Giving Structural Descriptions of Tree-like Objects from Binary Images Using Genetic Programming

Róbert Ványi
Department of Informatics, University of Szeged
Árpád tér 2, H-6720 Szeged, Hungary
e-mail: vanyi@inf.u-szeged.hu

Gabriella Kókai
Department of Programming Languages
Friedrich-Alexander University
Martensstr. 3. D-91058 Erlangen, Germany
e-mail: kokai@informatik.uni-erlangen.de

Abstract

The aim of this paper is to present a general method for structurally describing binary images. Such a method has to be reasonably fast and should give a correct and usable result. The convergence speed highly depends on the fitness calculation, therefore several possible fitness functions and their computations are discussed. To achieve a correct result, the evolutionary operators and the evolution process have to be designed carefully. They are also discussed in this paper[1].

1 INTRODUCTION

There are many cases when some kind of shape is to be reconstructed from a digital image. A well known experiment to describe a shape using a structural description was made in 1992, when Koza applied genetic programming on Lindenmayer systems to try to evolve the shape of the so called quadratic Koch island [5]. After that several attempts have been made to improve the convergence speed. Jacob applied sequences of symbols to avoid unnecessary drawing, and this way reduced the search space and improved the quality of the evolved shapes [4]. As a real world application the outline of the GREDEA system was developed by the authors [11]. This system is specialized for the medical domain, and consist of many parts, of which only one is the structural description. The implementation of this system is still in progress as well as the improvements of the already existing parts. However the basics of the description part are fixed now and during the implementation it became clear, that it can be extended to a greater domain to be a general method for structurally describing binary images.

One very important part of such a method is the fitness function. Also the evolutionary operators and the evolution process have to be designed carefully. In this paper the details of this general method are discussed as follows. First the possible forms of description are discussed. Then in Section 3 the outline of the evolution process is given, followed by the genetic operators in Section 4. The fitness functions are detailed in Section 5. In Section 6 a possible solution for a special problem, namely for the evolution of thick lines is presented. The experiments are detailed and the results are discussed in Section 7. Finally the conclusion and some future plans are mentioned in Section 8.

2 FORMS OF DESCRIPTION

First the objects are given from which the structural description must be extracted. These are the binary images.

2.1 BINARY IMAGES

An image is usually a function which maps colors or gray level values to a set of pixels. This means it is a function defined over $\mathbb{Z} \times \mathbb{Z}$ and its values come from $\mathbb{Z}_k = \{0, 1, \ldots, k-1\}$ ($k \geq 2$). Also the domain of this functions is finite. Later it will be easier to have a definition for this domain, which will be referred to in this paper as container.

Definition 1 A *container* is simply a Cartesian product of (usually two or three) finite sets. Depending on the number and cardinality of these sets we can speak about the *dimensions* of the container.

In this paper when images with size $n \times m$ are used, a container $I = \mathbb{Z}_n \times \mathbb{Z}_m$ is assumed. Now it can be defined what digital image is.

[1]This work is supported by the Deutscher Akademischer Austauschdienst (DAAD) and by the Bayerisches Habilitationsförderpreis

Definition 2 A function $f : I \rightarrow \mathbb{Z}_k$ $(k \geq 2)$ is called *digital image*, where I is a container. The size of the container gives also the size of the image. If $k = 2$ the image is called *binary image*. From now on, when $k = 2$ then 0 means the background and 1 means the foreground.

There are several ways to describe an image, or the structure of an image. Some of them are considered here, these use graphical primitives with simple structural information. To convert these structures into (binary) images, the so called turtle graphics is used [1]. Methods using only primitive sets, or having complicated structure such as Iterated Function Systems [2] are not considered here.

2.2 TURTLE GRAPHICS

The turtle is an object in space such that its position is defined by a vector and its orientation is defined by a coordinate-system. The coordinate-system determines the heading ($\vec{h}$), the left ($\vec{l}$) and the up ($\vec{u}$) directions. Using these vectors, the turtle can be moved in space. There are several turtle operations, also known as *turtle commands*. The most important ones are drawing ('F') and turning left and right ('+' and '$-$'). There are special turtle operations, called *stack operations*. These are denoted by '[' and ']', and are used to save and restore turtle position and orientation. Using stack operations, branching structures can be produced.

In the following the structures are given, which describe images with the help of turtle graphics. The first two representation forms are very simple, but the used techniques and results can be applied on more complex structures also.

2.3 CHAIN CODE

The simplest chain code is a nonempty string over alphabet $\{N, S, E, W\}$. This means drawing unit-length lines in the direction North, South, East, and West. The interpretation of this string with a turtle is the $+(\alpha)F(1)-(\alpha)$ command sequence, where $\alpha = 0, 180, 270, 90$ for N, S, E, W respectively. $-(\alpha)$ is needed if we want the turtle to always look in the direction of North. An advanced version of this code is replacing the symbols with the α values. This means a vector of angles is used instead of a string of symbols. The most general way is to have a vector of (α, ℓ) pairs, which are interpreted as $+(\alpha)F(\ell)-(\alpha)$ command sequences. An interpretation $+(\alpha)F(\ell)$ is also possible. In this case α will be relative to the current heading.

Since the chain code can be considered as a real vector, any type of real-vector Evolutionary Strategies (ES) or Genetic Algorithms (GA) can be applied on it. The problem with this representation is that changes at the beginning of the chain code result a huge change in the represented image. In contrast to that changes at the end yield a small change in the image. This problem can be reduced by using strategy parameters for ES, or by restricting evolution to the elements not too far from the end.

2.4 BRACKETED CHAIN CODE

In order to draw not only chains but branching structures two special symbols ('[' and ']') can be inserted into a chain code. These represents the *push* and *pop* turtle commands as discussed previously. So a bracketed chain code is a chain code with added brackets in a well-nested way.

Definition 3 A string S is called *well-nested* if and only if for any S' prefix of S, the number of '[' symbols is not less than the number of ']' symbols, and the number of these symbols is equal in S.

Example 1 Given a bracketed parametric string $A(1)[A(2)[A(3)]A(4)]A(5)A(6)$. The branching structure, or *drawn tree* defined by the sequence can be seen in Figure 1.

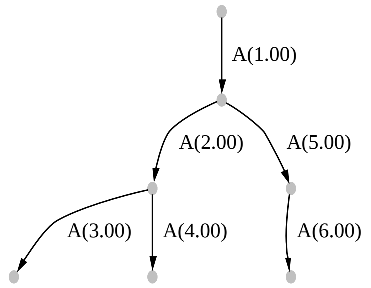


Figure 1: Example for Branching Structure

One type of evolution what may be applied on bracketed chain code is the real-vector evolution mentioned at chain code. However bracketed chain code describes a more complex structure, not only a chain, but a tree, so Genetic Programming (GP) can also be applied on it. One possibility is to use the drawn trees, and apply the common GP tree operators on them. The other possibility is the application of the syntactical trees.

2.4.1 Syntactical trees

For bracketed chain code, or bracketed strings the notion of syntactical trees can be defined as follows.

Definition 4 The *syntactical tree* for a bracketed string u is the following graph $G = G(V, E)$.

- $V = V_s \cup \{u\}$, where V_s is the set of all well-nested substrings, not having a separation into nonempty well-nested substrings.
- $xy \in E \iff x \prec y$ and $(\forall y')(x \prec y' \Rightarrow y \preceq y')$, where $\prec$ is the substring relation.

Example 2 The syntactical tree for the string given in Example 1 can be seen in Figure 2.

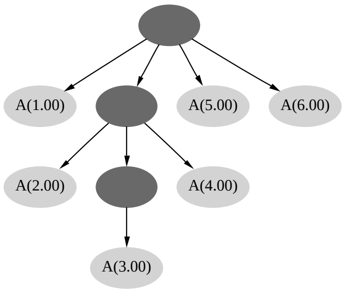


Figure 2: An Example for Syntactical Tree

2.5 LINDENMAYER SYSTEMS

Lindenmayer introduced special kinds of descriptive grammars in order to describe the natural development processes of plants [7]. These are called Lindenmayer grammars or *Lindenmayer systems* (L-systems). The most important characteristic of L-systems compared with Chomsky grammars is *parallel rewriting*. Different L-system classes are introduced, but in this paper only *deterministic context free L-systems* (*D0L-systems*) are used.

However, in order to enlarge the number of the described objects, L-systems are associated with parameters [10]. This means parametric strings are rewritten. In a rewriting rule for a given symbol on the left side variables may occur. These get values when matching the symbol during the rewriting process. The right side of a rule may contain not only variables, but also expressions, which are evaluated using the current values of the variables.

Chain code and bracketed chain code can also be given by an appropriate parametric $D0L$-system, so in the following only parametric $D0L$-systems will be considered. This way the interpretation of chain code is easy, since it is done by the L-system. Also using sequences may be "coded" into the rules. This means instead of restricting the genetic operators to produce a string of allowed sequences only, new symbols are inserted, which represent the allowed sequences. The operators work on strings of these symbols. The resulted strings are turned into "real" turtle operators by several fixed rewriting rules in the L-system.

There have already been several experiments applying GP on L-systems, such as works by Koza [6], Jacob [3], and many others [8], [9]. However they used either a restricted L-system, or a non-pixel based fitness. In most of these works the shapes were opened or closed lines, and not branching structures. The fitness function was usually counting the matching lines, that is the destination image was already in vector format and not in bitmap.

3 EVOLUTION PROCESS

If the evolved objects are chain codes, we can simply apply ES. If they are branching structures, the situation is more complicated, since tree operations may also be used. In this case changing the parameter values may be problematic. Parameters can be considered as nodes in the tree, but even in that case, their value will not change using a simple tree operator. On the other hand, changing them is very easy using ES. Therefore one possible solution is to combine these two methods, namely ES and GP, and apply them on the parameters and on the structure respectively.

However parameter vectors with different lengths cannot be recombined using ES. At least it does not make sense in our case. Thus ES must be restricted to descriptions representing the same structure. This can be considered as dividing the parameter vectors into subpopulations. The interaction between these populations, that is changing the structure is done by GP. Direct interaction between individuals of different subpopulations is not possible. This gives the structure of the population and the evolution process, shown in Figure 3 and in Algorithm 1.

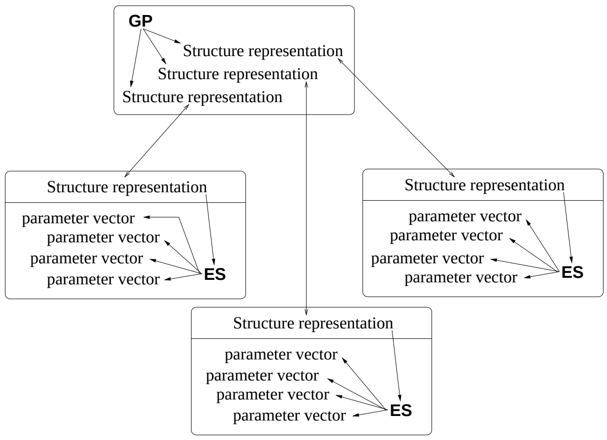


Figure 3: Structure of the Population

```
repeat
  create random population
  for all individual do
    create random parameter vectors
    repeat
      for all parameter vector do
        insert the parameters into the structure
        draw image and calculate fitness
      end for
      apply ES operators on the parameter vectors
    until some halting criteria for subpopulations
    select some parameter vector /* e.g. best */
    set fitness equal to the fitness of this vector
  end for
  apply GP operators on individuals
until some halting criteria
```

Algorithm 1: The Outline of the Evolution Process

4 GENETIC OPERATORS

In this section the possible operators are given. Some of them can be applied on parameter vectors, the others are tree operators, which means they can be used for the evolution of the structures.

4.1 PARAMETER EVOLUTION

As parameters are given in a vector, they can be evolved using the common ES operators, namely recombination and mutation. However the recombination could be sometimes inefficient, therefore a modified version is defined, which tries to predict the shape of the fitness landscape.

4.1.1 Fitness weighed recombination

Sometimes it may be assumed, that the maximum is closer to the individual with higher fitness. Therefore a weighed average is taken as the result of the recombination. Given two individuals $x' = x'_1 \dots x'_n$ and $x'' = x''_1 \dots x''_n$ with fitness values $\varphi(x')$ and $\varphi(x'')$. The result of a fitness-weighed intermediate recombination is a vector x, where

$$x_i = \frac{x'_i \varphi(x') + x''_i \varphi(x'')}{\varphi(x') + \varphi(x'')}$$

The other problem occurs when the function is (locally) monotone. In that case the optimum is not between the two individuals. Therefore the operator is defined to give a new individual outside of this interval, depending on the relation between the fitness values.

Given two individuals x' and x'' with fitness values $\varphi(x')$ and $\varphi(x'')$, such that $\varphi(x') < \varphi(x'')$. The new individual x will be the following

$$x_i = 2x''_i - \frac{x'_i \varphi(x') + x''_i \varphi(x'')}{\varphi(x') + \varphi(x'')}$$

This is very similar to the gradient increment method, but it has an interesting property. Namely this slows down changing as the fitness is changing more intense, which is useful when the fitness landscape changes dramatically.

4.2 STRUCTURE EVOLUTION

For the structure evolution several tree operations are defined. These operators may be applied on drawn trees, and on syntactical trees as well, as the representation requires.

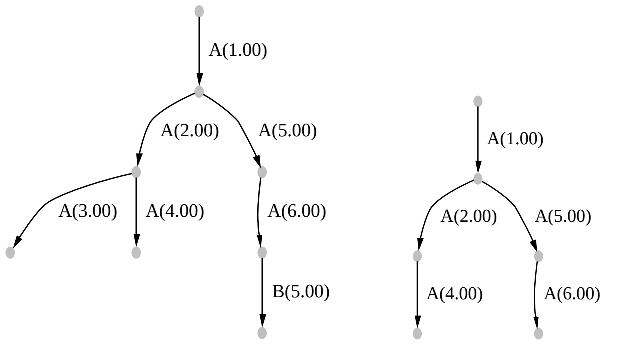
As crossover operator, the common GP crossover is used, which exchanges randomly chosen subtrees. Considering mutation, adding or deleting a subtree is usual, but here two additional mutations are also defined, which exploit the properties of the used trees. The four types of mutations are

a-mutation Adding a subtree (Figure 4a. and 6)

d-mutation Deleting a subtree (Figure 4b. and 7)

b-mutation Breaking an edge (Figure 5a. and 8)

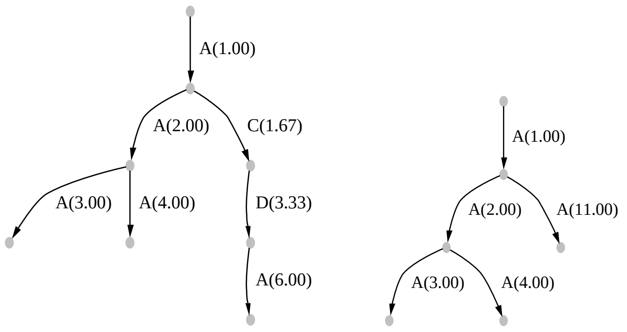
c-mutation Joining two edges (Figure 5b. and 9)



a.) Adding a subtree b.) Deleting a subtree

Figure 4: Simple Tree Mutations

When trees have graphical interpretation, the following holds. After breaking an edge, that is inserting a new vertex, the position of the new vertex in space is randomly chosen. Joining, that is deleting a vertex is defined to leave all other vertices in their original place. The figures show also the difference between drawn tree mutations and syntactical tree mutations. While the first ones operate on edges, the latter ones change nodes.



a.) Breaking an edge b.) Connecting edges

Figure 5: Advanced Tree Mutations

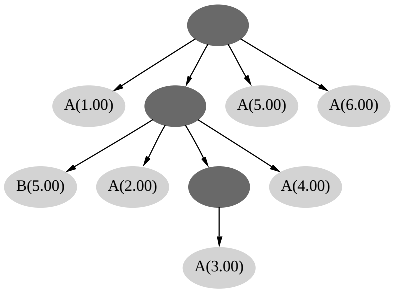


Figure 6: Syntactical Tree Mutation, Type A

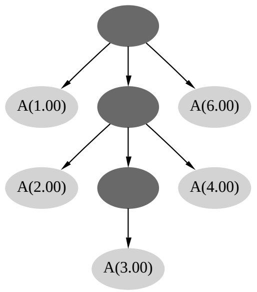


Figure 7: Syntactical Tree Mutation, Type D

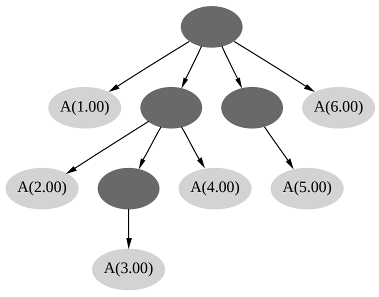


Figure 8: Syntactical Tree Mutation, Type B

An important fact is that using these mutations any tree can be produced. Consider $\mathcal{A}, \mathcal{D}, \mathcal{B}$ and $\mathcal{C}$ to be the set of all possible mutations of type a, d, b and c respectively. The following lemma is the corollary of a well-know fact from graph theory, that a tree can be built from the empty tree by adding pendant edges.

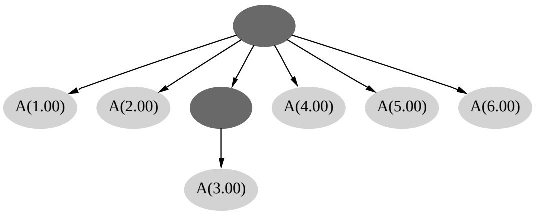


Figure 9: Syntactical Tree Mutation, Type C

Lemma 1 Given two trees T_1 and T_2. Then *exists* a sequence of mutation operators $\sigma_1, \sigma_2, \ldots, \sigma_n$ such that $\sigma_i \in \mathcal{A} \cup \mathcal{D}$ and $\sigma_n(\ldots\sigma_2(\sigma_1(T_1))) = T_2$

But why do we need b and c mutations? Because they give the possibility to make the tree more similar to the destination tree with fitness non-decreasing steps, so it is more probable that the process finds these steps.

Claim 1 Given a pixel-based fitness function φ and given two individuals T_1 and T_2 such that $\varphi(T_1) \leq \varphi(T_2)$. Then there exists a *monotone non-descending* trajectory in the fitness landscape from T_1 to T_2

5 FITNESS FUNCTIONS

In this paper the fitness function is always a pixel-based local additive fitness, since this type of fitness can be calculated during the drawing time.

Definition 5 Given an image container I and two images: $d : I \to \mathbb{R}$ and $g : I \to \mathbb{R}$. Φ is a pixel-based local additive fitness if it has the following form:

$$\Phi(d, g) = \sum_{(x,y)\in I} -error(d(x, y), g(x, y)),$$

where $error : \mathbb{R}^2 \to \mathbb{R}$ is the *comparator*, or *error function*. This fitness function is local since the fitness value of a pixel depends only on its own values.

In our case image d is constant, it will be the *destination image*. It must be compared with different *generated images*. As it can be seen from the definition, this type of fitness can be given by defining the error function.

5.1 SIMPLE AND MODIFIED QUADRATIC ERROR

A usual error function is the so-called quadratic error, where $error(i, j) = (i - j)^2$. In case of binary images the error is $-1, 0$, or 1, so the quadratic error can be 0 or 1. This error function can be modified by distinguishing the different errors, and this way we get the most general form of a comparator function for binary images.

$$error(i,j) = \begin{cases} W_0 & \text{if } i = j = 0, \\ W_1 & \text{if } i = j = 1, \\ W_2 & \text{if } i = 0, \text{ and } j = 1, \\ W_3 & \text{if } i = 1, \text{ and } j = 0, \end{cases}$$

where $W_0, \ldots, W_3$ are predefined constants.

5.2 MULTI-LEVEL FITNESS

The problem with quadratic error is that it does not say anything about the structure of the destination image. If i=0 and j=1 we know that we have drawn a pixel to a wrong place. But is it far from the shape? To take this distance into consideration using a local fitness, the destination image is modified as follows.

$$d'(x,y) = D(d,x,y),$$

where $D(d,x,y)$ is the distance of (x,y) and the set of 1 pixels in d. D is usually the Euclidean distance, but the simpler Manhattan distance can also be used. After this modification, the error function will be the following

$$error(i,j) = \begin{cases} W_0 & \text{if } i > 0, \text{ and } j = 0, \\ W_1 & \text{if } i = 0, \text{ and } j = 1, \\ i & \text{if } i > 0, \text{ and } j = 1, \\ W_3 & \text{if } i = 0, \text{ and } j = 0. \end{cases}$$

Note that if $d(x,y) = 1$ then $D(d,x,y) = 0$.

5.3 DISTANCE BASED FITNESS

Using the idea from the previous fitness function a more sophisticated version can also be given, where the fitness is a function of the distance. The destination image is modified as previously and the error function is the following

$$error(i,j) = \begin{cases} W_0 & \text{if } i > 0, \text{ and } j = 0, \\ W_1 & \text{if } i = 0, \text{ and } j = 1, \\ W(i) & \text{if } i > 0, \text{ and } j = 1, \\ W_3 & \text{if } i = 0, \text{ and } j = 0. \end{cases}$$

By defining $W(0) = W_1$ this function can be simplified:

$$error(i,j) = \begin{cases} W_0 & \text{if } i > 0, \text{ and } j = 0, \\ W(i) & \text{if } j = 1, \\ W_3 & \text{if } i = 0, \text{ and } j = 0. \end{cases}$$

5.4 ADDING BORDER PATROL

Sometimes two lines are so close to each other that the error will not prevent the individual to cross this gap. Therefore parts of the destination image may not be covered by the algorithm at all. To avoid lines crossing empty areas the skeleton of the inverse image is selected, and the image is preprocessed again as follows

$$d''(x,y) = \begin{cases} \infty & \text{if } (x,y) \in skeleton(1-d) \\ d'(x,y) & \text{otherwise} \end{cases}$$

Definition 6 The *skeleton* is a one-pixel thick representation of the object, such that it goes through the middle of the image, and preserves its topology.

5.5 OVERWRITING PIXELS

It can come often, that some pixels are drawn more than once. To handle this case an additional *checking image* $c: I \rightarrow \{free, visited\}$ is introduced, where *visited* means that this pixel has already been drawn. Thus the error function $error(i,j)$ can also be extended to $error_o(i,j,k)$, where the penalty for overwriting is $error_o(i,1,visited)$.

6 HANDLING THICK LINES

When thick lines are also allowed there might be several problems. One problem is overwriting. When two rectangles connect at the middle point of one edge, because of the rotation, some parts may be overlapping. That must not be counted as an overwrite.

Therefore, a slightly *modified checking image* c' is used, where $c': I \rightarrow \{free, exhausted, prohibited\}$. Also the drawn pixels are divided into two groups. One is the *center line*, the other is the set of the "thickening", called *extension pixels*. The calculation scheme can be seen in Table 1, where $\tilde{c}'(x,y)$ is the new value of the checking pixel. This table shows that for example if an exhausted pixel is overwritten then it is changed to prohibited, and some error will be given, but not as much as for normal overwriting.

type	$c'(x,y)$	$\tilde{c}'(x,y)$	**fitness**
main line	free	prohibited	full gain
main line	exhausted	prohibited	some error
main line	prohibited	prohibited	huge error
extension	free	exhausted	some gain
extension	exhausted	exhausted	nothing
extension	prohibited	prohibited	some error

Table 1: Fitness for Thick Lines with Overwriting

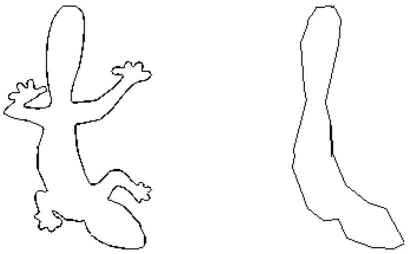
Some tests have already been done, but another problem was found when thick lines were evolved directly. In that case, a huge thick line may cover fine structures. To solve this, the evolution could be divided into two parts. First the skeleton of the image is taken, and a thin line evolution is made. In the second part the evolved structure is not changed any more, only the parameters describing the line width.

7 EXPERIMENTAL RESULTS

To test the method, several experiments were made.

7.1 CLOSED LINE

One experiment was to evolve the outline of the GECCO gecko (Figure 10a.). For the first run the legs were removed, which made the problem easier. The resulted object can be seen in Figure 10b.



a.) GECCO gecko b.) Generated image

Figure 10: Result of the First Test

The fitness value with respect to the time can be seen in Figure 11. The running times for a certain population are shown in Figure 12.

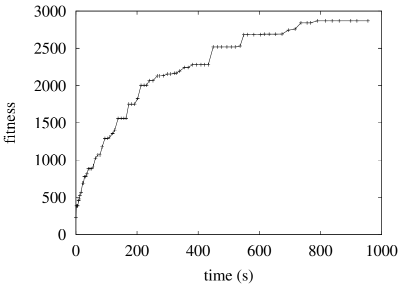


Figure 11: Fitness Value of the Best Individual

The test was also done without removing the legs. First subpopulations with 250 vectors were used. Next time they were smaller, having only 100 vectors. The latter run made the description faster, however sometimes inaccurate. (Figure 13-15)

7.2 BRANCHING STRUCTURE

The third experiment was done using a branching structure. Several tests were run in the different stages of development. Run 0-6 used the simple tree

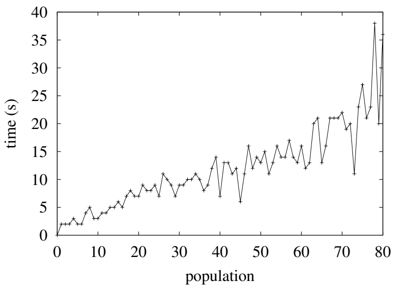
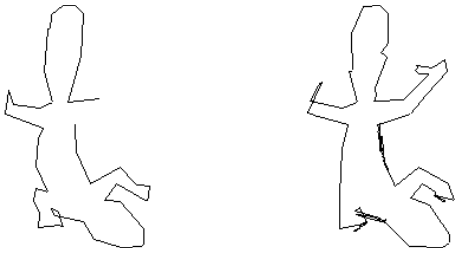


Figure 12: Calculation Times



a.) large subpopulation b.) small subpopulation

Figure 13: Results of the Second Test

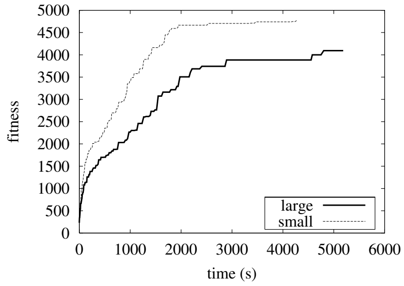


Figure 14: Fitness Values of the Best Individuals

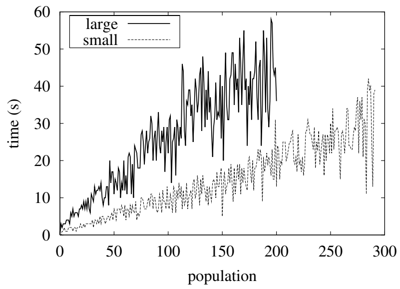


Figure 15: Calculation Times

operations, 7-9 applied also the advanced operators. Additionally run 9 used distance based fitness with $W(i) = i^2$. All other runs used $W(i) = i$. The test image can be seen in Figure 18b. The fitness values are shown in Figure 16., and some generated images in Figure 17 and 18a.

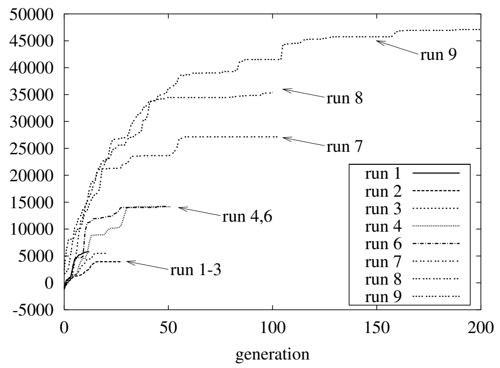
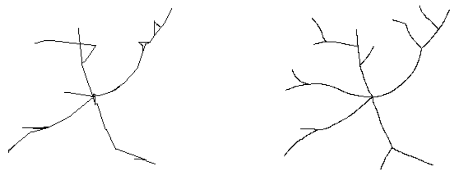


Figure 16: Evolving a Branching Structure

a.) run 0 after 111 steps b.) run 7 after 103 steps

Figure 17: Results for Branching Structure



a.) run 9 after 200 steps b.) Original image

Figure 18: Better Results for Branching Structure

8 CONCLUSION, FUTURE WORK

In this paper a method for evolving structural descriptions of binary images was presented. Possible solutions and improvements were given, such as

- Various genetic operators
- Two-level evolution process
- Fitness functions with several extensions

The results show that this method can be applied to describe shapes, however some refinements will be useful to make. It is among our future plans to implement skeleton calculations, since the biggest problem is the covering of the small gaps. Having the skeleton, some tests need to be done using thick lines as well. In our research the correct parameter settings were not considered either. It is also a topic for the further examinations. Since the chain code and the L-system can easily be extended into three dimensions, it would be interesting to do some experiments in that area too.

References

[1] H. Abelson and A. A. diSessa. *Turtle Geometry.* MIT Press, Cambridge, 1982.

[2] P. Collet, E. Lutton, F. Raynal, and M. Schoenauer. Polar IFS + parisian genetic programming = efficient IFS iverse problem solving. *Genetic Programming and Evolvable Machines*, 1(4):339–362, Oct. 2000.

[3] C. Jacob. Genetic L-system programming. In *Proceedings of the Parallel Problem Solving from Nature, International Conference on Evolutionary Computation, PPSN III*, volume 866 of *Lecture Notes of Computer Science*, pages 334–343. Springer-Verlag, Berlin, 1994.

[4] C. Jacob. *Principia Evolvica – Simulierte Evolution mit Mathematica.* Dpunkt Verlag, 1997.

[5] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, Massachusetts, 1992.

[6] J. R. Koza. Discovery of rewrite rules in Lindenmayer systems and state transition rules in cellular automata via genetic programming. In *Symposium on Pattern Formation (SPF-93), Claremont, California, USA*, 1993.

[7] A. Lindenmayer. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, 18:280–315, 1968.

[8] K. J. Mock. Wildwood: The evolution of L-system plants for virtual environments. In *Proceedings ICEC 98*, pages 476–480. IEEE-Press, 1998.

[9] G. Ochoa. On genetic algorithms and Lindenmayer systems. In *Proceedings of the Parallel Problem Solving from Nature, International Conference on Evolutionary Computation, PPSN V*, volume 1498 of *Lecture Notes of Computer Science*, pages 335–344. Springer-Verlag, Berlin, 1998.

[10] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag, New York, 1990.

[11] R. Ványi, G. Kókai, Z. Tóth, and T. Pető. Grammatical retina description with enhanced methods. In R. Poli, et.al., editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 193–208, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.