Evolving Fractal Art with a Directed Acyclic Graph Genetic Programming Representation

Daniel Ashlock and Jeffrey Tsang

Abstract—A class of fractals called orbit capture fractals are generated by iterating a function on a point until the point’s trajectory enters a capture zone. This study uses a digraph based representation for genetic programming to evolve functions used to generate orbit capture fractals. These variations on the genetic programming system are examined using two fitness functions. The first fitness function maximizes the entropy of the distribution of capture numbers, while the second places a geometric constraint on the distribution of capture numbers. Some combinations of representation and fitness function generate fractals often, while others yield interesting non-fractal images most of the time.

I. INTRODUCTION

Fractals are objects that have non-integer dimensions. There are a large number of different algorithms for producing fractals. Fractals, in forms such as snowflakes or trees, have intrigued and excited people from artists to mathematicians, from naturalists to engineers, for centuries. Use of the formal term fractal to describe this sort of complex object was introduced by Mandelbrot [15] in the 1980s and ever since then we have become increasingly concerned with understanding, using, and creating fractals. Soon after the popularization of fractals, researchers became intrigued with the idea of using one biologically inspired system to control another resulting in the creation of fractals through the use of evolutionary computation. Fractals have been evolved for a number of different purposes, but the research into the evolution of fractals can be loosely divided into the evolution of fractals for artistic purposes, the evolution of Iterated Function Systems (IFSs), and the evolution of L-Systems.

A. Evolved fractals as art

Algorithmic generation of complex, beautiful images is facilitated by using approximations of fractal objects as the basis of the code. Algorithms that render portions of the Mandelbrot set [2], for example, have access to an effectively infinite number of possible images and so the augmentation of human search capabilities, and even automation, made possible by evolution are valuable tools. Newton’s method, best known as a tool for finding the roots of a mathematical expression [20], can also be used to generate orbit capture fractals. An example is shown in Figure 1.

Julia sets were the target of [4], [6] which used an evolutionary algorithm with user specified fitness functions in order to locate aesthetically pleasing fractals. Ashlock [1] used evolutionary computation to search for “interesting” (defined in several different fitness functions as complexity and approximating aesthetic pleasure) visualizations of the Mandelbrot set. Evolution was also used to create tiled mosaics composed out of evolved fractals[10].

Modified versions of the Mandelbrot set [23], called “tweaks”, were evolved, first with human-in-the-loop evolution and then against reference images. The “tweaks” to the Mandelbrot set were largely the addition of real number variables to the equations used to visualize the set in a 2D-plane. The resulting images show everyday objects through the lens of the Mandelbrot set.

B. Evolving L-Systems

L-systems are a model of the development of plant morphology. An example of a simple, two-dimensional L-system appears in Figure 1. L-systems were first proposed by Aristid Lindenmayer in 1968 as a means of modeling the patterns of growth in plants as a mathematical system [13]. In [11], it was demonstrated that L-systems could be evolved using genetic programming. A fitness function was used which proposed two bounding cubes and rewarded the formation of “leaves” in the outer cube, while penalizing them in the inner cube.

Traxler [22] used a genetic algorithm with human-in-the-loop fitness evaluations to evolve L-systems to generate appealing fractal plants. This human-in-the-loop evolution of fractal plants was expanded on by [9] through the addition of a more complex environment for the plants to evolve in.

Moving away from interactive evolutionary approaches, [18] showed that plant-like evolved L-systems could be constructed using a fitness function derived from the principles which are presumed to guide the evolution of biological plants.

C. The Fractal Inverse Problem

Barnsley initially introduced the idea that fractals could be used to encode images, by finding the correct iterated function system (IFS) [7]. This statement introduced the “inverse problem”: for a given image, find an IFS that approximates the image; this could be used as an image compression technique. Barnsely used affine functions $A\vec{v}+\vec{c}$ as his iterators. As the inverse problem is in the complexity class NP, many researchers have tackled it with evolutionary techniques, evolving fractals to try to fit known images. Lutton [14] used genetic programming to evolve non-affine iterated function systems, while Poli [19] did the same.
with affine IFS. The snowflake-like image in Figure 1 was generated with an affine IFS.

D. Other uses of evolved fractals

In addition to these three primary areas of interest, other occasional uses for evolved fractals have been explored. Bentley [8] used subsets of the Mandelbrot set as “fractal proteins” in artificial cellular systems, demonstrating that networks of interactions between evolved fractal proteins degraded more gracefully than others. DNA sequences were visualized in [3] using a representation called Chaos Automata.

The remainder of this study is structured as follows. Section II specifies the representations and fitness functions used to evolve orbit-capture fractals. Section III gives the design of experiments. Section IV gives results and discusses them while Section V draws conclusions and outlines possible next steps.

II. Representation and Fitness

We begin by specifying the type of fractal evolved in this study. An orbit-capture fractal requires a function $f : \mathbb{C} \rightarrow \mathbb{C}$ from the complex plane to itself. In effect this is a pair of functions, each of two variables $(f(x, y), g(x, y))$, that transforms a point $(x, y)$ into another point $(u, v)$. A capture set $S$, which is a region in the plane, is also required. To determine membership of a point in the fractal, the function is applied repeatedly to the point to generate a sequence, the orbit. If the sequence never contains a point in $S$ then the point is in the fractal. If the point is not in the fractal then capture number of the point is the first time that the orbit is within $S$. In practice we cannot compute the entire sequence and prove it will never contain a point in $S$, so an iteration limit $L$ is selected. If after $L$ iterations, no point in the orbit hits $S$, then the original point is assumed to be in the fractal. The function used to generate the sequence is what is evolved in this study. The capture set used was:

$$S = \{(x, y) : x^2 + y^2 > 4\},$$

the exterior of a circle of radius two. The iteration limit was set to $L = 512$, a large but manageable value.

The most common representation for genetic programming is a parse tree that specifies a formula [12]. This representation has a fundamental problem: if a computed value is needed in multiple places then we need a separate subtree to compute it each time it is used. We can get around this problem by shifting to a directed acyclic graph (DAG) representation to permit a value computed at one point in the data structure to be used at multiple points in the data structure. Figure 2 shows an example of a DAG representation.

![Fig. 2. A function stack in tabular and directed graph form.](image)

Definition 1: A **function stack** is a list of nodes numbered $0, 1, \ldots, n-1$. Each node contains an operation and two arguments, one of which may not be needed. The arguments can be ephemeral constants, one of which is available in each node, any of the input variables $x_1, x_2, \ldots, x_k$, or the output of nodes with a higher index number. The output of the function stack is the value of the lowest index node.

Function stacks are a type of Cartesian genetic programming [16], [17]. An example of a evolved function stack
is shown in Figure 2. Tree-based genetic programming most commonly uses subtree crossover, which is highly disruptive. This can be useful when a high degree of exploration is required, but also leads to bloat, a phenomena where the population of trees pack on unused code which protects them from crossover-based disruption. The use of a fixed size linear data structure like a function stack completely avoids bloat and permits the use of crossover operators suitable to strings or arrays, which, while still potentially disruptive, are often less disruptive than subtree crossover.

It remains to specify the functions and inputs used to evolve function stacks in this study. These are given in Table I. The \textit{wav} function computes the weighted average of the first argument \(x\) and the second argument \(y\) as

\[ Wav(x, y) = \alpha x + (1 - \alpha)y \]  \hspace{1cm} (2)

Where \(\alpha\), the weight, is the fractional part of the node’s ephemeral constant.

\begin{table}[h]
\centering
\caption{Functions and inputs available to function stacks.} \label{table:func-inputs}
\begin{tabular}{ll}
\hline
\textit{Inputs} & \\
\hline
\(X_i\) & Input variable \\
\(L_i\) & Reference to the output of another node \\
\hline
\textit{Unary functions} & \\
\hline
Neg & Negates its argument \\
Scl & Scales its argument by the node’s ephemeral constant \\
Sqr & Computes the square of the argument \\
Sin & Computes the sine of the argument \\
Cos & Computes the cosine of the argument \\
Atan & Computes the arctangent of the argument \\
\hline
\textit{Binary functions} & \\
\hline
Add & Adds the arguments \\
Sub & Subtracts the arguments \\
Mlt & Multiplies the arguments \\
Dvp & Perform protected division on the arguments \\
Max & Computes the maximum of the arguments \\
Min & Computes the minimum of the arguments \\
Wav & Computes a weighted average of the arguments \\
\hline
\end{tabular}
\end{table}

The function stack shown in Figure 2 uses the value of its zeroth node as an output. It is possible to retrieve the output of any node of the function stack. We need a function that accepts and returns two variables in order to generate a sequence of points \((x, y)\) in the plane. Function stacks make this easy: we can use the value of the zeroth node as \(x\) and the value of the first node as \(y\). While it is possible to have the computation of \(x\) and \(y\) be via completely distinct functions, it is more likely that this technique will lead to functions that share nodes.

\textbf{A. Three Variations of the DAG Representation}

This study uses three variations of the DAG representation, all of which specify and evolve the function stack in the same fashion. The difference lies in the way they modify the arguments of the function stack each time it is used. During testing it was found that the unmodified function stack representation seldom produced fractals, leading to the design of the two modifications. If \((u, v) = (f(x, y), g(x, y))\) is the function from the plane to itself specified by a function stack then the three representations are derived from this function as follows.

1) The first variation uses \((f(x, y), g(x, y))\) without modification. This often generates non-fractals and is included for comparison.

2) The second variation uses

\[(f(x^2 - y^2, 2xy), g(x^2 - y^2, 2xy))\]

as the function. If we think of \((x, y)\) as the complex number \(z = x + yi\) then we are squaring that number before applying \((f, g)\) to its real and complex parts, that is, we take \((f(z^2), g(z^2))\).

3) The third variation is similar to the second except that a constant perturbation is included:

\[(f(x^2 - y^2 + 0.26, 2xy), g(x^2 - y^2 + 0.26, 2xy))\]

This replaces complex squaring of the point before applying \((f, g)\) with the complex function \(g(z) = z^2 + 0.26\); as before, \(z = x + yi\). This function corresponds to an iterator that produces the Julia set shown in Figure 3.

The second and third representations are small modifications of the first, placing a header onto the function stack representation. This header is based on complex squaring that forces the resulting function to be nonlinear.

\textbf{B. Fitness Functions}

\textbf{Definition 2:} The \textbf{capture number} for a point \(z\) in the complex plane, under the influence of an iterator \(I\), is the number of times \(I\) must be applied to transform \(z\) into a member of the capture set.

Two fitness functions were used. Both are based on sampling the capture numbers for a function stack on a regularly spaced grid of points in a square subset of the plane with corners \((-1.6, -1.6)\) and \((1.6, 1.6)\). An iteration limit is used to deal with the fact that some points are never captured; a point not captured within the limit is assigned
the limit as its capture number. The iteration limit \( L = 512 \) places these numbers in the range \([0, 511]\) or, if the value is 512, marks the points as being in the fractal. The first fitness function, entropy fitness bins the iteration numbers less than 512 into 64 equally spaced bins. The bin counts are normalized to obtain an empirical distribution of capture numbers. The fitness function is:

\[
fit = \frac{E}{m+1}
\]

where \( E \) is the Shannon entropy of the distribution and \( m \) is the number of points marked as being in the fractal. If \((p_1, p_2, \ldots, p_k)\) are the non-zero values in the empirical distribution then

\[
E = -\sum_{i=1}^{k} -p_i \cdot \log_2 (p_i)
\]

Maximizing entropy while penalizing points that land within the fractal encourages the function stack to generate a picture in which the fractal itself occupies a modest portion of the image and in which the iteration numbers are as evenly distributed as possible.

The second fitness function, mask fitness, is drawn from [5]. To compute this function, the capture numbers, including the values of 512, are normalized by dividing by 512 to obtain real numbers in the unit interval \([0, 1]\). This fitness function aims to match the grid of normalized capture values to the corresponding values from the function

\[
h(x, y) = \frac{1}{1 + 3x^2 + 3y^2}
\]

and penalizes root-mean square error of the mismatch. The graph of the target function is a hill centered on the origin, which encourages high iteration numbers near the center of the sampling window and small ones at the edges. In the earlier study this fitness function excelled at locating minibrots, small copies of the Mandelbrot set within the Mandelbrot set. An example of a minibrot is shown in Figure 4.

C. Making Pictures

To render a picture, the iteration numbers, or fractal set membership, of a 960 \( \times \) 960 grid are computed. The points correspond to pixels in the image with members of the fractal are rendered in white and other points colored according to a periodic pallet indexed by their iteration number, which is given by

\[
\begin{align*}
Red &= \frac{(\cos(0.138t + 0.8) + 1)\%}{2} \\
Green &= \frac{(\cos(0.127t + 0.8) + 1)\%}{2} \\
Blue &= \frac{(\cos(0.092t + 0.2) + 1)\%}{2}
\end{align*}
\]

where \( t \) is the capture number and RGB intensity is normalized to \([0, 1]\).

III. DESIGN OF EXPERIMENTS

A collection of 30 independent runs of the evolutionary algorithm were performed for all three variations of the representation and for both fitness functions. The evolutionary algorithm used was steady state [21] using size seven single tournament selection. In this model of evolution, seven members of the population are selected and the two most fit reproduce, replacing the two least fit members of the tournament with the children. Reproduction uses two point crossover of the list of nodes of function stacks and from 1–3 mutations with the number of mutations selected uniformly at random. Mutation changes the operation in a node 40% of the time, one of the arguments of the node 40% of the time, and adds a Gaussian random number with mean zero and standard deviation 0.02 to an ephemeral constant 20% of the time. The functions stacks evolved were 12 nodes long.

A population of 400 function stacks was used. Fitness evaluation on 441 sample points for up to 512 iterations is computationally expensive, so evolution was run for 10,000 fitness evaluations, reporting summary statistics every 100 fitness evaluations. The most fit function stack found in each run was saved and rendered as a picture.

IV. RESULTS AND DISCUSSION

We begin with the basic representation that generated pictures like those shown in Figure 5. These pictures satisfy the fitness functions quite well, but they do so by using linear gradients, for entropy fitness, and slightly modified linear gradients for the mask fitness. The first panel of Figure 5 shows what a linear gradient looks like when rendered as an image. The most fit result of the entropy fitness experiment with the first representation is shown in Figure 6. It is the maximum of two linear maps, producing two linear gradients that meet along a line. When mask fitness was used, the function stacks made small modifications of linear gradients.
that fit them to the mask. Of the sixty runs performed with the first representation, only two were not linear gradients or simple modifications of such gradients.

\[
\begin{align*}
L0 & \text{ max} & L2 & L6 \\
L2 & \text{ wav} & L5 & L9 \ \text{wt} \ 0.4224 \\
L5 & \text{ sub} & L9 & X2 \\
L6 & \text{ add} & L11 & L11 \\
L9 & \text{ sub} & X1 & -0.0077 \\
L11 & \text{ wav} & X1 & -0.8962 \ \text{wt} \ 0.1038
\end{align*}
\]

Fig. 6. The most fit function stack for the runs using the first representation with entropy fitness. Nodes not used during fitness evaluation are not shown.

The results with the first representation are examples of an evolutionary computation system doing exactly what it was asked to do, rather than what the designer wanted it to do (designer fail!) The linear gradients are very easy to encode as function stacks: for the entropy fitness function they do an excellent job of distributing the capture numbers evenly and, with the right parameters, avoiding points within the fractal. The fitness of the function stack shown in Figure 6 is 5.89314. With 64 bins used to compute entropy, the maximum possible fitness is \( \log_2(64) = 6 \). While they were slightly less ideal for the mask function, modified linear gradients still were still able to do achieve very high fitness.

A. Results for Modified Representations

The two modified representations were far more likely to generate fractals, and the non-fractal pictures they generated were often interesting. Figure 7 gives exemplary images for both fitness functions using the second representation while Figure 8 does this for the third representation.

The first three images in Figure 7 are examples of interesting but clearly non-fractal images as are images 2, 9, and 11 in reading order from Figure 8. The remaining images have appearances that suggest they are fractal, though no formal proof has been made. These experiments show that both changes in the representation solved the problem of evolving only linear gradients.

While incorporating domain specific information into an evolutionary algorithm or representation is good practice when possible, there is some concern that the third representation incorporates complete directions for making a particular Julia set into the evaluation of the function stacks. The fourth and eighth images, in reading order, in Figure 8 show elements of the Julia set shown in Figure 3. These were the only results that had this feature, so the incorporation of the iterator did not represent too great a hint relative to the fitness function.

The first mask fitness image in Figure 7 has an intriguing flower-like character; two other images in Figure 7 and one in 8 have a similar character. These images are the most different from the types of fractals the authors have produced in their earlier work. There are several obvious visual categories of images. Eight of the images contain what might be characterized as “colored sand”, others incorporate large regions that appear as if they were colored with a gradient fill. Both symmetric and asymmetric shapes appear, showing that the second and third representations are able to generate a substantial diversity of images. The “colored sand” results from connected domains which have a common capture number consisting of one or a few pixels. While it was not tried in this study, this type of fractal often has interesting features if the rendering is zoomed in.

V. CONCLUSIONS AND NEXT STEPS

This study presented three representations and two fitness functions for producing evolved art with a fixed-size DAG representation for linear programming. The second and third representations arose in a successful attempt to repair the first, which generated uninteresting images. The system demonstrates the ability to evolve a broad variety of images with a relatively generic evolutionary computation system. The novelty in the study consists in the representation and fitness functions.
Fig. 7. Examples of fractals evolved with the second representation.
Fig. 8. Examples of fractals evolved with the third representation.
A. Exploring System Parameters

Since this study succeeded in finding fractals that render to complex, visually appealing images, relatively little parameter optimization on factors like population size, mutation type and rate, crossover type and rate were performed. The evolutionary algorithms were also run for a relatively short time and relatively short (an so quickly evaluated) functions stacks were used. Since the objects encoded by function stacks are iterators, the nodes in each function stack are applied to the numbers being manipulated hundreds of times on average. Lengthening the function stacks may do nothing much, it may open up new vistas of images, and, independently of its effects on image character, lengthening may slow the rate of evolution.

The second and third representations used were the result of forcing a complex quadratic function in as the first thing that happens in any function stack. There is immense scope to use other “headers”, different quadratics, cubics, or even transcendental functions. The set of operations available to the function stacks can also be changed or extended. Both these directions open up different and potentially novel spaces of images.

B. Better Rendering

A single rendering technique using a fixed periodic palette was used to produce images of all the evolved function stacks. The examples of other types of fractals in Figures 1 and 4 give a sense of the effect of using other palettes or other rendering methods. Changing, or even evolving the parameters of, rendering techniques is another broad area to extend this work.

The evolution of images in this study depends on a sample grid of points but is independent of the rendering code. Images can be re-rendered to any size permitted by available computer memory by simply sampling the capture numbers of finer meshes of points. An example of this is shown in Figure 9.

REFERENCES


Fig. 9. A larger rendering of the flower life image from Figure 7