Towards a General Framework for Program Generation in Creative Domains

Marc Hull
Department of Computing
Imperial College
180 Queen's Gate
London
SW7 2RH
mfh@doc.ic.ac.uk

Simon Colton
Department of Computing
Imperial College
180 Queen's Gate
London
SW7 2RH
sgc@doc.ic.ac.uk

Abstract

Choosing an efficient artificial intelligence approach for producing artefacts for a particular creative domain can be a difficult task. Seemingly minor changes to the solution representation and learning parameters can have an unpredictably large impact on the success of the process. A standard approach is to try various different setups in order to investigate their effects and refine the technique over time.

Our aim is to produce a pluggable framework for exploring different representations and learning techniques for creative artefact generation. Here we describe our initial work towards this goal, including how problems are specified to our system in a format that is concise but still able to cover a wide range of domains. We also tackle the general problem of constrained solution generation by bringing information from the constraints into the generation and variation process and we discuss some of the advantages and disadvantages of doing this. Finally, we present initial results of applying our system to the domain of algorithmic art generation, where we have used the framework to code up and test three different representations for producing artwork.

Keywords: Automatic program generation, genetic programming, evolutionary art.

1 Introduction

Finding an efficient approach for producing artefacts in a particular creative domain is often more of an art than a science. Many general artificial intelligence techniques exist that could potentially be used with varying degrees of success, but most are so complex that it can be difficult to tell in advance which will perform better than others. They are also heavily dependent on the problem representation used and a number of other parameters that can greatly affect their performance. Typically this can be alleviated by using knowledge of the problem to predict which search strategies will be most successful. However, in creative domains the problem may not be well-defined enough to be an accurate guide.

Our aim is to build a general, pluggable framework where problems can be expressed using a concise syntax and tested with all of the different artificial intelligence techniques written for the system. In this paper, we address the problem of how the user specifies a search space to our system. To cover a wide range of domains and learning techniques, the representation used must be general enough to cover many different data structures, but not too general as to include invalid solutions in the search space. To achieve this, we opted for a tree-based structure, similar to those used in Genetic Programming, with implicit type rules controlling the shape of the tree and explicit constraints to disallow particular node patterns. In later work, we hope to concentrate on evaluating different search strategies and attempting to define classes of domains for which particular techniques work best.

In section 2 we provide the background to our project by describing existing genetic programming tools and some of their applications to creative artefact generation, against which we compare our approach. In section 3 we provide details of the framework in terms of how we split the specification of problems into three concise parts, which enables quick, flexible prototyping of different representations. In section 4 we describe how solutions are generated via an iterative two-stage process that employs explicit user-given constraints about the nature of the programs to be generated in order to avoid producing invalid solutions.

To demonstrate the potential of the framework for productive generation of programs, we have implemented an evolutionary search mechanism where the user acts as a fitness function (a standard approach), and we have applied it to algorithmic art generation. Using the concise problem specification syntax we were able to quickly prototype three different representations for generating programs that produce artwork when executed, namely colour based, particle based and image based approaches. Finally, in sections 6 and 7, we describe the future work planned for the framework and summarise our conclusions drawn so far.

2 Background

Genetic Programming was first popularised by Koza's 1992 book (Koza, 1992) as a way to find programs that optimise a measure of fitness. By manipulating variable-length parse trees that represent Turing-complete programs, it is sufficiently general to be able to represent solutions from many other machine learning techniques (Banzhaf et al., 1998). Many tools and libraries exist for using this technique to evolve programs or program fragments to perform particular tasks, e.g. DGPF (Weise and Geihs, 2006), the Genetic Programming Engine[1] and GALib[2]. Typically, these allow the user to specify a representation containing the ingredients (terminals and non-terminals[3]) to be used in the solutions. The user also supplies a fitness function that evaluates each solution, returning a numeric result indicating its suitability for performing the task. To start with, a set of random solutions are generated using the given representation and these are then evaluated by the fitness function to determine which perform better than others. Roughly speaking, the best performing solutions are then combined with one another in an attempt to form new solutions that produce a higher value when evaluated by the fitness function. This process is then repeated until the required minimum fitness value is exceeded, at which point the solution with the highest fitness value is taken as the final solution.

In the pure GP approach, solutions can be constructed using any combination of terminals and non-terminals provided in the representation. However, often this results in solutions being produced that are not valid in the context of the problem. A simple example is a problem that requires a permutation of values, where any solution that contains the same value twice would be invalid.

A standard GP approach would still allow such invalid solutions to exist, but one remedy to this is to apply additional constraints to the representation to explicitly remove certain terminal and non-terminal combinations from the search space. This then leads to the problem of constrained solution generation and variation, where the technique used for producing and altering solutions must take the constraints into account in order to avoid invalid solutions.

Several approaches for handling constraints in evolutionary techniques have been tested, with the most notable being penalty functions, repair functions and decoder functions. Penalty functions (Baeck et al., 1995) evaluate solutions against each constraint individually and subtract a value from their fitness for every constraint violated. Repair functions (as used in Michalewicz and Nazhiyath (1995)) allow solutions that violate constraints to be generated, but then modify these solutions in an attempt to find the most similar variant that passes all of the constraints. Decoder functions (as in Gottlieb and Raidl (2000)) are employed during the genotype to phenotype mapping phase and ensure that the solution genotype maps on to a phenotype in which all constraints will always be satisfied.

Constraint handling is particularly appropriate when the solution phenotype is a computer program in a high-level language, since such languages often impose many complex constraints such as scoping rules and type compatibility upon their programs. Strongly Typed Genetic Programming helps to alleviate some of these problems by allowing the representation itself to be typed and then modifying the generation and evolution algorithms to only produce type-safe trees (Montana, 1995). However, this is often achieved by tying the GP system to a particular language, such that the rules of that language are implicit in the representation (as in JGAP[4]).

Genetic Programming has also previously been used to evolve programs that produce artefacts from creative domains such as pictures and music. Machado and Cardoso's NEvAr system (Machado and Cardoso, 2002) is a well-known generator of algorithmic art which evolves an algorithm for setting the red, green and blue colour components of each pixel in an image. Johanson and Poli have applied a similar technique to music generation (Johanson and Poli, 1998). Their system produces programs in a custom language that describes how to play chords and when to pause between notes. Many other systems also use automatic program generation to produce creative artefacts, however a full survey of these is beyond the scope of this paper.

3 Framework Details

Our aim is to provide a framework that is general enough to accept problems from a wide range of different domains, yet concise enough to allow for quick prototyping and easy modification. We have chosen a variable-sized tree representation to encode solutions, similar to those used in Genetic Programming, since this is sufficiently general to also encode representations used in other machine learning approaches. However, an overly general representation would include solutions in the search space that may not be valid solutions to the problem being solved. Hence, the framework must also allow users to easily constrain their representations to remove invalid solutions for specific problems.

To allow users to specialise their representations, we use a combination of node type constraints upon our parse trees and logical constraints for removing unwanted node patterns. The former is based upon the constraints of Montana's Strongly Typed Genetic Programming system (Montana, 1995) and allows the type systems of programming languages to be respected. Meanwhile, the latter allows for constraints over node dependencies to be expressed. This can, for instance, be used to enforce that instances of two node types may only exist together and not independently. We have found this to be particularly useful for expressing relationships between function calls and function declarations when evolving programs.

Finally, we also provide a method for translating the

---

[1]A genetic programming library for the .NET framework (http://gpe.sourceforge.net/)

[2]A C++ library of genetic algorithm components (http://lancet.mit.edu/ga/)

[3]These are described as terminals and functions in Koza (1992), but we use the term non-terminal here to distinguish from functions in programs.

[4]An open-source, Java-based genetic algorithms package (http://jgap.sourceforge.net)

tree structure used internally to represent solutions into text output, which is analogous to the genotype-phenotype mapping in Genetic Programming. In our experiments, we use this to convert our solutions into programs, scripts or data that can be accepted by other programs. We then use the behaviour of these programs to evaluate the success of the solution.

To keep the roles of specifying the representation, imposing constraints and compiling solutions to text separate, users provide each of these to our system in a separate file. The following subsections explain how these files work in further detail.

3.1 Representation File

Solutions in our current system are represented by trees whose structure is specified by the user in the representation file. At a basic level, this file allows the non-terminal and terminal node types of the tree to be specified, in a similar way to most other Genetic Programming systems. However, these node types are also involved in typing constraints that allow the structure of the trees to be controlled.

These typing constraints allow the terminals and non-terminals of the representation to exist in an inheritance hierarchy, such that groups of node types that are semantically linked (e.g. `True`, `False`, `And`, `Or`) can inherit from a common node supertype (e.g. `Boolean`) that represents this link. Each non-terminal node type then specifies which arcs it has to each of its child nodes, and also inherits any arcs declared in its supertypes. Each arc is also annotated with node types that restrict the nodes that can be children of it. An arc annotated with a node type `X` will only accept child nodes that are instances of the `X` type or any types that inherit from `X`. Additional features such as abstract node types, primitive types and multi-child arcs are also supported but there is insufficient space here to describe them in detail.

The following shows an example of the syntax used to specify a representation for simple numeric expressions.

```
representation NumericExpressions {
  abstract type NumericExpression;
  type Zero : NumericExpression;
  type One : NumericExpression;
  type Two : NumericExpression;
  abstract type BinaryOperator
                 : NumericExpression {
    NumericExpression left;
    NumericExpression right;
  };
  type Add : BinaryOperator;
  type Sub : BinaryOperator;
  type Mul : BinaryOperator;
  type Div : BinaryOperator;
};
```

3.2 Constraints File

In addition to the implicit typing constraints provided in the representation file, the user can also specify explicit constraints upon the solution trees in the constraints file.

Since the system is not tailored to output in a specific language, this file can be used to add constraints that are specific to the output language for this particular problem. It can also be used to add domain-specific constraints, which in the case of program generation could remove a large proportion of non-compiling and invalid solutions from the search space.

Constraints are currently expressed in a syntax that is based upon first-order logic, but is tailored to expressing conditions about tree structures. The language includes `and`, `or` and `not` operators, which follow their traditional logical semantics, as well as `exists` and `all` operators that have special meanings. In particular, they only match nodes at or within a particular part of the tree, and they can optionally bind these matches to variables that are then used in the evaluation of their sub-expressions.

The following shows how this syntax can be used to express the constraint that `Div` nodes cannot have `Zero` nodes for their `right` children in the representation from Section 3.1.

```
constraint NoDivideByZero {
  all Div in root as divideNode (
    not (
      exists Zero at
      divideNode.right
    )
  )
};
```

3.3 Compiler File

Once a solution tree has been generated, the compiler file is consulted for the transformations required to convert the tree into the specified output language. The user provides these transformations as string templates for each node type which describe how nodes of that type should be represented in the output. The string templates are specified in the Velocity templating language[5], so that references to the compiled output of child nodes are represented by enclosing the child name between `${` and `}` delimiters. This also allows the templates to include control flow constructs like `for` loops over node children.

The following gives the compiler code for translating the numeric expression representation from Section 3.1 into C-style expression syntax.

```
compile Zero [|0|];
compile One  [|1|];
compile Two  [|2|];
compile Add  [|(${left})+(${right})|];
compile Sub  [|(${left})-(${right})|];
compile Mul  [|(${left})*(${right})|];
compile Div  [|(${left})/(${right})|];
```

4 Constraint Handling

In section 2 we highlighted three existing methods for handling constraints in evolution-based systems; penalty functions, repair functions and decoder functions. For our

---

[5]An open-source, Java-based string template language (http://velocity.apache.org/)

system, we have tried a new approach to constraint handling, in which information concerning the constraints is used to guide the initial process of solution generation, then constraint-aware variation operators are used to produce only valid children.

To guide the generation and variation operators, we use an approach that attempts to determine whether the tree being modified violates the constraints either directly or indirectly. A direct violation is where the nodes in the tree contradict at least one of the constraint conditions, whereas an indirect violation is where a partial tree[6] restricts the possible nodes that can be placed to only those that will contradict at least one of the constraint conditions. To make this problem tractable, the constraint language was restricted to only a small number of operators, which allowed us to hard-code a number of routines that were able to reason about the constraints at the sacrifice of losing Turing-completeness of the language. The following subsections cover the algorithms used for guiding the generation and variation of trees in further detail.

4.1 Solution Generation

Solutions are generated using an iterative two-stage process of checking which possible valid node instantiations can be made and then choosing one based on knowledge of previous good solutions. To start with, the generator component takes a partial tree as input, so to generate a tree from scratch a root node must first be instantiated and passed to the generator. As the first step, the generator sets all unset arcs to point to placeholder nodes and then adds all possible combinations of placeholder nodes and their type-compatible node types to a list of possible choices that can be made. The generation process then proceeds as follows:

- The tree constraints are evaluated with respect to the nodes currently in the tree to produce the constraints that must be satisfied by the remaining nodes to be added.

- Each of the possible choices is checked against the constraints and is removed if they would directly or indirectly violate them.

- If there is a placeholder for which there are no remaining possible choices, the algorithm backtracks.

- Otherwise, one of the choices is picked at random, its node type is instantiated and its corresponding placeholder is replaced with the new node instance. All alternative choices for that placeholder are then removed from the list of possible choices. All children of the new node are set to placeholder nodes and all combinations of the new placeholders and their type-compatible node types are added to the list of possible choices to be made.

- If there are no placeholders in the tree, the algorithm terminates, otherwise it repeats from step one.

---

[6] A partial solution tree is a tree where some branches end in non-terminal nodes rather than terminal nodes, and so some arcs have yet to be assigned child nodes.

4.2 Solution Improvement

Currently, we use minor variations on traditional GP techniques of crossover and mutation to produce new solutions from previous ones, with the initial population generated using the above algorithm of constrained random generation.

For crossover between two trees, T_1 and T_2, we randomly select a node N_1 from the first tree which determines its crossover point, but then we filter the nodes in the second tree by those that would form a type-compatible tree when interchanged with N_1. A node N_2 is then randomly chosen from this filtered list and the subtree rooted at N_1 in T_1 is swapped with the subtree rooted at N_2 in T_2. The tree constraints are then checked and, if violated, the swap is undone and N_2 is removed from the list of filtered nodes and another node is chosen. If no valid replacement node for N_1 can be found, a new crossover point is chosen in T_1. Mutation of a single tree is handled by randomly selecting a node N_1, removing the subtree rooted at N_1 and then passing the tree to the generator to fill in the gap.

4.3 Violation Detection

In this approach, the ability to reason about the constraints in order to predict which choices would directly or indirectly violate them has a large influence on the performance of the system. Currently, we preprocess both the constraints and the representation in order to build up the following meta-information that can be queried by the system in order to detect whether a violation has occurred:

- The *Must Type Set* of a node type contains itself and all of its supertypes.

- The *Transitive Must Type Set* of a node type is defined as the set of node types that must appear in a subtree rooted at a node of the given type.

- The *May Type Set* of a node type contains itself, all of its supertypes and all of its non-abstract subtypes.

- The *Transitive May Type Set* of a node type is defined as the set of possible node types and supertypes that can appear in a subtree rooted at a node of the given type.

- The *Shortest Terminal Length* of a node type is the minimum number of arcs that must be traversed from nodes of that type before a terminal node is reached.

A fixed set of rules are then used to determine whether a violation has occurred. These rules contain a pattern part that is matched against parts of the constraints and a condition part that, based on the current tree and the results of queries over the metadata, returns whether or not the constraint can be satisfied. For example, one rule looks for constraints of the form `exists NodeType in Subtree`, where `NodeType` and `Subtree` are variables, and will then check whether `NodeType` is within the May Type Set of any placeholder nodes within `Subtree`. If it is not, then this rule has successfully determined that the constraint

can never be satisfied by any complete trees built upon the current partial tree.

4.4 Evaluation

So far, we have tested our constraint handling approach on a number of small examples from very simple constraints, such as asserting that a certain node type must appear in all solutions, to complex ones based on node type co-dependency. Although we do not have enough results to produce a full quantitative analysis of the approach, we have noticed good performance in the face of complex constraints where the space of valid solutions is sparse. Unfortunately, this is often hidden by poorer performance when faced with simple constraints (due to the overhead of the system) or combinations of constraints that are not covered by our reasoning rules. This is partly because, when faced with a problem for which no rules exist, our system degenerates to an exhaustive search of the solution space, which can result in repeatedly taking paths that lead to dead ends.

However, one advantage that our system may have over penalty-based approaches (which we intend to check empirically) is that the destructive effect of mutation and crossover is reduced by guaranteeing that offspring will always be valid solutions. Such destructive effects (when children have lower fitness than their parents) can lead to introns and bloat in members of the population, which can hamper the evolutionary process (Soule and Foster, 1997).

5 Application to Algorithmic Art

To test our system, we prototyped three different problem specifications for generating different types of algorithmic art. The three types that we focused on were:

- Colour-based artwork, where the algorithm used to set the colour of each pixel in the picture is evolved, in a similar vein to NEvAr (Machado and Cardoso, 2002).

- Particle-based artwork, where the algorithm used to set the position and colour of 1000 particles is evolved and the particle trails are plotted over 100 time steps.

- Image-based artwork, where the algorithm used to set the colour of each pixel in an image can also use colour values from a source image.

A different problem specification was written for each of the above types, but each one outputs code in a language based on Processing[7], a scripting language used by graphic artists that is tailored to providing high-level drawing operations. These scripts were then executed to produce the resulting images.

5.1 Colour-Based Artwork

In this representation, the resulting programs loop over all pixels in the output image and set their hue, saturation and value based on some algorithm. The part of the

---

[7]See http://www.processing.org

program that loops over all the pixels is constant between solutions, however the algorithms used to set the hue, saturation and value of each pixel can vary. To allow for this, the representation has node types for constructing floating-point expressions which include constants (within the range 0.0 to 1.0 inclusive), simple mathematical functions (add, subtract, multiply, divide, sin, cosine and random) and variables (the x and y position, expressed as screen proportions).

Since the representation is quite restricted, the type system in the representation is enough to ensure that all produced solutions compile. However, there are still a number of compiling solutions that we want to rule out, such as those that cause errors at runtime or produce pictures that we know will be judged badly. To remove these from the search space, we added the following constraints to the constraints file:

- There must be a variable somewhere in the solution tree, where a variable is a reference to the x or y position in the image or a call to random. All solutions that do not contain variables will always produce images in which all pixels have the same colour.

- No constant representing the number one must ever appear as an operand of a multiply expression. Any solution that contains this combination could be simplified and so is redundant.

- All functions must contain at least one variable as one of their operators. This avoids constant subexpressions that may create values outside of the desired range or may be more simply expressed as a single constant.

The compiler file then specifies the mapping between the node types and the corresponding scripting code that draws the image. All of the numerical expression node types map to their expected operators, function calls or variable names, while the root type maps to the code that loops over the image and uses the generated expressions to set the hue, saturation and colour components of each pixel as shown below:

```
compile Main {
  |int width = 500;
  |int height = 500;
  |public void setup() {
  |   size(width, height);
  |   background(
  |     hsv(0.0f, 0.0f, 0.0f)
  |   );
  |   for (float y=0; y<1;
  |        y+=1/(float)height) {
  |     for (float x=0; x<1;
  |          x+=1/(float)width) {
  |       float h = ${hue};
  |       float s = ${saturation};
  |       float v = ${value};
  |       pixel(x, y, hsv(h, s, v));
  |     }
  |   }
  |}
};
```

Overall, the colour-based artwork problem specification consists of 130 lines of text spread across these three files which describes 29 node types, 5 constraints and 24 compiler rules. With this, we could generate, crossover and mutate solutions using the algorithms described in Section 4.1 and 4.2 and then compile and execute the resulting scripts and inspect the images generated. In figure 1, we present some example images generated using this representation.

5.2 Particle-Based Artwork

The second representation to be tested used a simple particle simulation as a basis for producing artwork. The generated part of the solution is the algorithm used to control the position and colour of 1000 particles over time. The static part of the solution creates the 1000 particles and then plots their trails over 100 time steps. In addition to this, a convolution is applied to the resulting image after each time step, the kernel of which can also vary. This has the result that lines drawn in early time steps will often appear more blurred than those drawn in later time steps, so that an impression of how the simulation has progressed over time can be seen in the resulting image.

The representation used here was very close to the colour-based representation, except with the variables now tracking the position, colour, previous position, time and index of each particle. Where the colour-based representation only evolved three numerical expressions, this representation evolves 12; six to initialise the position and colour of every particle and six more to update the position and colour of every particle in every time step, in addition to three constants that control the background colour of the image.

The constraints upon the representation are also more complex than those for the colour-based artwork, mainly due to the additional variables that are only in scope for particular parts of the program. For example, it makes no sense to reference the time step number or a particle's previous position in its initialisation expressions, so these are explicitly disallowed in the constraints.

Overall, the particle-based artwork problem specification consists of 238 lines of text which describes 44 node types, 6 constraints and 38 compiler rules. In figure 2, we present some example images generated using this representation.

5.3 Image-Based Artwork

The third representation to be tested used an existing image as input and could query this image for its hue, saturation and value components at any point, then use these values in numerical expressions for setting the colour of each pixel in the output image. This allowed it to produce image-filter style images by setting the output pixel colours to some function of the source pixel colours. It could also produce warps of the source image by assigning the output pixels to pixels at different positions in the source image based on some numeric function. Finally, it could also evolve the kernel of a convolution filter to be applied as a post-processing step. The result of this is that a range of images are produced, some of which obviously

contain the source image filtered in some way, and others which merely use it as a source of semi-random values.

The constraints for this representation were very similar to those of the colour-based representation, except that additional constraints were added to force all solutions to use the source image colours somewhere in its computation of the output image colours. This ensured that the search space of this representation did not include the search space of the colour-based representation as a subset.

Overall, the image-based artwork problem specification consists of 171 lines of text which describes 34 node types, 6 constraints and 28 compiler rules. In figure 3, we present some example images generated using this representation along with the source image used to produce them.

6 Future Work

In section 5, we used the domain of algorithmic art to test the usage of our framework for creative artefact generation, where we were able to use simple problem specifications to produce artworks of a similar nature to those produced with bespoke systems. However, many of the design decisions made during the development of our system have been motivated by our interest in scaling it up to handle much more complex problems efficiently. We are currently using the system in domains such as interactive art and the generation of simple computer games and have plans for 3D model and landscape generation.

Early results from these domains show that the evaluation of solutions is much more time-consuming than that of the algorithmic art shown here. For interactive domains in particular, the user must often try various input combinations in order to test for a response. We therefore believe that it is important for the system to extract more information from each evaluation. One way to achieve this is to allow the user to drill down into each solution in order to target the specific parts that are performing poorly. The system could then refine these parts separately until they meet the user's satisfaction, when they could be recombined with the rest of the solution.

We also intend to investigate general ways of allowing our system to refine solutions semi-autonomously in order to reduce the number of evaluations performed by the user. This could be done by allowing users to specify their preferences to the system as a fitness function over the phenotype, or a machine learning approach could be used to learn their preferences from the initial evaluations made during each session. By using a logic-based learning method such as Inductive Logic Programming (Muggleton, 1991), this opens up the possibility of the user understanding and altering the learned fitness function.

Finally, we acknowledge the that evaluation of systems and the artefacts they produce is an essential aspect of computational creativity which is missing from the work presented here and we aim to fill this gap. We are already planning a number of studies for assessing both the usability of our system and the appeal of the artefacts that it produces. We are also looking into ways to quantitatively evaluate parts of our system where possible.

7 Conclusions

We have presented the first description of our generic framework for automated program generation, and demonstrated its usage in an evolutionary art setting. We have found that bringing constraint checking into the solution generation process can help weed out systematically poor solutions and produce solutions faster than traditional generate-and-test approaches. As we saw with the application to three separate art generation problems, our framework enables rapid development of program generation systems. This has helped us to quickly prototype, test and refine various representations for different program generation problems.

Our current implementation is lacking in a number of areas that prevent us from applying it to solve more complex problems. Although the use of constraints helps to rule out many bad solutions, we will need to supplement this with more sophisticated constraints and a flexible fitness calculation mechanism. This is because, for more complex representations, we've found our existing constraint language is insufficient for ruling out enough bad solutions to enable convergence on good solutions within a reasonable time. However, this work is ongoing, and we expect to find a number of ways to address these issues in order to allow us to test the system on a wide range of different domains.

Acknowledgements

References

Baeck, T., Fogel, D., and Michalewicz, Z. (1995). Penalty functions. *Handbook of Evolutionary Computation*.

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

Gottlieb, J. and Raidl, G. R. (2000). The effects of locality on the dynamics of decoder-based evolutionary search. In *Proc. of the Genetic and Evolutionary Computation Conference 2000*, pages 283–290.

Johanson, B. and Poli, R. (1998). GP-music: An interactive genetic programming system for music generation with automated fitness raters. In *Genetic Programming 1998: Proc. of the 3rd Annual Conference*, pages 181–186.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*.

Machado, P. and Cardoso, A. (2002). All the truth about NEvAr. *Applied Intelligence*, 16:101–118.

Michalewicz, Z. and Nazhiyath, G. (1995). Genocop iii: A co-evolutionary algorithm for numerical optimization problems with nonlinear constraints. In *Proc. of the 2nd IEEE International Conference on Evolutionary Computation*, pages 647–651.

Montana, D. J. (1995). Strongly typed genetic programming. *Journal of Evolutionary Computation*, 3:199–230.

Muggleton, S. (1991). Inductive Logic Programming. *New Generation Computing*, 8(4):295–318.

Soule, T. and Foster, J. A. (1997). Code size and depth flows in genetic programming. In *Proc. of the 2nd Annual Conference on Genetic Programming*, pages 313–320.

Weise, T. and Geihs, K. (2006). DGPF - an adaptable framework for distributed multi-objective search algorithms applied to the genetic programming of sensor networks. In *Proc. of the 2nd International Conference on Bioinspired Optimization Methods and their Application, BIOMA 2006*, pages 157–166.
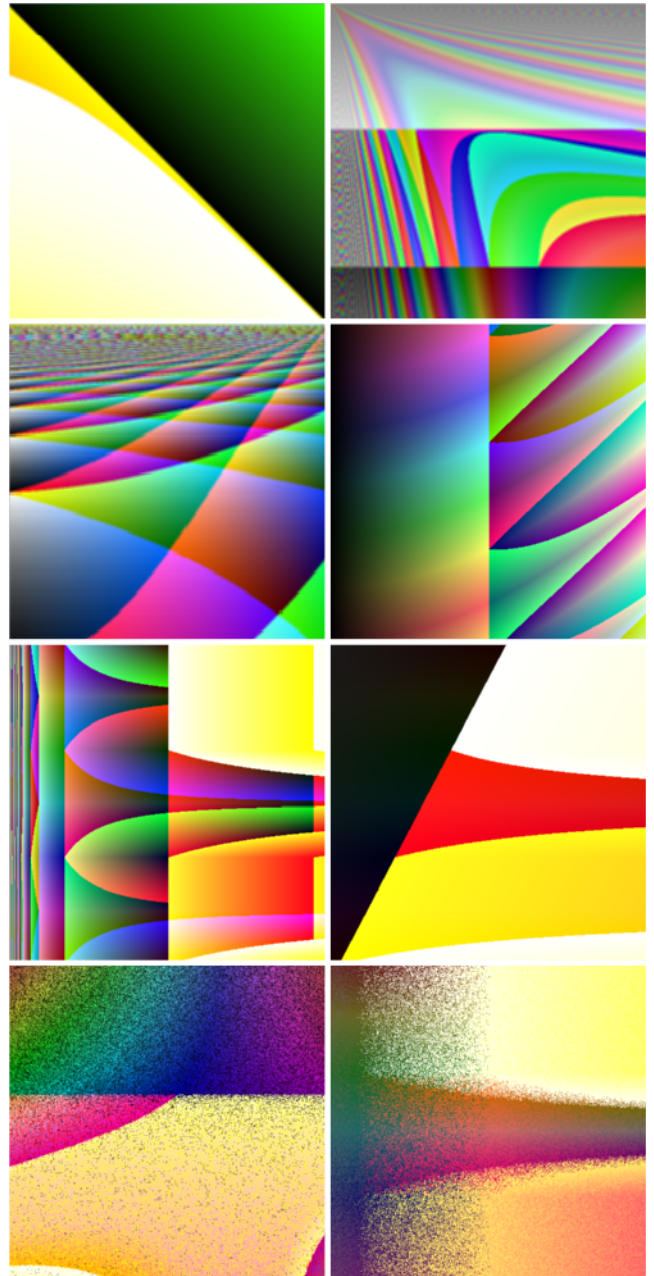

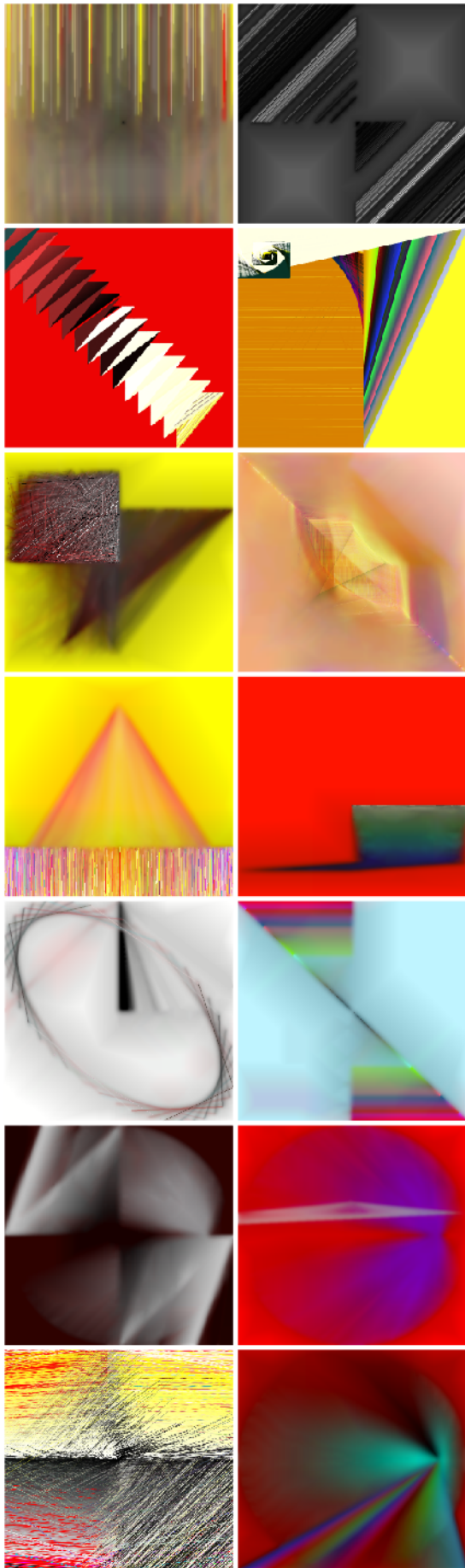Figure 1: Evolved images - colour-based approach
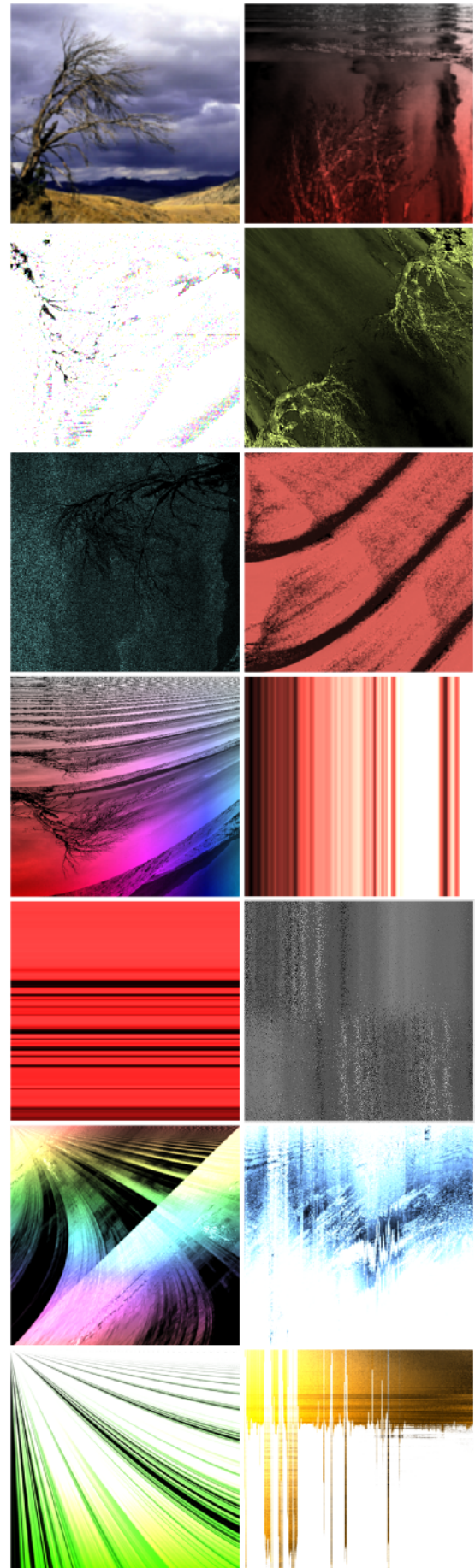
Figure 2: Evolved images - particle-based approach

Figure 3: Original and evolved images - image-based approach