implementation, of the tournament selection. The paper tries to eamine emperically, what implementation is best. The 3 different procedures of tournament selection examined is:**

1. **Both individual in tournament is selected by random.**
2. **Only one individual is selected by random.**
3. **Mutation, crossover and selection is all done in parallel. The principle here is, that if mutation or crossover gives a fitter individual then keep that individual.**

All three variants, can be implemented in a given Evolutionary Algorithms. The emperically study described in this paper, shows that for the chosen benchmark problemes, and the chosen genetic algorithms, variant 2 is superior for all, but one of the tested bench mark problemes.

1 Introduction

A methodological problem, in the field of Evolutionary Computation, is the enourmously number of different possibily of implementations for Evolutionare algorithm. The general description, of a evolutionare algorithm, is namely rather vague, and leaves a great deal up to the programmer implementating the algorithm. That even though minor variationes, can influence the performance of the algorithm enormously. Here we tries to examine which implementation of tournament selection is best, so the programmer has some guidelines when constructing a given EA, and he dont have to, just choosing how to implement the selection operator by random. This paper will not just examine which implementation is the best, but also shed some light on how much, the different implementations influence on the performance of EA. To do the experiment we have to select a algorithm and some benchmark problems. Here 7 rather different benchmark problemes is examined, but unfurtunately only one algoritms is examined.

2 Model description

The following 3 different implementation variants, of the tournament selection, is examined:

- `Variant 1:` Both individuals for the tournament selection is chosen by random.
  1. double[] newpop=new Individual[popsize]
  2. for(i=0;i¡popsize;i++){
  3. Pick two random individuals I and J.
  4. Clone the best and put in newpop[i];
  5. }
  6. pop=newpop; With this method we further have to use elitism is, i.e the best solution is stored before crossover and mutation, and then reinsert afterwards. This is to prevent the currently best solution, to be selected out. A disadvantage with this method is, that even though elitism is used, we still risk that the 2.best or 3.best.. is selected out, because some of the individuals dont even get evaluatet in the selection process, but are simply picked out by random number generator. Furthermore the extra use or random number generator takes computing time, and so does elitism. To implement elitism we also need and axuillaridatastructure, to store the elite individuals individuals genes.

- `Variant 2:` Here only one individual is choosed by random

  1. double[] newpop=new Individual[popsize]
  2. for(i=0;i¡popsize;i++){
  3. Pick one random individual I.
  4. If I is better than newpop[i], clone I and overwrite newpop[i]
  5. }
  6. pop=newpop;

  Variant 2 is easier to implement than variant 3, because you have natural elitism, this also makes the algoritm slightly faster for the same number of evaluations. All genes are evaluated in the selection process, so the most fitt individuals are almost certain to remain after selection, and the chance of having dublicated the best genes, is as good as in variant 1. This holding on, to the best genes, can have the unfortunate side effect, that one get stuck on a local optimum. So on a problem with many local optimum, one could naively think variant 1 is more efficient. But on a more monotome problem this variant is obiviosly very effective.

- `Variant 3:` In this variant we do mutation, crossover and selection in parallel, rather than in sequence:

  1. double[] newpop=new Individual[popsize]
  2. while(not done){
  3. create mutationpop by mutation of pop
  4. create crossoverpop by crossover of pop
  5. for(i=0;i¡popsize;i++){
  6. newpop[i]= most fit of mutationpop[i], crossoverpop[i] and pop[i]
  7. }
  8. pop=newpop;
  9. }

The pricip here is, that we only keep the individual created by crossover and mutation if they are fitter, than the original individual. On the other hand, we never dublicate a good individual. We keep all good individuals, but unfourtunately also all the bad, which didnt got mutatet or crossed to a fitter individual. Variant 3 is actually quite different from 1 and 2. In 3 we dont even use a random generator for the selection proces. Unfortunately we have to do more evaluations in the selections process, because we now have to select the best individual out of 3 individual, instead of out of 2 individuals as in variant 2 and variant 3. That is, in variant1 and 2, we makes 2 evaluation to select one individual, but in variant we have to make atleast 4 evaluation, to select one individual. An interesting thing, about the selection principle behind variant 3, is that it in some way, actually reasembles the real biologi evolution more closely than do variant 1 and 2, because in real biologi individuals dont get cloned. Also bad mutation is discarded directely in variant 3, likewise in biological contekst a bad mutation might be cancer.

3 Description of the experimental setup

To do the experiment, we have to choose some parameters. But it ought to be noted, that have some other values of the parameters been chosen, we might have got completely different result. The parameters, are namely known to, interference in complex ways.

The chosed parameters:

- Number of evalutions is fixed on 200000.
- Gaussian mutation operator, with variable variance var(t)=1/(1+t).
- Probability for mutation is fixed on 0.75.
- Arithmetic crossover operator.
- Probability for crossover is fixed on 0.9.
- Population size is 100.
- Number of identically runs is 50. In the initialisation of the population, all individuals are uniformely randomly distributed on the interval, given in the benchmark problem. For the last generation, means and standard variantion is computed over 50 identically runns.

The tested benchmark problems:

- Ackley F1 20D Intended usage:Hard test for global optimization. The problems contains "minimum rings" around the global minima with almost the same fitness, as the global mininimum. Problem details Function:

$$\begin{aligned} f(\overline{x}) &= 20 + e - \\ & 20 * exp(-0.2 * \sqrt{(\sum_{i=1}^{20}(x_i^2)/20)}) - \\ & exp(\sum_{i=1}^{20}(2 \cdot \pi \cdot x_i)/20) \end{aligned}$$

where

$$-30 \leq x_i \leq 30$$

Type:Minimization No. of minimas:More than 1000 Optima radius:0.15

- Griewank F1 20D Function:

$$\begin{aligned} f(\overline{x}) &= 1/4000 \cdot \sum_{i=0}^{20}(x_i - 100) - \\ & cos(x_1 - 100/\sqrt{1+1}) \cdot \\ & cos(x_{20} - 100/\sqrt{20+1}) + 1 \end{aligned}$$

where

$$-600 \leq x_i \leq 600$$

Type:Minimization

- Rastrigin F1 20D Intended usage:Test of multimodal on problems with extremely many peaks. Problem details Function:

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

where

$$-5.12 \leq x_i \leq 5.12$$

Type:Minimization No. of maxima:More than 50 No. of minima:More than 50 Optima radius:0.2 Optima descriptions:The minima are located near (0,0,...,0)

- Rosenbrock F1 20D Function:

$$f(\overline{x}) = \sum_{i=1}^{20}(100 \cdot \sqrt{x_i - x_{i-1} \cdot x_{i-1}} + \sqrt{x_{i-1} - 1})$$

where

$$-100 \leq x_i \leq 100$$

Type:Minimization

- Schaffer F6 Intended usage:Hard test for global optimization. The problems contains "minimum rings" around the global minima with almost the same fitness as the global minima. Problems details Function:

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

where

$$-100 \leq x \leq y \text{ and } -100 \leq y \leq y$$

Type:Minimization No. of minimas:More than 1

- De Jong F4 30D Function:

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

where

$$-1.28 \leq x_i \leq 1.28$$

Dimensions:30 Type:Minimization No. of minimas:1+ Optima radius:0.2 Known optimas:GMIN(0.0,0.0,...,0.0) Important:Only 5000 evaluation are used on the testrun of Schaffer F6.

- Ursem multimodal F8 20D Intended usage:Scalable testproblem where the peaks are not located on axis-parallel lines. Problem details Function:

$$f(\overline{x})2 \cdot cos(2 \cdot \pi \cdot (x_1 \cdot x_2 \cdot ... \cdot x_n))$$

$$-4 \cdot (\sum_{i=1}^{n}(x_i + 1^2) + (cos(2 \cdot \pi \cdot x_i))$$

where

$$-5 \leq x \leq 5 - 5 \leq y \leq 5$$

Type:Maximization No. of maxima:Many No. of minima:Many Optimum radius:0.2

Optimum descriptions:The global maxima and most of the local maximas are located, at one end of the search space. Some of these maximas are hard to detect. Known optima:GMAX(-1,-1,...,-1),

Table 1: Values for last generation.

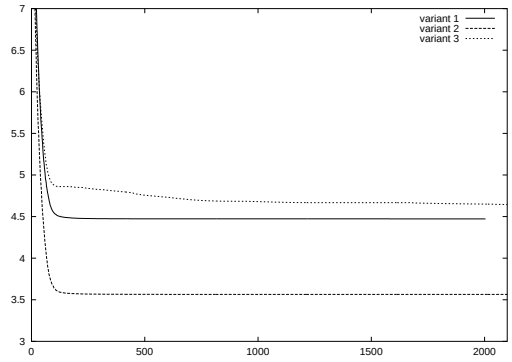
Function	Variant	Mean	standard diviation
Ackley	1	4.47	1.41
	2	3.56	1.35
	3	4.63	1.09
Griewank	1	0.0480	0.163
	2	0.0312	0.075
	3	1.91	1.45
Rastrigin	1	8.83	3.24
	2	8.15	2.76
	3	9.15	3.61
Rosenbrock	1	76.2	113
	2	48.4	96.3
	3	412	1028
Schaffer	1	0.0145	0.0142
	2	0.0151	0.0139
	3	9.35E-5	3.53E-4
DeJong	1	3.76E-7	2.26E-6
	2	4.30E-16	3.43E-16
	3	3.46E-5	9.44E-5
Ursem multimodal	1	2.25	0.0719
	2	2.26	2.63E-4
	3	-367	52.3

Table 2: Variant 1 is with both selection individuals chosed randomly. Variant 2 is with one selection individual chosed random. Variant 3 is with mutation, crossover and selection in parallel.

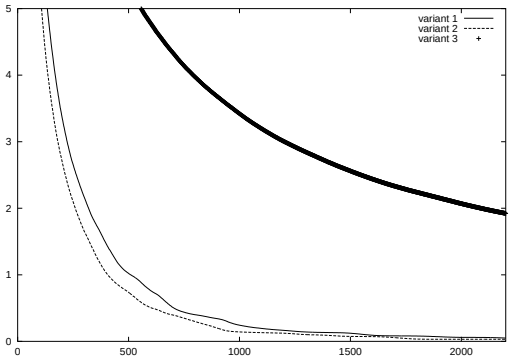
4 Conclusions

Ackley: For this benchmark problem, variant 2 is a little better than variant 1 and variant 3. It is also seen, that both variant 1 and variant 3 converge very fast to a optimum. Unfortunately for many of the runns, the found optimum is only local. **Griewank:** Variant 2 is a little better than variant1, and variant 3 gives very bad performance for griewank. Variant 3 gives a mean average best fitness at 1.91 where variant gives 0.0480. **Rastrigin:** Variant 2 is still a little better than variant 1. Variant 3 performs pretty good as well. But the performance differences, is generally very small for this benchmark problems **Rosenbrock:**Here variant 2 converge to optimum much faster than variant 1. Variant 3 performs very poor compared to both variant 1 and variant 2. **Schaffer:** For schaffer variant3 actually performs pretty amasingly. I reaches a mean best fitness at 9.35E-5 where, variant 1 reaches 0.0145 and variant 2 reaces 0.0151. This is the only benchmark problems where variant 3 performs best. This problem is characterized by it contains minimum rings, around the global minima with almost the same fitness as the global minima. And indeed it is seen, that variant 2 and variant 1 converge too a best average fitness around 0.015. Where variant actualy goes for the global optimum. Variant 3 that uses parallel mutation, crossover and selection, is characterized by never throwing the best solution away, but at the same time it also keeps many of the worst individuals. That is it keeps great diversity in its population, for this benchmark problem that helps advoid premature convergence. **DeJong:**For this DeJong, variant 2 performs much better than variant 1 and variant 3. Variant 1 and variant 3 performs pretty equal. It is furthermore seen, that at the lower generation variant 2 and 1 are allmost equal, but at the last generation variant 2 is much better. The reason for this, is proably that variant 2 evaluate all individuals in the selection process, where variant 1 select some individuals out by random. That is, variant 2 risks selecting some of the best fitt individuals out. That, selecting the best individual out, is naturally worse, the better individuals you got. **Ursem multimodal:**Here variant 2 performes alot better than variant 1 in terms of standard diavation, but actually a little worse in terms of best average fitness. Variant 3 performs terrible for this problem. Variant 3 have mean best fitness -367, where variant 2 have 2.26 and variant 1 have 2.25. **For all benchmark:** For all problems variant 2 performs slightly better than 1. At the same time variant 2 is easier to implements because it has natural elitism, so you dont have to code in elitism. Variant 2 also involves less computational overhead by saving a random generator operation, also the more precise solution you want the more the reason to choose variant 2. From a teoretical point of view, it is not suprisingly that variant 2 performs better than variant 1, because in variant 2 we evaluate all individuals and there by not only keep the best individual, but also almost certain keepin the 2. 3. .. best solutions. One can say that in variant 2 all individual is evaluatet atleast once, where in variant 1 some genes dont even get evaluated at all in the tournement selection, but are simply picked out by random. All in all i simply dont see much use of variant 1, because variant 2 for all the problems turney out atleast as good as variant 1.

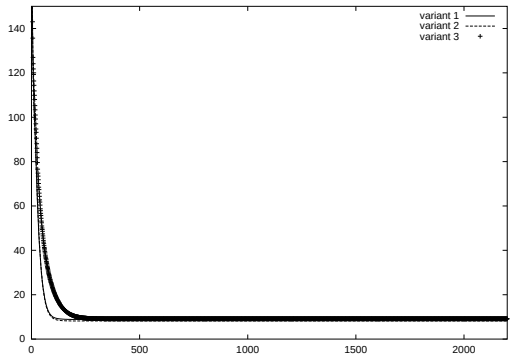
For all but the schaffer benchmark variant 3 performed much worse than 2 and 1. For the Schaffer benchmark problem variant 3 actually performed much better than 1 and 2. So for special problems like Schaffer, with lots of local optimums, variant 3 can be attractive. The reason i think one ought to consider variant 3 is that for some problems variant 2 simply can get stuck on the local optimum, where variant 3 keeps bad genes too, that can get one out of the local optimum by crossover. The result that variant generally performed worse than variant 1 and 2, are not easily explaned. But the fact that variant 1 and 2 can clone the good individualls, proably helps them converge faster than variant 3. A better solution on problems like Schaffer than using variant 3, might be to use a religion-based spatial model for evolutionary algorithms, which also keep better diversity of its population. One should also bear in mind, that we only tested the 3 different tournamens selection implementations, on one evolutionary algoritms.



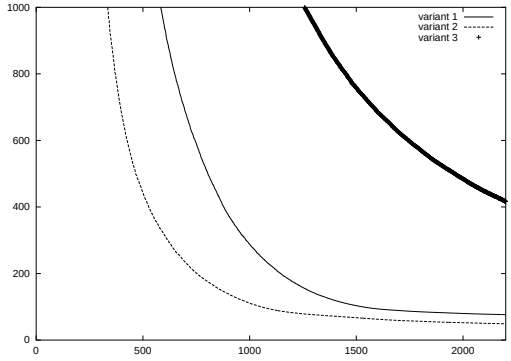
(a) Ackley F1 20D



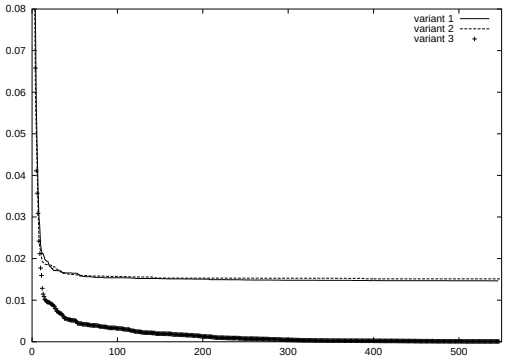
(b) Griewank F1 20D



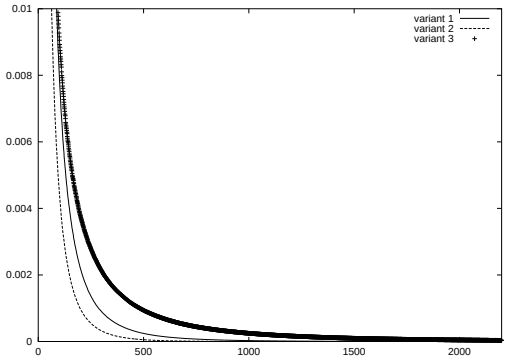
(c) Rastrigin F1 20D



(d) Rosenbrock F1 20D



(e) Schaffer F6



(f) DeJong F4 30D

Combining particle systems with religion based EAs

Henrik Refslund

Abstract— **Particle Systems is a simple way to explore a continuous search space. They require relatively few evaluations, and they are simple to implement. The idea behind this paper is to improve the computational power, by extending a standard particle system with a "Religion" concept similar to that found in Religion Based Evolutionary Algorithms. The motivation is, that you in this way get both a local and a global frame of reference for each particle, without having to keep track of the position of particles relative to each other.**

1 Introduction

The search spaces considered in this paper are all continuous and real-valued. They can be in any number of dimentions. It is assumed that a search space is wrapped in such a way, that if a particle moves out of the search space in one direction, it reenters elsewhere in the same direction.
There is no inherent obstacle against discrete or unwrapped search spaces, but the implementation is prettier and more general this way.

The standard particles system consists of a number of particles. Each particle has a position in the search space, and hence represents a solution. If a total ordered fitness function is assumed, each particle can be evaluated only once per iteration for all comparion and selection purposes. For each iteration of the algorithm, a new position is calculated for all particles, based on their current position and velocity[17].
So, a particle is a gradually changing solution. In this way particle systems doesn't differ much from other search techniques such as local search. How should the next position for a particle be chosen? Inspired by local search, one might ask for a hill climbing effect by picking the best next position from the immidiate neighborhood around the particle. This, however, will require knowledge of the surrounding neighborhood.
The present model gets it's strength from simplicity. There is no local neighborhood concept. Likewise, particles doesn't sample their surroundings when deciding on their next move. This keeps the number of evaluations down at exactly one pr. particle pr. iteration. This is an important fact, since evaluations are often the single most expensive operation in a real life setting.

[17]The velocity and position update functions are explained below

2 Model Description

A particle look like this:

Particle	
x	Current position
y	Current velocity
p	Best position found by particle
religion	
fitness	

The only new field, compared to standard particle systems, is `religion`. All particles belong to a religion, which serves as a local frame of reference for that particle. A `Particle` object might need other fields, such as the fitness in the best position found so far(`p`) in order not to introduce extra evaluations.

Besides the set of particles, the algorithm will also need to keep track of information such as:
The global best solution (and the fitness at this position).
The best position for each religion (and fitnesses).

The main algorithm can be sketched as:

```
Generate random set of particles
while (!done) {
  for all particles, p {
    - Update velocity vector, p.y
    - Update position, p.x
    - p.fitness = Evaluate()
    - Update best solutions
      (for p, p's religion, and globally)
  }
  if (some criteria) {
    Convert a number of particles
  }
}
```

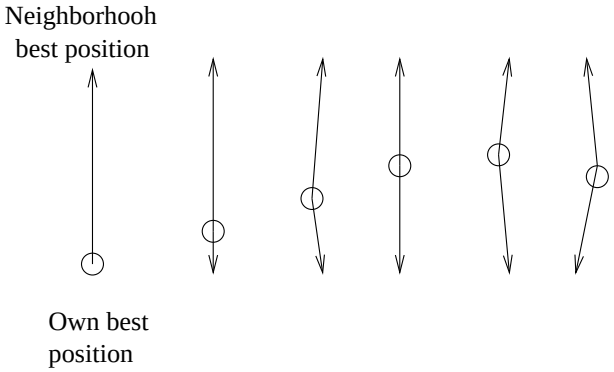


Figure 1: Stuck in the middle.

2.1 The Velocity Update Function

This a quite important part of the algorithm. In the standard particle system it looks like this:

$$\mathbf{v} = \mathbf{v} + \mathbf{(own\ best\ -\ p)} + \mathbf{(neighborhood\ best\ -\ p)}$$

In the present model, the velocity update function looks like:

$$\mathbf{v} = \mathbf{v} + \mathbf{(religion\ best\ -\ p)}$$

In both cases a bit of random noise is added, even though it is not explicitly stated. Also, it is possible to multiply coefficients to each term.
I wanted the particles of given religion to converge towards a single (local) optimum in the search space. As the figure 1 shows, taking in both the "own best" position and the "religionwise best" position is a problem to this end. The figure doesn't show the current velocity and noise, since both terms can be assumed relatively small[18]. It illustrates how a particle would get stuck between its own best position and the religionwise best position, since the two vectors cancel each other out. This fear was confirmed by experiments.

With the given velocity update function, the particles of a religion will all cluster together around the best position collectively known, without any other need for information sharing other than that mentioned above. Multiplying a large coefficient to the **v**-term, will give a large clustering area, since particles will be slow to change their direction when passing the rel best position.

2.2 The Position Update Function

Not much work is needed here. Just move the particle as indicated by the new velocity:

$$\mathbf{x} = \mathbf{x} + \mathbf{v}$$

2.3 Convertion

After having updated all particles the algorithm may convert a member of one religion to another religion. This convertion should probably be based on the best fitnesses found by the religions in question. This can be done in constant time.
This step is similar to the selection process in standard Evolutionary Algorithms. The effect is, that a successful religion will attract more members than one stuck at an unoptimal hill (or valley, depending on maximization or minimization, respectively). In addition, a converted particle will have to wander from it's current position to the position of it's new religion, exploring the search space in between.

[18] It makes sense to set a upper limit on the velocity, if you don't want to just sample random positions.

3 Experimental Setup

3.1 Benchmark Problems

I've have developed the model with two different types of search spaces in mind.
One, a search space consisting of a number of peaks and valleys. Two, a search space with slopes, ridges, flat areas, and even infeasible regions. It would be nice, if the model works for both kinds of problems. I've put together two benchmark problems for this purpose. Both benchmark problem are in two dimentions, since this makes it possible to visualize the results:

f(x, y) = $sin(x)+$
$sin(2 * sqrt(x^2 + y^2)) + sin(y)$

g(x, y) = $0.2 * |x - 5| + 0.5 * y + 6$, 1st quad
$-sqrt(x^2 + y^2) + 14)$, 2nd quad
6, 3rd quadrant
$0.2 * |x - 5| + 6$, 4th quadrant

In addition, `g(x, y)` contains a few infeasible regions.
f(x, y) is shown in figure 2. Figure 3 shows benchmark problem 4, except for the infeasible regions. Benchmark 4 is a minimization problem, so the global optima is in quadrant 2 (at **(x, y) = (-10, 10)**).

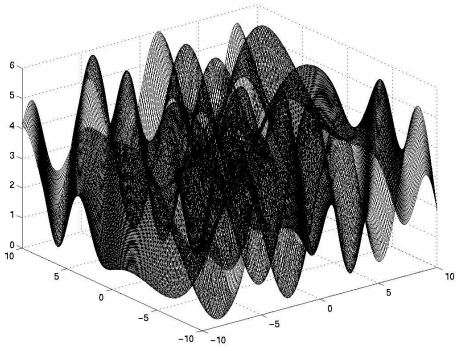


Figure 2: Benchmark problem, **f(x,y)**.

The algorithm works quite well with both kinds of benchmark problems. A religion/swarm has the ability to climb along a ridge or up a slope (or down, as applicable). If a particle enters an infeasible region nothing bad happens. It will just continue on it's way towards the religion best position. It can be discussed whether this is an acceptable behavior: If all particles of a religion is contained in the same (maybe even convex) infeasible region, they will probably not leave it, until thay are converted one by one. This is of course a waste of computation. However, a particle may need to cross an infeasible region in order to get to a better position.

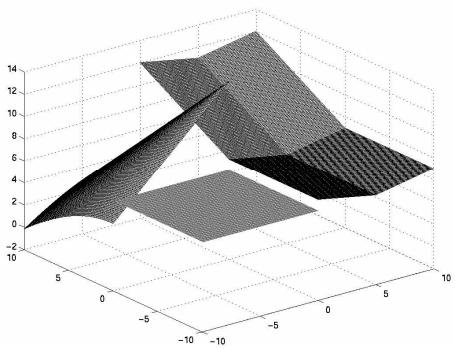


Figure 3: Benchmark problem, **g(x,y)**.

3.2 Stages of the Algorithm

In general, a run of the algorithm, can be divided into three stages:

- **1:** Starting out, the particles are scattered randomly across the search space.
- **2:** After a while, each religion will cluster around a hill.
- **3:** After many convertions, the best fit religion will have converted all other particles.

There are several parameters that influence this behavior: The frequency and number of conversions, the velocity update function, and so on.

4 Results

I've have run an implementation of the algorithm on the above benchmark problems, and

summarized the results below. The graphs show averages of (1) average fitness of all particles, (2) average best fitness of religions, and (3) overall best fitness.
In general, the overall best fitness improves in small steps. The average fitness of all particles stays rather high, because some religions stay at local optima, which are inferior to the global optimum. Figure 4 shows the averages of a number of runs with benchmark **f(x,y)**.

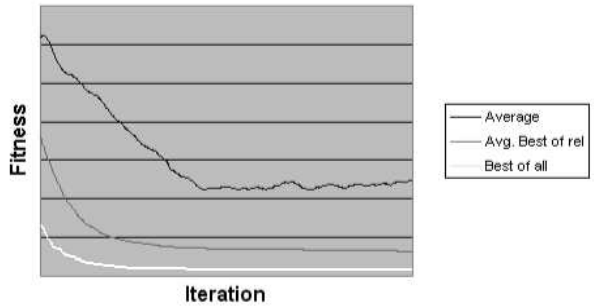


Figure 4: Benchmark 3.

If the number of religions is set to 1, the algorithm will behave as a standard particle system. Experimenting with this, it is my impression, that the many-religions variant is less likely to get stuck at suboptimal hills. Figure 5 shows the averages of a few runs.

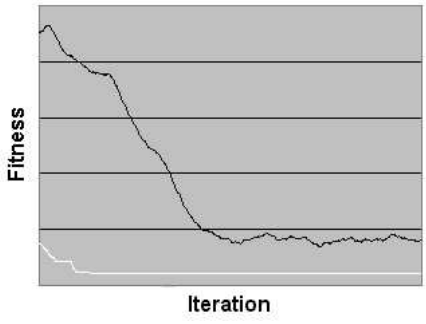


Figure 5: A single religion.

In both examples the algorithm has run for 500 iterations. As can be seen, the best solution quickly stagnates. This corresponds to the swarms settling on local optima.

5 Possible Improvements

I haven't mentioned the `mutation` concept, found in evolutionary algorithms. It could, however, easily be applied to the model. For instance, you could pick a particle and move it to a new (random) position every once in a while. As with convertions, the moved particle would have to travel back to it's swarm, exploring the search space in between. As described above, the only mutation is the noise added in the velocity update function.
Another option, is to add temperature to the system, as in simulated annealing. This temperature, could then control parameters such as the frequency and number of convertions, the coefficients in the velocity update function (average and max speed, inertia, ..).

6 Conclusions

The model seems to reduce the number of runs getting stuck at suboptimal hills, compared to a standard particle system. It is quite simple to implement. It only requires a liniar amount of space and keeps the number of evaluations at a minimum. Apart from the evaluations there are no heavy computations involved.
This simplicity comes at a cost. In the given form, it can still produce mediocre results, even if it runs for a long time (since it doesn't escape the final optimum without mutation). Another major drawback, which it shares with many other EAs and the standard particle system, is the high number of parameters. With a bad combination af these parameters, it will be likely to perform poorly.