An Engineering Approach to Evolutionary Art

J.I. van Hemert
jvhemert@cs.leidenuniv.nl

M.L.M. Jansen
mjansen@cs.leidenuniv.nl

June 19, 2001

Abstract

We present a general system that evolves art on the Internet. The system runs on a server which enables it to collect information about its usage world wide; its core uses operators and representations from genetic programming. The output consists of images that are decoded from tree structures. We show how this general system can be used to evolve two types of art: A Mondriaan like art and a type known as mandala. Both types are implemented with the mind of an engineer.

1 Introduction

In evolutionary art we let a set of images, i.e., pieces of art, evolve using principles from evolutionary computation. We generate new pieces of art using genetic operators. At the same time a user of our system will steer the course of evolution by influencing the selection process. In evolutionary computation terminology: Our user represents the fitness function

For a user to be able to select images she likes we need an interface that shows her all possible choices. In a previous study we created a C++ program with a graphical user interface (van Hemert and Eiben, 1999). Here we adopt a different approach by removing this strong coupling, replacing it with a much more flexible web interface.

The general system as presented here can be applied to evolve any two dimensional picture limited by memory and by representation as binary trees. We have chosen two forms of art to show how the system works. First, an abstract style that mimics art by a Dutch painter named Mondriaan. Second, a less well defined form of art known as mandala.

We outline the rest of the paper. First we explain how our system fits in the field of evolutionary art. Then we present the general system from the viewpoint of a user in Section 3. In Section 4 we take a look at the inner workings of the system. Two examples show how the general system can be put to work in Section 5. Then, in Section 6 we state our findings and we finish with future remarks in Section 7.

2 Evolutionary Art

In evolutionary art we strive for a system that creates art using the principle of evolution: The survival of the fittest, or in this case, the survival of the most beautiful. Often this goal is achieved using an evolutionary algorithm of some form. Many different types of art have been created this way, starting about ten years ago with Karl Sims (Sims, 1991). Nowadays, some people have turned it into a business (World, 1996). All systems share a common feature: Human intervention to determine what is nice and what is ugly, in other words, a human fitness function.

A variety of art created by evolution in the computer exists from visual to audio, but here we restrict ourselves to two dimensional visual art. When examining other systems that also produce visual art we notice that these systems are often based on fractals. Just by looking at galleries of pictures produced by these fractal driven system we get a feeling for the immense size of the space of possible pictures. The way these systems work resembles an artificial life approach: By creating simple rules incredible complex behaviour, in this case pictures, can emerge. We take the opposite approach by first visualising what our art should look like, then we take genetic programming and try to map its tree structures such that we achieve our art. The role of the user stays the same: To search for pieces of art that are aesthetically pleasing.

Most evolutionary art systems run on a single machine, which, in itself, is not a striking property, but the main restrictions these systems have is that they interact solely with the person behind the same machine. Even if such a system would be popular its output will not go beyond the user and her machine. Here we strive for a system that is accessible for many people at the same time, gathering information about the decisions these people make. All of this is made possible through the Internet and the common gateway interface (CGI).

The fact that the system is on-line all of the time helps us, the researchers, to get the assistance of many people. This type of research is based largely on subjective decisions. To be able to make statements on these decisions we require a large amount of data. Normally we would have to search actively for subjects that are willing to assist in these experiments, but here they can voluntarily and anonymously visit the page and use the system.

3 Art on the Web

When you tell your browser to surf to one of the art pages of the system you will be presented with a set of images that have been created randomly, similar to the screen captured in Figure 1. To the left of this set, let us call this your *breeding ground*, is another set that is smaller. This smaller set is the *top selection*, which we will discuss later. The images in your breeding ground are all laid out on the page and every one of them is accompanied by a button. The buttons hold the key to the selection process whereby you can steer the evolutionary process of creating new images.

You can enable or disable the button that belongs to an image. This action is equivalent to stating if you like or respectively, do not like a particular image. By pressing `generate new paintings` the system is told to generate a new set of paintings using the button settings of the user. In a nutshell, this is the main functionality of the system.

After a request for new paintings, the screen that is presented has two important features. First, the paintings that had their button enabled reappear at the same location, with their button still enabled. Second, all the other paintings are replaced by new paintings that, in most cases, look similar to the paintings with enabled buttons. The latter is precisely why a user is able to steer towards images that have properties she likes.

At any time it is possible to change the number of images presented. If the number is increased new images randomly generated are added. Decreasing the number of images just results in deleting current images from the bottom.

The way in which the images are stored internally makes them suitable for presentation in any size. On the main page a user can view images using five scales. By clicking on one of the images a new page is loaded with the chosen image presented in a large scale that will fill a browser's window in most cases.

The top selection represents the most often selected images by everyone that has been playing around with the system. Your selections are also used to update this set of stored images, thereby influencing the dynamic way in which these images are selected for presentation to you and other users.

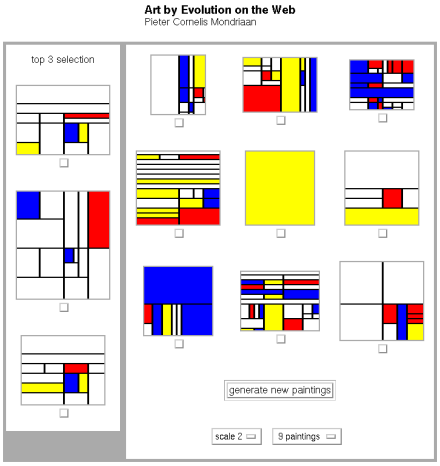


Figure 1: Screenshot taken from a browser with the system producing Mondriaan like art

4 Behind the Scenes

The core of the system is developed in Perl using two modules. The first module, Simple Common Gateway Interface Class (CGI), handles all the Internet related functions. The second module, Database Independent Interface for Perl (DBI), provides the link between the system and the database where we store images, more precisely we store the coded versions. The whole system, Perl and MySql database, runs on a Pentium II class machine with Linux as its operating system.

Basically, the core of the system consists of two parts. First, an evolutionary algorithm that creates new images according to a user's preferences. Second, a database that is used to store images and related information. Below we present first the evolutionary algorithm followed by a coding scheme that we need, then we explain the usage of the database.

4.1 Evolutionary Algorithm

The evolutionary algorithm used here is a crossing between a genetic program and a generic evolutionary algorithm. Most of its features are from genetic programming (Koza, 1992), but it does not share the main paradigm of creating executable material that can be applied to many different inputs. However, the system can easily be extended to run on input data if required, thereby making it fully compliant to genetic programming.

The genotype consists of a n-ary tree and two integers. The tree is used to create the composition and the colouring, while the integers provide the dimensions of the final image. Although the exact meaning of the elements in the tree depends on the type of art that we want to create we can safely say that in most cases the terminal and function set both have a separate role. The elements of the function set are used to create a composition, i.e., a placement of objects, whereas the terminals are used to determine objects to be placed.

The integers of the genotype determine the width and height of the canvas on which we are about to create an image. The actual image is created by doing a pre-order walk of the tree, creating the composition and objects therein recursively. The phenotype is then a two dimensional image. The precise mapping from genotype to phenotype is determined by the type of art we want to create. Two different types shall be discussed in Section 5.

To initialise the genotype we have to perform two steps. First we determine the value of the two integers by generating two uniform random numbers from the integer domain $\{10, \ldots, 17\}$. Then we generate a tree with a maximum depth of six using a method based on the grow method (Koza, 1992). Every time we generate a node we first determine of which type, function or terminal, it will be. The chance that the node is selected from the function set is $1/\textit{current depth}$. The rationale here is that the chance of creating nodes is higher at the top, while at the bottom we have more chance of creating leafs. Eventually at the maximum level we only create leafs.

We use two genetic operators, both taken from genetic programming. Crossover is used to create one offspring out of two parents by swapping two subtrees

from the parents. Mutation is a bit different than in the literature, we use a single point mutation as described in (Banzhaf et al., 1998), but we mutate every node in the tree with a chance of $1/$*number of nodes*. Besides the tree we also pass on the dimensions. With equal chance offspring receives the height of one of the parents, we repeat the same procedure for the width. Then we change the offsprings dimensions by choosing randomly a number from $\{-2, -1, 0, 1, 2\}$ and adding it to the dimension. We enforce that the resulting values lie in the interval $\{1, \ldots, 30\}$.

The fitness function is induced by a user. Basically, she determines the parent set directly because the fitness function is boolean. She either likes or does not like an individual which is translated into the parent selection process as a chance to become parent or no chance at all to become a parent. This is similar to the deterministic selection process in evolution strategies.

The parents are selected randomly from those individuals that have been selected by a user. The other individuals are all replaced by offspring created using the two aforementioned genetic operators. If there is only one parent, no crossover is performed and all the other individuals are replaced by randomly generated ones.

4.2 Flat Trees

As our system runs over the Internet we need to examine some of the aspects of the evolutionary algorithm. Normally in an evolutionary algorithm we have, at any time during a run, a certain state of all the data structures. This can be implicit, such as the number of evaluations performed, or explicit, the current population. Our system is stopped after each generation, showing its state as images on a web-page. If the user wants to continue the algorithm for another generation the system will need information on its current state. The passing through of the current state happens using CGI variables. These variables contain values for each member of the population and a couple of visual attributes, such as the number of images and the scale at which images are shown.

A problem hides in using CGI variables as they can only hold strings. Our system uses n-ary trees to do its evolutionary computing. Thus we require a certain coding and decoding of our tree structures, but as our system is running online a user could be waiting for output. Thus our coding and decoding needs to be fast. Instead of dealing with this we circumvent the problem by never dealing with trees as we are used to. We deal with the trees coded as a linear structure, never representing them as a pointer structure. This is one of the five different approaches to programming trees Keith and Martin looked at (Keith and Martin, 1994).

We call a linear coded tree a *flat tree*. Let us look at what kind of operations we might need to perform on a flat tree. First of all we will have to be able to generate them randomly using the initialisation method described above. Very important for the decision of our coding is the mapping that we want to use. In the two art forms in this paper we benefit most from a pre-order walk. Lastly, we need to perform genetic operations on the structure. Taking these

constraints into consideration we have opted for a pre-order walk of the tree, where we process the node as we visit it (Knuth, 1968).

Generating a random flat tree is not much different from a pointer version. Instead of creating the pointers we just output the node using a recursive function that first creates the root of a subtree, then the children from left to right. Decoding the tree to an image is trivial as we want to do a pre-order walk thus we need to run through the structure linearly. Mutation is also easily performed linearly, using the length as the number of nodes. Crossover however, is more complicated.

To perform a subtree exchange we first have to identify where the subtrees lie inside the linear structures. Algorithm 1 shows a simple way of finding the index that corresponds to the right most leaf of the subtree. Basically, all that we need to remember while walking through the linear structure from left to right is how many leafs we still need to complete a subtree. Every time we advance to a node we decrease this number with one and, in the case of an internal node, we increase this number again by the arity of the function. As soon as we reach zero we have a whole subtree. An example of the result of this process is in Figure 2.

Algorithm 1 Finding the end of a subtree in a flat tree that represents a pre-order walk

Require: t is a flat tree and s denotes the start of a subtree of a n-ary tree
Ensure: s is now the index of the end of the supplied subtree

```
if t[s] is a function then
  n ⇐ arity(t[s])
else
  n ⇐ 0
end if
while n ≠ 0 do
  s ⇐ s + 1
  n ⇐ n − 1
  if t[s] is a function then
    n ⇐ n + arity(t[s])
  end if
end while
```

Performing a crossover is nothing more than choosing a random node inside each parent as usual (s_1 and s_2), followed by running Algorithm 1 on both parents with these random nodes to find the end of both subtrees $e_1 =$ `end-subtree`$(parent_1, s_1)$ and $e_2 =$ `end-subtree`$(parent_2, s_2)$ and finally we swap the linear pieces from both parents $parent_1[s_1 \dots e_1]$ and $parent_2[s_2 \dots e_2]$.

Our mutation operator looks at every element in the structure and changes it with a chance of $1/size$. If an element is to be changed we replace it by an element randomly chosen of the same type (terminal or function). Also, functions are always replaced with functions of the same arity to make sure that the tree structure stays intact.

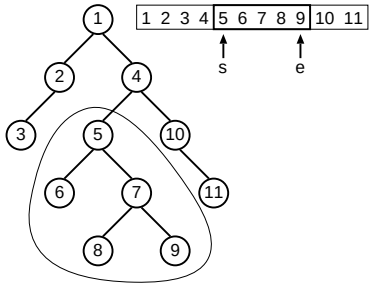


Figure 2: A tree (left) coded as a flat tree (right) with the same subtree selected in both

The decoding of a flat tree is straightforward by recursion. The linear structure is already a pre-order walk of a binary tree and that is exactly what we need in the applications described later. The elements in the tree have almost no meaning to the core system, in that all it knows is how to distinguish functions from terminals, which is necessary to maintain correct tree structures. The semantics of these elements are determined by the decoding algorithm, which we describe in Section 5.

4.3 An On-line Gene Bank

The system presented so far does not provide us with much information as all of the interaction is between one user and the system, which is not stored. Thus, as soon as a user is bored and decides to point her browser to another page, we loose something valuable as users are hard to find and it is even more difficult to keep their attention. If we want to gain information on what the average person likes and dislikes, we need to record the select actions of many visitors. Hence, we connect the system to a database that stores the paintings that are seen by visiting users.

Our database runs on MySql, a free database product that is designed to be fast. As we have shown in the previous section our individuals are represented by flat trees. These are essentially unique strings that can be stored without any conversion into the database. We store every picture presented to a user whenever she initiates a new generation of the evolutionary algorithm. An individual is stored into a table corresponding with its type of art. The definition of this table is in Table 1. Besides the flat tree and the dimensions we also store the time and date when the individual was first inserted and when it was last updated. Most important are the good and bad entries, because these number represent the number of times a painting has been selected, respectively, not selected.

Whenever we encounter an individual that is not present in the database we call upon Algorithm 2. This is the moment when we store the flat tree and dimensions, as well as set the creation date and time. If the individual is

Table 1: Table in database that stores the selected paintings, the database keys are in bold font

name	*definition*	*description*
id	varchar(250)	chromosome (flat tree)
good	bigint(8)	number of times selected
bad	bigint(8)	number of times seen, but *not* selected
size_x	int(1)	width of painting
size_y	int(1)	height of painting
changed	timestamp	last time selected
created	datetime	first time selected

currently selected we set $a = 1$ and $b = 0$, otherwise we set $a = 0$ and $b = 1$. In other words, we give the first point for good xor bad.

Algorithm 2 Storing a new individual

insert into *table* (id, good, bad, sizex, sizey, created)
values (*flat_tree*, *a*, *b*, *size-x*, *size-y*, now())

Every time we do a step of the evolutionary algorithm we update all the individuals that are in the current population and already present in the database by updating the good or bad value. We increase its good value by one if the individual was selected or its bad value by one if it was not selected. The database system automatically sets its time stamp. Algorithm 3 shows how an individual's record is updated.

Algorithm 3 Updating the good value of an existing individual. Similar algorithm in case of updating the bad value

update *table*
set good = good+1
where id = *flat_tree* **and** sizex = *size-x*
and sizey = *size-y*

We present the top selection of individuals using Algorithm 4. Every one of them can be selected as an additional parent. Another possibility is to browse through the whole set of stored individuals of a type. First all the paintings are sorted in the same way as when selecting the top x and then we present them 25 per page. A seperate page on the web is used to browse through the whole database.

Algorithm 4 Retrieving the top x individuals sorted first by -good then by bad and then by the inverse of the changed time stamp

select id, sizex, sizey
from *table*
order by -good, bad, -changed
limit x

5 Two Art Forms

We show two different types of art that the system currently produces. Besides these, currently two other types can be produced. One abstract style similar to the paintings of Doesburg, an artist who had much influence on abstract art in the Netherlands in the first half of the 20th century. Another style of art that has been explored using evolution by Karl Sims (Sims, 1991) and later by Tatsuo Unemi (Unemi, 1999), where we evolve equations that calculate the colour of a pixel using the pixel's coordinate as input.

The images are created using the Gd Graphics Library (GD) for Perl. This library outputs images as Portable Network Graphics (PNG).

5.1 Mondriaan Art

Pieter Cornelis Mondriaan (1872–1944) (Deicher, 1995) is considered one of the most prominent 20th century geometric painters. He is also known under the name Mondrian because in 1910, when he moved to Paris, he discarded his father's name, replacing it with Piet Mondrian. Mondriaan's art started out as portraits of real life, most notably landscapes. These portraits gradually became more abstract, conveying an idyllic and calm scene. After he returned from Paris in 1914 he would only produce art in the spirit of pure form. Later, he did a number of important contributions to the magazine "De Stijl", wherein Mondriaan published twelve chapters on his vision on new art. Around 1921 he entered the final stage of his style by restricting his pallet to the colours yellow, blue, red, white and black, while at the same time creating geometric shapes using only straight black lines that intersect at right angles, which have become, more or less, a trademark for Mondriaan art.

We will attempt to generate paintings from Mondriaan's last period. These paintings share common properties that make them easy to define inside a computer. First, only a few colours are used. Second, horizontal and vertical black lines are used that start and end at the edge of the canvas or at a perpendicular black line. Third, planes that can be constructed with these black lines are filled entirely with one colour.

A painting is represented as a tree structure and two integers for the height and width. A tree consists of nodes taken from a function set and leafs taken from a terminal set. The functions will provide the composition by splitting up the canvas, while the terminals will fill in the colours of the composition. Hence, our function set exists of horizontal `H` and vertical `V` splitters and our terminal

set exists of Mondriaan's pallet: white, yellow, blue and red. When decoding a tree to a painting we start out with the whole canvas and, if the root of the tree is taken from the function set, split the canvas either horizontally or vertically. In the same way we continue with the two halves of the canvas. Eventually we will encounter a terminal at which point we determine the colour we use to fill in the corresponding part of the canvas. An example of such a decoding is in Figure 3. This representation is also used in the work of Schnier and Gero (Schnier and Gero, 1998).

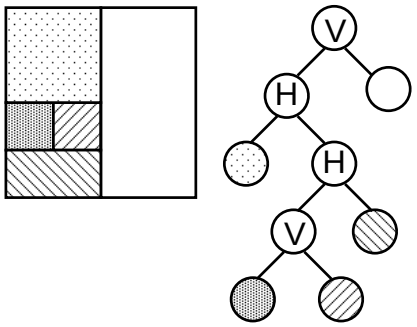


Figure 3: Representation of the phenotype (left) and the genotype (right) of Mondriaan like art

5.2 Mandala Art

Mandala is the Sanskrit word for circle, and it is used to symbolise wholeness. Mandalas can be found in a lot of places. They are used mostly for meditation and thus are part of several religions, but they also show up in science (Jung, 1968; Gontar, 2000) and in art. There is a direct link with several natural phenomena, and some examples of these are: the solar system, the planet earth, spider webs and flowers. All of these show the basic characteristic of a mandala: a circle with a unifying centre.

As said above, mandalas are usually round figures. In addition, there is usually also a high degree of regularity, which makes them ideal to be used as an art form in this system. Everybody is able to make a mandala, since it is a reflections of a person's thoughts. This means that there is a lot of freedom in creating mandalas. To make the task feasible for use in an evolutionary algorithm, we have to make some simplifying assumptions, and all of these will be reflected in the terminal and function sets.

The function set defines only one type of node, which we will call split nodes. There are three of them, the only difference being the exact location where the split is made. Splitting means dividing an area in two parts. Secondly, the terminal set defines only two types of nodes, with each type having several instances which define different parameters. The first type defines the so called *solid fill* nodes, where each node assigns a colour to a specific area. The second

type defines what we will call *fill type* nodes, where we define a specific area to be filled using a pattern of lines, where the appearance of a pattern is defined by three parameters: the number of lines in the fill pattern, the angle at which the first point of a figure is drawn, and the colour in which a pattern is drawn. Every tree that represents an individual is of type binary, with nodes from the function set as interior nodes, and nodes from the terminal set as leafs.

The mapping from a tree to a picture is as follows. The tree is represented as a flat tree, as described earlier. We perform a pre-order walk of the binary tree by traversing the linear structure from left to right. We start with a given circular area of maximum radius, and the first node (if this is not a leaf) cuts this area in two parts, where we repeat the procedure for both parts. By proceeding this way, we end up with a circular area sliced up into several rings. For every ring there is a corresponding leaf, and this leaf defines the way in which we will fill the area. One point must be made clear here. Although every area is actually ring-shaped, when we assign a colour to an area, we mean that we fill the area from the outer boundary up to the centre, and not only up to the inner boundary. This also implies that, for every interior node, its left subtree must define the outer area and the right subtree the inner area. Figure 4 shows the result of such a mapping.

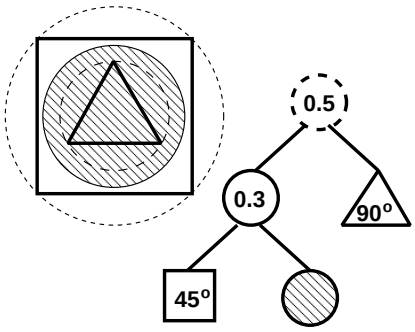


Figure 4: Representation of the phenotype (left) and the genotype (right) of mandala type art, the dotted lines are not shown in the final result

6 Conclusions

We presented a general system to evolve abstract art using tree based data structures taken from genetic programming. The evolutionary and interface part are linked with two simple examples of abstract art through a general representation.

The system runs on a machine hooked up to the Internet and it uses the common gateway interface to interact with humans. This opens up the system and our research to a large pool of people that can explore the search space of our implemented forms of abstract art.

To get the system to work through the common gateway interface we have introduced flat tree structures with operators that work directly on them. Furthermore, as these flat trees are essentially linear structures we can easily transfer them to other systems, such as a database, where we can store them.

To learn what people find aesthetically pleasing we have extended the system with memory. This memory is a database that stores every piece of art ever seen by a user. Also, it keeps track of how often a piece is selected and not selected. This database now contains a couple of thousend of pieces of art.

Although the database of selected images is constantly changing we provide the piece of art that is currently the most often selected. Figure 5 shows a Mondriaan like images which is ranked high in the gene bank. An often selected Mandala is in Figure 6.

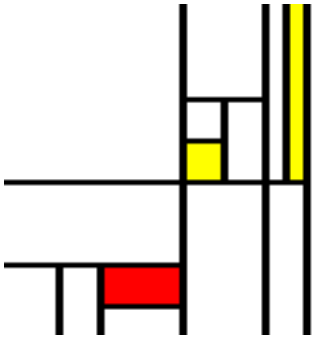


Figure 5: The most often selected Mondriaan image

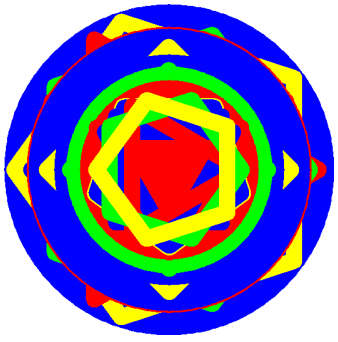


Figure 6: The most often selected mandala

We invite the reader to visit our system on the Internet to fully experience the dynamics. Its address is

`http://www.liacs.nl/~jvhemert/eartweb/`

7 Future Research

Now that we have a general system that handles the evolutionary and interface part work starts on a more general notion of the decoding. This is a small step towards the idea of Steven Rooke of having an open standard for coding individuals. We start by extending the ways of producing abstract art presented here into a more general form.

The uniform way of storing and handling the representation of the individuals makes it possible to exchange individuals between types of art. It would be a nice extension if the system could show how an individual looks decoded into different forms of art, thereby letting it compete in different environments at once. The simplest approach is to use one table to store the individuals of every type of art. The measurements of good and bad will have to be adjusted to include for which type a certain individual was counted. Hopefully this leads to individuals that are fit under more than one environment giving us a general composition that people on average prefer.

As mentioned earlier, it is expensive and time consuming to perform experiments that require subjective responses by humans. As our database is gradually expanding we hope to devise a way of using the accumulated data to automate the process of assigning good and bad values. Furthermore, we could create a more refined fitness function using the counters good and bad.

Eventually we hope to link structures of our system to objects from our physical world, such as the layout of a document or web page. This idea has been used in work of Thorsten Schnier (Schnier, 1999) where an evolutionary algorithm is used to create the floor plan of a building.

Careful observation of the description of the evolutionary algorithm reveals that we are not talking about a genetic program as defined by Koza (Koza, 1992). The individuals are decoded to images, but every time the result will be the same. For now there is no way of providing input to an individual such that it will create a different image.

A Mandala Art in More Detail

The goal is to create an algorithm that uses genetic programming to create mandala figures. Since a computer can not decide what looks nice or not, we need the help of a human being. She selects images that please her, and based on this selection the algorithm creates offspring that hopefully pleases her more. All of this is done using common concepts from evolutionary computation, such as crossover and mutation. We refer to (Koza, 1992) and (Banzhaf et al., 1998) for an introduction. If we want all of this to work we must find a way to represent mandala figures in the computer. The (internal) representation, and the way to transform this representation into an actual figure, are explained in the following subsection.

A.1 Data structure

Every individual (coding of a figure) must be coded using some convention. Here we use binary trees to code individuals. This coding into binary trees is done to allow easy use of methods such as crossover and mutation. Note that every binary tree that is constructed represents a valid figure. A binary tree is one where the bottom nodes are leafs, all other nodes are interior nodes and every interior node has exactly two children. For an interior node the left subtree defines the outer area, and the right subtree defines the inner area of that node. Figure 7 shows an example.

Below we will describe the different types of interior nodes and leafs. We will also define the function and terminal sets

A.1.1 Interior nodes and function set

The interior nodes divide an area at a given point, hence the name 'split', see (1). Their function is described here and Figure 7 illustrates the process of splitting areas.

The **function set** is:

$$\mathrm{F} = \{\text{split.3, split.5, split.7}\} \tag{1}$$

An area is a ring-shaped part of a circle. Note that the root of the genotype of the individual in Figure 7 is an interior node. All individuals have a pre-determined dimension, so the figure has a certain maximum radius, which encloses the entire figure. What happens now is that the root node cuts the area from the center to the border of the figure at a certain distance between these two. The exact point at which the cut is made depends on the particular type of interior node. The result of it all is that we end up with two new areas, which together form the entire area. The so called inner area is still a circle, though with a smaller radius than before cutting (its radius is the same as the length from the center to the cutpoint). The outer area is ring-shaped, with inner radius equal to the radius of the inner area and outer radius equal to the maximum radius. Both areas can be subdivided further in the same way. This is shown in the lower part of Figure 7 for the left subtree of the root node.

All nodes from (1) split areas, the only difference being the location where the cut is made:

split.3 splits the corresponding circular area into two parts, at $\frac{3}{10}$-th the distance from the inner boundary

split.5 the same, but for $\frac{1}{2}$ the distance

split.7 the same, but for $\frac{7}{10}$-th the distance

If we split several times, the circle we start with is divided into several ring-shaped parts (with the exception of the innermost area, which is a circle). Every part has two boundaries: an inner boundary and an outer boundary. For our

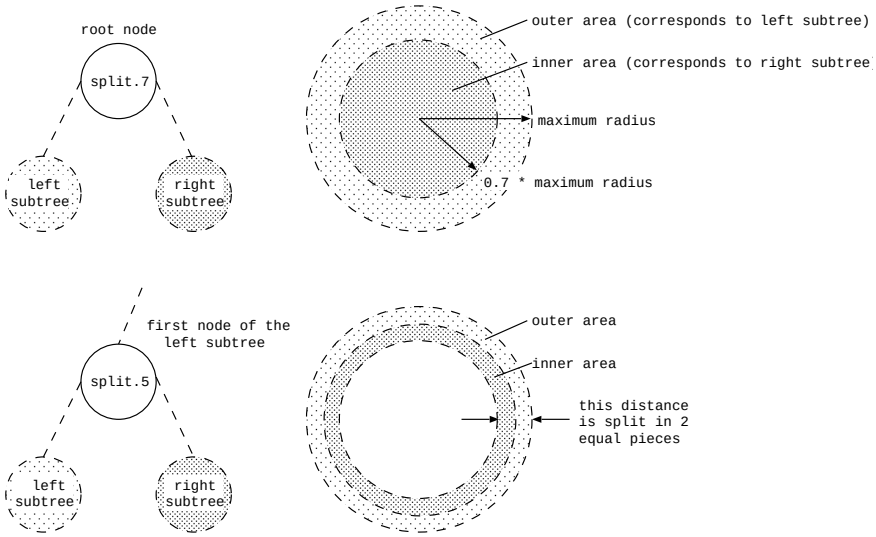


Figure 7: Some examples of splitting an area

discussion of the leafs, it is easier to think of every area as being a circle, with radius equal to the outer boundary.

A.1.2 Exterior nodes (Leafs) and terminal set

A leaf belongs to one area. Leafs determine in what way the corresponding areas are filled.

The **terminal set** is:

$$\mathrm{T} = \{\mathrm{Uniform}\langle color\rangle, \mathrm{Fill}\langle n\rangle\langle color\rangle\langle angle\rangle\} \tag{2}$$

where

- color $\in$ {Red, Green, Yellow, Blue}
- n $\in \{3, 4, 5\}$
- angle $\in \{0, 45, 90, 180, 270\}$

Possibilities are to fill an area with a certain color, or to apply a filling pattern to the area. Adopting the view defined at the end of the previous subsection, the filling will run from the outer boundary up to the center. This will only work if we start filling the areas from the biggest one down to the smallest one, which can be achieved by performing a left to right, depth first tree-traversal, as will be shown later.

The two different types of leafs can thus be described as:

Uniform <color> Fills the corresponding circular area using the given color.

Fill <n> <color> <angle> Draws a regular figure of n lines in the corresponding circular area, using the given *color*. We start drawing at an offset of *angle* degrees. For this we must agree on a convention for the positive direction of rotation, as well as on the start (an angle of 0 degrees). This is depicted in Figure 8.

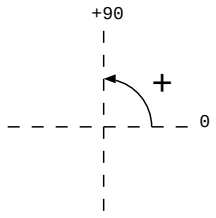


Figure 8: convention for the rotation axes

In the case of a Fill node, not all combinations of n and *angle* are allowed, because this would lead to cases where the distinction would not be noticeable. For example, in the case where n=4, *angle*s of 45 and 90 would yield identical figures. Figure 9 gives an overview of the viable combinations.

n	*angle*
3	0, 90, 180, 270
4	0, 45
5	0, 90, 180, 270

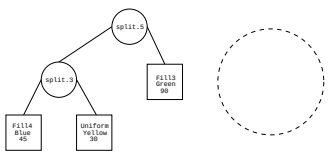
Figure 9: viable combinations of Fill node parameters

So we have 4 types of leafs for uniform fills, and 40 types of leafs that define filling patterns, which adds to a total of 44 different leafs.

A.2 Transformation of genotype to phenotype

I will now show how to derive the phenotype from the genotype of an example individual. The genotype of this individual is shown in Figure 10. Remember that a genotype corresponds to the way in which figures are coded internally, and that a phenotype defines the actual mandala figure, i.e. it is a real picture. The translation from genotype to phenotype is done by performing a left to right, depth first tree-traversal. This is done so that the resulting figure is constructed from outside to inside, which is required because in this way the different parts of the image are always drawn in the right order.

1. In the first step we have not examined any nodes. We know however that the figure fits within a predefined area (which is indicated by a dashed circle: the maximum radius).



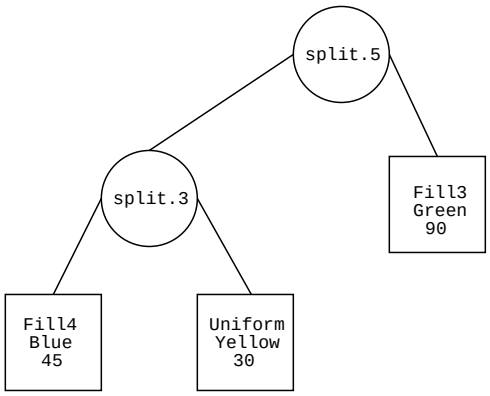
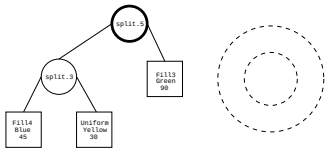
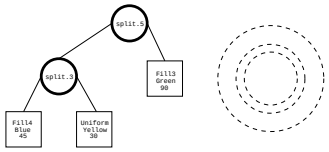


Figure 10: The genotype of the example individual

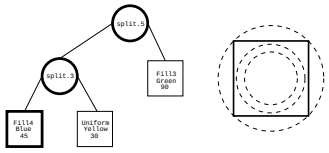
2. We now start examining the binary tree. The root node is of type `split.5`, so we must divide the area into two parts, at half the distance from center to the outermost boundary (the dashed circle of step 1). This new boundary is also shown as a dashed circle.



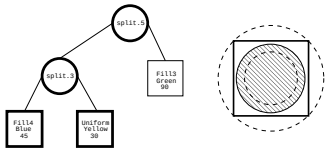
3. The next node is of type `split.3`, which means that the outer area (which runs from the dashed circle of step 2 to the dashed circle of step 1) is split at $\frac{3}{10}$-th this distance, measured from inside to outside.



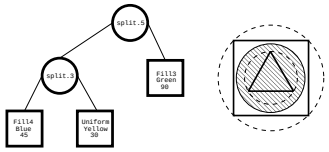
4. This node (which actually is a leaf) is the first one to do some drawing. It draws a blue square inside the outermost area. The first point of the square is drawn at an angle of 45^o. Remember that only leafs do any actual drawing.



5. This leaf again does some drawing. This time it paints the inner area of the `split.3` node yellow.



6. The resulting leaf draws 3 green lines in the inner area of the root node, starting at an angle of 90^o.



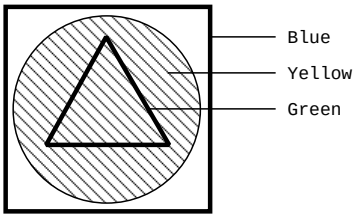


Figure 11: The corresponding phenotype

Figure 11 shows what the resulting mandala figure looks like. Note that if all leafs would have been of type Fill, the three filled circles that emerged would have been drawn in the right order. This explains why the left subtree of a node defines its outer area, and why performing a left to right, depth first tree-traversal yields a correct phenotype.

A.3 Initialization method and maximum depth

Now that we have defined both the terminal and the function sets, it is fairly straightforward to create some random trees to start the algorithm with. We will use a method based on the grow method (Koza, 1992). Every time we generate a node we first determine of which type, function or terminal, it will be. The chance that the node is selected from the function set is $1/\textit{current depth}$. The rationale here is that the chance of creating nodes is higher at the top, while at the bottom we have more chance of creating leafs. Eventually at the maximum level we only create leafs. We must still decide on a maximum depth parameter (to keep the size of initial individuals within certain bounds), and a value of six seems reasonable.

A.4 Genetic operators

We will use crossover and mutation in this algorithm, and both have their distinctive features.

A.4.1 Crossover

The crossover method randomly selects a subtree in each parent and swaps them. One important feature of the crossover operator used here, is that it only produces one offspring. So in effect, one of the children is thrown away.

A.4.2 Mutation

Mutation works as follows for all nodes in the tree. With a certain chance ($1/$*number of nodes*) we will change the node. If a node is changed, we randomly select a new node of the same type. So an interior node will be replaced by an interior node (this works since all nodes from the function set have an arity of 2), and a leaf is replaced by a leaf. This type of mutation is called point mutation (Banzhaf et al., 1998).

A.4.3 An example of crossover and mutation

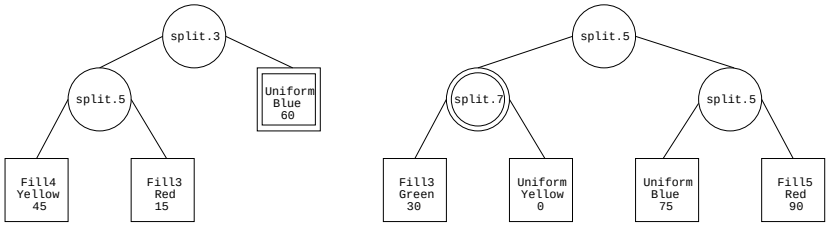


Figure 12: Two parents that are selected; the crossover points are indicated using double lines

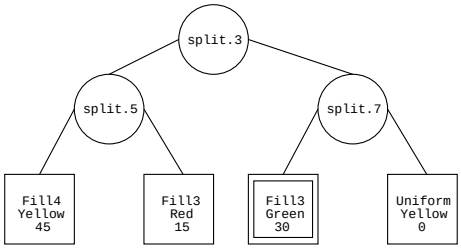


Figure 13: After crossover; one node is selected for mutation (double lined)

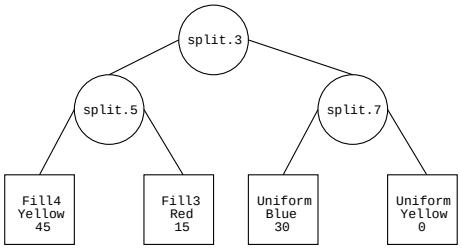


Figure 14: The resulting child

Figures 12–14 illustrate both genetic operators. In Figure 12 we see (the genotypes) of two parents. The crossover operator is applied, and the result is shown in Figure 13. Next, the mutation operator is applied. Note that the node that is selected for mutation is a leaf (Figure 13). This leaf is replaced by another leaf, which we choose randomly. In this case only two of the parameters are changed. We switch from a filling pattern of 3 green lines to a uniform blue area for the corresponding area.

A.5 user equals fitness function

The main difference between the approach we use here and the 'standard' approach is that the fitness function is not an integral part of the algorithm. Rather, for selection to take place, we need the help of a human being (the user). She plays a significant role in every generation of the evolutionary process. She decides, for every individual in a population in a certain generation, whether the individual is fit or not. The user therefore defines the fitness function, which is of type boolean.

A.6 Selection method

For each crossover event, both parents are randomly selected from the set of selected individuals. This set is composed by the user, since all individuals that are selected end up in this set.

References

Banzhaf, W., Nordin, P., Keller, R., and Francone, F. (1998). *Genetic Programming: An Introduction*. Morgan Kaufmann.

Deicher, S. (1995). *Mondrian*. Benedikt Taschen Verlag GmbH, Köln.

Gontar, V. (2000). Theoretical foundation of jung's mandala symbolism based on discrete chaotic dynamics of interacting neurons. *Discrete Dynamics in Nature and Society*, 5(1):19–28.

van Hemert, J. and Eiben, A. (1999). Mondriaan art by evolution. In Postma, E. and Gyssens, M., editors, *Proceedings of the Eleventh Belgium/Netherlands Conference on Artificial Intelligence* (BNAIC'99), pages 291–292.

Jung, C. (1968). *Collected works of C. G. Jung*, volume 12. Princeton University Press, 2nd edition.

Keith, M. and Martin, M. (1994). Genetic programming in C++: Implementation issues. In Kinnear Jr., K., editor, *Advances in Genetic Programming*, chapter 13, pages 285–310. MIT Press, Cambridge, MA.

Knuth, D. (1968). *The Art of Computer Programming, Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, Mass.

Koza, J. (1992). *Genetic Programming: On the Programming of Computer by Means of Natural Selection*. MIT Press.

Schnier, T. (1999). *Evolved Representations and Their Use in Computational Creativity*. PhD thesis, Key Centre of Design Computing, Department of Architectural and Design Science, University of Sydney, NSW, 2006, Australia.

Schnier, T. and Gero, J. (1998). From Frank Lloyd Wright to Mondrian: Transforming evolving representation. In Parmee, I. C., editor, *Adaptive Computing in Design and Manufacture*, pages 207–219. Springer Verlag, Berlin.

Sims, K. (1991). Artificial evolution for computer graphics. *Computer Graphics*, 25(4):319–328.

Unemi, T. (1999). SBART2.4: Breeding 2D CG images and movies, and creating a type of collage. In *The Third International Conference on Knowledge-based Intelligent Information Engineering Systems*, pages 288–291.

World, L. (1996). Aesthetic selection: The evolutionary art of steven rooke. *IEEE Computer Graphics and Applications*, 16(1).