Strongly Typed Genetic Programming To Promote Hierarchy Through Explicit Syntactic Constraints

Christopher Harris

Department of Computer Science
University College, London
Gower Street
London
WC1E 6BT
United Kingdom
Email: C.Harris@cs.ucl.ac.uk

ABSTRACT

Genetic programming uses a trivially hierarchical representation for its problem solutions but has no mechanism for hierarchically organising concepts and abstractions. Use of strong typing to constrain tree construction can help improve the performance of GP by enforcing a hierarchy of abstraction in parse trees. Several examples of explicit structuring methods using STGP are presented in the context of a template matching problem. Explicit structuring can help improve the performance of GP in this application area.

1. Hierarchical representations in GP

How GP comes up with the solutions it produces to problems is an issue at the heart of its understanding. The powerful representational aspects of GP allow us to produce an infinite variety of programs within a relatively constrained structure, but we remain ignorant of how these programs are produced except in specific instances of a single program through genealogy.

This paper looks at how GP constructs solutions, and how we can use a number of methods to influence this process. The most important factor is choice of primitive set, but this itself encompasses two issues. Firstly, the designer must decide what effect each primitive is to have in terms of transforming its inputs. Secondly, the designer must have some idea of how these primitives are going to work together.

The Building Block Hypothesis states that solutions are formed from the combination of short, relatively fit schema tathat lead to increased fitness when combined. In effect, a building block helps solve part of the problem and, when combined with other blocks, more of the problem is solved than either block could do alone. This hypothesis is reminiscent of the classic problem solving technique of divide-and-conquer, where sub-problems are identified and solved before being combined at progressively higher levels to solve larger sub-problems, eventually leading to a solution to the whole problem. This hierarchy of 'solving power' is a tempting explanation for how GP produces solutions.

The important issue of scalability could also be addressed if we knew how GP built up solutions hierarchically. The ability to produce solutions to large instances of the problem could come directly from solutions produced for small instances.

[O'Reilly 1995, Chapter 3] considers hierarchy in GP. Differentiations are made between various forms of hierarchy. A *hierarchical process* identifies and promotes useful subtasks, combining them into higher-level components, and producing solutions that are hierarchical in nature. *Hierarchical control* is a program structure that divides the tasks at the top level into subtasks, and the execution of a large task by the efficient sequencing of execution of subtasks. This differs from hierarchical process in that the former is constructive, producing new solutions from a 'pool' of subtask solutions, whereas the latter is the coordination of these partial solutions into a sequence. *Hierarchical structure* is a static characteristic of the representation of the solutions, in GP structure is trivially hierarchical because solutions are tree structured. Deeper levels of hierarchical structure can be seen in computer programs that use abstractions in both data and procedure.

O'Reilly defines a hierarchical solution as one that exhibits both hierarchical control and hierarchical structure. A hierarchical process would be able to construct solutions to a problem with a primitive set that doesn't explicitly encode subtasks as single nodes. If a primitive set does do this, i.e. if it is highly specific, then solutions can be found even without a hierarchical process simply by random combination of components. On a series of experiments evolving sorting programs with primitive sets of varying levels of specificity, it was concluded that although GP can produce hierarchically structured solutions, it does not do so with a hierarchical process. Although GP may stumble across a way to solve the problem that exhibits both hierarchical control and hierarchical structure, it does so through accident and not through a process of subtask identification and promotion.

Even if GP does not operate via a hierarchical process, using hierarchy can still be a good method of increasing the ability of GP to solve a problem. The use of hierarchy cannot be divorced from the notion of abstraction, indeed the very notion of a hierarchy implies the existence of a number of different levels of abstraction organised in a particular fashion, so we can use abstractions within GP to make use of hierarchy.

The abstraction of procedure within program trees has received a lot of attention due to the promise it holds for scalability. The idea of producing reusable subroutines for incorporation multiple times within a program tree is that it is only required that the subroutine have a useful effect once, and this effect is then gained for free throughout the rest of the tree. Although the mechanics of the various methods (ADFs [Koza 1994], Module Acquisition [Angeline 1993], Adaptive Representation [Rosca and Ballard 1996]) differ, their aim is the same, to produce solutions with hierarchical character by abstracting out procedures within the solution.

The other kind of abstraction within programs, of data, has not been explicitly dealt with in the GP literature. The technique of STGP is capable of producing data types of various levels of abstraction but work on STGP so far has concentrated on its ability to improve search by cutting down the search space. This chapter looks at ways of using STGP within the context of a single problem domain, to deliberately structure the search space such that solutions will have a hierarchy of data abstractions within them. It must be noted that the use of 'data abstraction' does not mean we will use abstract data types, nor will we include the ideas of inheritance hierarchies [Haynes, Schoenefeld & Wainright 1996], although these also may have a place in producing hierarchical solutions. We do however use complex data types such as container classes i.e. lists. We coin the term 'hierarchy of representation' (as distinct from 'hierarchical representation' which is easily confused with hierarchical structure) to mean the explicit structuring of program trees such that less complex data structures are to be found at the lower levels of the tree, and these are combined at higher levels through use of more complex data types. This explicit structuring, produced only through careful design of the primitive set and the type system for the problem domain, is an approach to hierarchy that has not been used in GP before. This paper illustrates how we can use this kind of structuring to improve the performance of GP through both the structuring of the low level data and of the operations performed on it at the higher level.

As a test bed we will use a template matching problem. This is a good choice of problem because we know every instance is potentially solvable (even trial and error would only take a few thousand attempts to find any picture with the setup detailed in Section 4), and we will show that GP is easily capable of solving low-order instances of the problem. We also show that performance on higher-order instances is poor, so we have a good way of judging improvements in performance.

2. Matching templates with GP

Before we can decide on an algorithm by which we are going to match features in an image, we need to look at the way the data is encoded in the image. The image representation will limit the ways in which we can apply GP to the detection of features.

We choose to use two representations of images for the experiments in the next few chapters. The first is a basic, low-level raster representation using a single number per pixel to represent grey scale values. The second is a line-based description consisting of a collection of line segments on a unit canvas. To simplify things further, we restrict ourselves to black and white images which makes similarity metrics easier (since we only have a binary decision at each stage rather than some graded response),

For a raster image, matching a template image with a target image involves getting some measure of similarity for the template and each region of the target. To save computation time we assume the images are of the same size so we can simply do a direct comparison. Matching is achieved by placing the template over the target and assessing some function based on the values of each pixel in the template and corresponding pixels on the target. This gives a similarity measure which can be used to drive the fitness function.

A simple matching method using raster images is correlation. Simply, the match between a template and a region of a target image is a function of the number of (assuming binary images) pixels in the template which have equal value to the corresponding pixels 'underneath' in the target region. Each pixel that matches increases the strength of the correlation, each mismatched pixel can either be ignored or can penalise the correlation, depending on requirements.

Such a method can be very successful but is very brittle in the face of small transformations of either image. In such cases correlation scores can drop off rapidly even with small deviations from the ideal, adversely affecting recognition performance.

To apply GP to the problem of template matching, we need to get GP to produce templates in some constrained random fashion which can then be assessed by a matching criterion. For completeness, we need to find a way of constructing templates that will allow us to produce an arbitrary binary image. This will give us the generality required to match any image we care to use as training data.

Since we need to match our templates with a bitmap image, we need some way of translating a symbolic description of an image into a bitmap. In computer graphics terms, we need to render our description. Our primitive set therefore must produce programs for drawing arbitrary graphics on a 2-D canvas. The most obvious way to do this is to provide a primitive set that draws geometric primitives in the image. Terminals in the set would be parameter values for functions to draw specific kinds of shapes, and the function set would consist of the shapes themselves plus transformations on the shapes produced lower down in the tree. This would allow the production of a genetic program that could produce just about any binary image, and allows

great flexibility on the part of the search. To encourage the production of images with some apparent structure, with no bias as to the shapes evolved, we use a primitive set based around straight line segments. A program using only straight line segments as drawing primitives has good generality and allows easy progression as more complex models are evolved. The initial primitive set used in experiments can be found in Table 1. The use of symbolic descriptions allows the manipulation of the shapes by transformation operators within the tree, then a single rendering operation on the line segments produced takes place after the tree has been evaluated.

The primitive set chosen is strongly typed. It is a property of image processing problems that a diverse set of data types is often used, and it is no surprise that different data types should be required when applying GP to solve such problems. In this case, it would be extremely difficult to find a primitive set that could produce a two-dimensional image from a program consisting entirely of functions that return single numbers. One way in which this might be envisaged is to encode x and y coordinates within the canvas as terminals and use GP to evolve an expression which returns a positive number for pixels that should be white, and 0 or negative number for black pixels. However this would be extremely unlikely to produce anything useful considering the possible complexity of the images. This problem illustrates the kind of situations where strong typing is not only desirable but absolutely necessary. This problem shows that GP can be used to evolve complex data structures, rather than programs, through primitives that build and manipulate those structures.

3. Adding structuring to solutions

A number of enhancements are presented in this section, aimed at improving the quality of the solutions produced by GP. These 'structuring methods' are directly gleaned from the problems identified with poor solutions from early test experiments. All enhancements come about solely by manipulation of the primitive set used.

It is worth pointing out here that, although from the point of view of the GP we are playing with a number of different types of data, the underlying implementation of the primitives remains virtually identical. In most cases no code changes are needed in the implementation even though we are changing the primitives used. This is because the types we introduce are only used for structuring purposes, and still map to the same underlying types in the implementation. All we are changing are the constraints on the process that generates and manipulates the trees. This is an important point - it shows how strong typing can be used as a general constraint mechanism of some power without a great deal of effort on the part of the implementer.

3.1 Changing the base level representation

We can use a more complex base-level element than a single line to build up images without necessarily biasing the search. Often, images are not unrelated collections of lines but exhibit some structure, perhaps being collections of shapes, or connected components. Both training images used in the experiments exhibit this property. The cross image is a single connected component of 6 lines. The face image is 6 connected components of between 2 and 4 lines.

If we use a notion of a connected component as our base level representation, it might be possible to match more quickly any structure inherent in the image since that structure would be a built-in property of our primitive set. As a low-level primitive, then, we can introduce a Component. This consists of a starting point and a series of 2-dimensional vectors marking the direction of each line from the end of the previous segment (or the start point, in the case of the first vector). For example, a Component with starting point (0.1,0.1) and vectors ((0.1,0.2), (-0.2, 0.3)) would produce two line segments with coordinates ((0.1,0.1), (0.2,0.3)) and ((0.2,0.3), (0.0,0.6)). Each component in the image could then feasibly be represented with a single, encapsulable, subtree in the program tree.

In the case of the cross image it is not possible to construct the whole image from a single component (there is no connected arrangement of lines that can 'draw' the cross in its box), but it can easily be represented with just two components. Even components containing only a couple of line segments could be useful as they can be replicated and transformed by the genetic operators, taking advantage of any regularity in the transform space of the image. The primitive set used for these experiments (called the 'component set') is detailed in Table 2 and replaces the NullLineList, line and Join primitives in Table 1.

3.2 Constraining the structure of program trees

The use of the NullComponent, NullLineList and NullVectorList primitives is designed to allow branches of the tree to terminate. The type systems require that each joining operation takes instances of the same type as they return as parameters, to allow an unspecified number of elements to be combined. By making this a binary process, i.e. a join primitive can connect two whole subtrees of base-level elements, we ensure maximum flexibility in terms of combinations and groupings of those elements.

The binary nature of these primitives does have a downside, however. At each point where a join primitive occurs, a Null primitive of the appropriate kind can be used as one of the subtrees. This leads to subtrees full of dead ends, littered with Null primitives which bloat to the programs and do not contribute to program fitness.

To reduce the incidence of Null primitives and thus reduce the chances of wasting genetic operations, we can change the way the join primitives work. Instead of allowing them to combine subtrees of elements in any way, we can use the type system to turn them into connectives, forming a backbone of such primitives, terminated by a single Null and chaining together base elements on the right-hand subtrees. This increases the chance of having more complex models produced by reducing the incidence of Null primitives in the program tree. The primitive set used here (called the 'chaining set') is detailed in Table 3and set is added to both the component set and the line based set, replacing appropriate primitives.

Primitive	Return type	Parameter types	Description
plus, minus, times, divide	double	double, double	arithmetic operators (protected division)
0, 1, 2, 0.33, 0.25, [0.1-0.9]	double	none	constants
Join	LineList	LineList, LineList	Joins one LineList to another
Rotate, XShear, YShear, Scale	LineList	LineList, double	Rotate a LineList, shear it in x or y direction, or scale about its centre.
Translate	LineList	LineList, double, double	Move a LineList around
line	LineList	double, double, double, double	Construct a line from endpoints
NullLineList	LineList	none	A terminator used for the Join primitive to allow trees to stop growing

Table 1. Primitive set for line-based representation

Primitive	Return type	Parameter types	Description
And	Component	Component, Component	Union of two Components
Join	Component	Point, VectorList	Construct a Component from start point and vector list
StartComponent	Point	double, double	Start point of a Component
Fragment	VectorList	double, double	Construct a 2-D vector
Add	VectorList	VectorList, VectorList	Join a number of VectorLists together
NullVectors	VectorList	none	A terminator for VectorList groups
Rotate, XShear, YShear, Scale	Component	Component, double	Rotate a Component, shear it in x or y direction, or scale about its centre.
Translate	Component	Component, double, double	Move a Component around
NullComponent	Component	none	A terminator used for Component unions

Table 2. Primitive set additions/replacements for component-based representation (replaces NullLineList and line primitives)

Primitive	Return type	Parameter types	Description
And	ComponentChain	ComponentChain, Component	(as for component set)
NullComponent	ComponentChain	none	terminate a ComponentChain
Add	VectorListChain	VectorListChain, VectorList	Join a a list of vectors
NullVectors	VectorListChain		Terminate a list of vectors

a. Primitive set additions/alterations for chaining (component base-level representation)

Primitive	Return type	Parameter types	Description
Join	LineList	LineList, Line	Join a list of lines (chain)
Rotate, XShear, YShear, Scale	Line	Line, double	Rotate a Line, shear it in x or y direction, or scale about its centre.
Translate	Line	Line, double, double	Move a Line around
line	Line	double, double, double, double	Construct a Line from endpoints

b. Primitive set additions/alterations for chaining (line base-level representation)

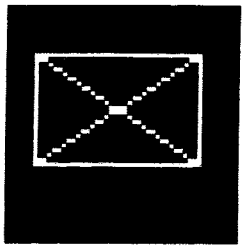
Table 3. Primitive sets for chaining

Primitive	Return type	Parameter types	Description
Transform	Component	Transformation	Convert a transformed component into a Component
Rotate, XShear, YShear, Scale	Transformation	Component, double	Rotate a Component, shear it in x or y direction, or scale about its centre.
Translate	Transformation	Component, double, double	Move a Component around

Table 4. Primitive set for transformation constraint

Primitive	Return type	Parameter types	Description
plus, minus, times, divide	double	ndouble, ndouble	arithmetic operators
0, 1, 2, 0.33, 0.25, [0.1-0.9]	ndouble	none	constants

Table 5. Primitive set for constant constraint (depth 2, replaces existing arithmetic operators)



a) 6-segment 'cross' picture

b) 17-segment 'face' picture

Figure 1. Training images used for template matching

3.3 Isolating transformations

With the chaining enhancement added, program trees hanging off the chain become dominated by transformations. This is due to the makeup of the subset of the primitives that returns a Component type, which has 5 transformations and the StartComponent primitive. This means that tree generation at points requiring a Component is dominated (since selection of primitives is random) by the transformation primitives. To remove this bias, we introduce a new function, Transform, as a conversion primitive. Each transform primitive is made to return a Transformation type, which is taken by the Transform function and simply returns a Component identical to that produced by the transform function underneath it. This is a type-based sleight of hand, no conversion is done on the data, only in the eyes of the type system. This reduces the size of the subset that returns a Component type from 6 to 2, giving the StartComponent primitive a much better chance of being selected.

The primitive set used for the experiments (called the 'transformation constraint' set) is detailed in Table 4.

3.4 Constraining constant generation

A lot of effort goes into calculating constant values to specify line endpoint coordinates. This can lead to the proliferation of subtrees exclusively to constant calculation, which can dominate the genome. This bloat of constant calculations absorbs many genetic operations and reduces the chance of larger scale structural changes occurring through an operation on a higher part of the tree.

Careful use of a type system of numbers can restrict the size of subtrees that calculate the constants. By specifying that functions that take double can only now take numbers of type ndouble, and supplying an appropriate set of ndouble terminals, it is possible to restrict the height of any subtree that needs to calculate a double to a maximum of 2. Note it is still possible to have single-node subtrees of type double, as these terminals remain in the primitive set. Again, this is a sleight of hand, since in the underlying implementation these values are all represented as doubles.

With subtrees restricted to such small depths, it is questionable whether GP has the power to calculate all the constant values we might want. It would depend on the number and values of the terminals available. It might be prudent to add ephemeral constants to the system to provide for some extra variability, but this is a separate issue from the use of strong typing. A further type, sdouble, can be introduced to allow the trees to grow to a maximum of three levels, providing better variability whilst still constraining the size of trees. At this point, the primitive set begins to get rather unwieldy, having large sets of terminals for each number type, although this has no adverse effects on the rest of the type system.

The primitives for this experiment (called the 'constant calculation constraint' set) are detailed in Table 5.

4. Experimental design

The experiments in this chapter use GP to evolve binary images that match a target image. In doing so, the solutions evolved are actually symbolic representations that map to a particular raster image when rendered. This means that as well as searching for a good match for a particular target image we are also converting from raw data into a higher-level description.

Using the primitive set described earlier, lists of lines are returned by each program. These lines are rendered into a binary image 50 pixels square, which covers the unit square in the space of lines. Any lines outside this unit square are clipped to the square before rendering. The image produced is then compared to the training image of identical size to produce a partial fitness value. The target image is the same size as the template to economise on runtimes.

For these experiments we used two training images in separate instances of the problem. These images were designed to reflect different levels of difficulty for GP by presenting targets with different complexities. The first image is fairly simple and consists of a box with lines across the diagonals, giving a total of six line segments to match. The second image is an iconic representation of a face marking the major features, and has a total of seventeen line segments (see Figure 1).

A total of 20 runs were made for each experiment, each with population size of 1000. The termination criterion stops runs after 100 generations had been completed, irregardless of the fitness values obtained. The run parameters can be found in Table 6. A total of 2 million individuals were evaluated, in a search space of 2^{2500} (the total number of combinations in a 50 pixel square binary image).

Each of these enhancements to the original primitive set cannot be looked at in isolation. In order to gain a good idea of the effect of using each method we must evaluate it in several contexts. For this reason we run an array of experiments using combinations of the four structuring methods. Most of these are only run with the component representation, due to time constraints. A total of 10

Objective:	To produce an image matching the training image with a set of line segments.
Terminal Set:	See Tables 1-5
Function Set:	See Tables 1-5
Fitness Cases:	2 experiments, each consisting of one 50x50 pixel image
Raw Fitness:	(no. of correct 'on' pixels - no. of incorrect 'on' pixels) - parsimony constraint
Standard Fitness:	(Max raw fitness) - raw fitness
Population size:	1000
No. generations:	100
Mutation rate:	20%
Crossover rate:	70%

Table 6. Tableau for template matching problem

experiments were performed.

1. line-based representation ("line")
2. line-based with chaining ("line + chain")
3. component-based representation ("component")
4. component-based with chaining ("component + chain")
5. component-based with transformation constraint ("component + trans")
6. component-based with constant constraint ("component + const")
7. component-based with chaining and transformation constraint ("component + trans + const")
8. component-based with chaining and constant constraint ("component + chain + const")
9. component-based with transformation constraint and constant constraint ("component + trans + const")
10. component-based with chaining and transformation constraint and constant constraint ("component + trans + chain + const")

For each set of experiments, 20 runs were performed using the face image as a target. Also, sets 1 and 3 were used with the cross image to show that GP can solve low-order instances of the problem. The cross image proved to be quite an easy task so a harder instance was needed. The face image is a much more challenging task and therefore appropriate to our needs.

4.1 Fitness function design

Perfect fitness scores are 184 and 132 for the cross and face pictures respectively. The fitness function used in training awards one point for each pixel correctly classified as 'on' and deducts one point for each pixel incorrectly classified as 'on'. Pixels classified as 'off' in the template do not contribute to the fitness. A perfect score therefore involves matching every pixel exactly, with no overspill into areas in the image where there are no 'on' pixels and is equal to the number of 'on' pixels in the images. Templates are thus encouraged to try out new lines, with the possibility of reward, but discouraged from simply filling up the image with positive pixels by the penalty term.

Once the fitness value for matching has been calculated, it is modified by a scaling parsimony factor. This factor, which is tunable by the user, penalises programs according to their length relative to that of the longest individual in the current population. It also scales the penalty according to the current best fitness value in the population to allow any penalty to scale with the progress of the run. This fitness value, after modification, is used to drive the genetic search but is not used when presenting the results. The unmodified fitness values of the best scoring individuals are used to show the results at the end of the run, demonstrating the ability of the best evolved individuals to solve the problem without reference to their size.

5. Results

The experiments detailed below are divided according to the structuring method used. Examination of the individuals evolved by the line-based primitive set highlighted various problems with the construction of the program trees which inspired the choice of structuring methods used in later experiments. To improve the performance of GP, particularly for the more difficult face image where performance is extremely bad, we need to examine the reasons for failure.

One possible reason for failure is a lack of complexity in the evolved trees. A genetic program cannot model an image containing 17 line segments if it cannot grow to contain at least 17 instances of the line primitive (ignoring colinear lines). The introduction of a parsimony constraint could also be a contributing factor in preventing the models from growing to sufficient complexity. A look at the best performing program trees shows that sufficient complexity rarely arises. For the cross picture, the average complexity of trees (in terms of line segments used) over all 20 runs was 3.3, but many runs exceeded that figure much earlier, going up to 12 line segments in some cases. Only 5 of the 20 runs had best-performing individuals with at least 6 segments by generation 100. For the face picture average complexity was 4.6 at generation 100, with peaks of up to 25 lines earlier on. Only a single run managed to produce an individual with more than 17 line segments at generation 100.

The primitive set as used in the line experiments also allows opportunities for improvement to be wasted by permitting useless genetic operations, and allows trees to be generated that have no use. The use of the NullLineList primitive, for example, means that any trees generated could

Representation	Chaining	Transformation constraint	Constant constraint	Average 'best of' 20 runs at generation 100	Maximum individual performance over 20 runs at generation 100	Fitness change when adding chaining		Fitness change when constraining transformations		Fitness change when constraining constants	
				ave	max	ave	max	ave	max	ave	max
Line				15.2	31	+16.1	+14.0	n/a	n/a	n/a	n/a
Component				9.9	26	+20.4	+24.0	+9.0	+8.0	-0.3	+7.0
Line	✓			31.2	45	n/a	n/a	n/a	n/a	n/a	n/a
Component	✓			30.3	50	n/a	n/a	-4.0	-9.0	-2.2	-11.0
Component		✓		18.9	34	+7.4	+7.0	n/a	n/a	+3.0	+11.0
Component			✓	9.6	33	+18.5	+6.0	+12.3	+12.0	n/a	n/a
Component	✓	✓		26.3	41	n/a	n/a	n/a	n/a	-4.8	+3.0
Component	✓		✓	28.1	39	n/a	n/a	-6.6	+5.0	n/a	n/a
Component		✓	✓	21.9	45	-0.3	-1.0	n/a	n/a	n/a	n/a
Component	✓	✓	✓	21.6	44	n/a	n/a	n/a	n/a	n/a	n/a

Table 7. Summarised results for face image experiments

be quickly terminated by the inclusion of this primitive, reducing the chances of producing a tree with sufficient lines to model the image. Additionally, the amount of effort devoted to calculating constant values for the line endpoints or for parameters to the transformation primitives is very large, and can dominate potential sites for genetic operations.

For any picture there exists a scope for building blocks to be found in the form of coordinate values. Since many line segments could have coordinate points in common, an evolved constant that codes for a correct endpoint could, once found, propagate by being copied to other line segments within the program. This is a single level of structural regularity that could be exploited. At a higher level, whole groups of lines can be replicated and/or transformed to exploit structural regularities in the image such as repeated components.

The results of the experiments involved in structuring the trees with the face image are shown in Table 7. These show the effects of using the various structuring techniques upon final fitness values at generation 100. Pictures produced by the best performing individuals for each set of experiments are shown in Figure 2.

5.1 Changing the base-level representation

Moving from a line-based description to a component-based one seems to have either no effect at all or a minor performance penalty. This runs against the intuitive notion that a more powerful representation would increase performance.

For the cross picture performance differences between the two representations are very small and can be disregarded. The line-based set achieves a best individual score of 162 from a possible 184. The average score over 20 runs at generation 100 was 81.15. This is less than half of the attainable maximum, and suggests that even though a very good match was found this is still a hard problem where the chances of success are slim. For the component-based set, the best individual scores 167, with an average score of 77.35. These scores are roughly comparable with those of the line based set so we conclude that changing the base representation has little effect.

For the face image, the best individual score for the line-based set was 31 from a possible 132, with an average at generation 100 was 15.15. For the component-based set maximum individual performance was 26, with an average of 9.9. These results are very disappointing and demonstrate the difficulty of the image as a target.

For the cross picture, the structural complexity of trees (in terms of line segments, totalled over all components) produced by the component set is very similar to that produced by the line-based representation. Average structural complexity produced by the component set at generation 100 was 2.6, with a maximum of 10. This is sufficient to produce the required model. However, only 2 of the 20 runs produced best individuals with the necessary complexity. This compares with average complexity of 3.3, with a peak of 9, for the line-based representation, where 5 runs produced sufficiently complex individuals,.

The 2 individuals produced by the component set that have sufficient complexity to produce the model have 10 and 9 line segments. These segments are contained within 8 and 9 component structures respectively. This indicates that the use of components to structure the layout of consecutive segments has little effect for the cross picture. Component structures then become simply a different way of generating single line segments in the majority of cases.

For a simple image such as the cross image, this is

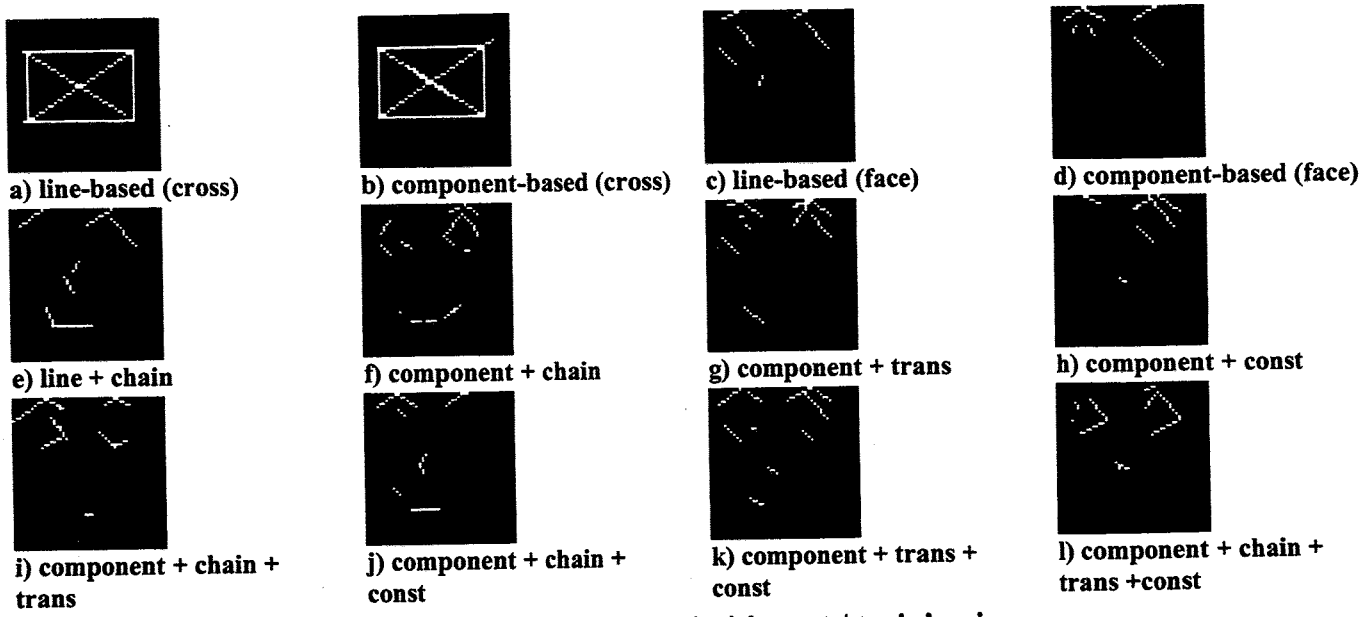


Figure 2. Best performing individuals for each primitive set / training image

understandable. It is equally easy for the GP to generate a new component and add a single line to it, as it is to add an extra line segment to an existing component. The extra structure added by the use of connected components is not needed if a representation based on isolated lines is sufficient to do the job. When considering the face image, the extra complexity needed would be expected to favour the component representation. In terms of complexity, the line-based representation never produces sufficiently complex models to capture significant portions of the training image. At generation 100, average complexity for the line-based representation is 4.6, with a maximum of 22. A minimum of 17 segments is needed to model the image, and only 1 run in 20 manages to produce a good performing individual with this complexity. For the component set, average complexity was 3.4 with a maximum of 15, so no runs produced a best-performing individual with the necessary complexity. The lack of complexity produced by the trees is addressed in section 5.2 where the gross structure of the trees is constrained.

The lack of progress associated with the move to a component base representation illustrates the problem with predicting performance when designing a complex primitive set for use with GP. Although such a representation seems more intuitive, the implementation of the necessary primitives may not be. A single line segment needs four coordinate points to evolve to values that produce a net positive fitness for that segment. A component needs two values initially plus a further two for each line segment, meaning that less effort is needed if we are looking at complex models. For single segments both representations need to evolve the same number of constants. In terms of efficiency of representation the component representation looks better because it can represent a complex model with fewer constants. Looking at the trees produced shows that this complexity is never used, however. Of the 20 best-of-run individuals at generation 100, only 4 had more than a single component in the tree, and only a single individual had a component containing more than one line segment. This means that the component structures were not being used to form complex shapes at all, but were simply forming single lines most of the time. The lack of complexity is a major factor in the poor fitness values obtained - the runs with the most complex individuals produced the best fitness values.

5.2 Chaining the construction of trees

Forcing program trees to be constructed with a 'backbone' of join primitives linking a series of data-producing subtrees leads to considerable increases in performance.

For the component representation, average best-of-run fitnesses over the 20 runs increase by a factor of three from 9.9 to 30.25, and the best individual overall performance doubles from 26 to 50. For the line-based representation, average best-of-run performance doubles from 15.5 to 31.2, and best individual overall performance increases from 31 to 45. These results indicate that both methods of producing templates benefit greatly from using chaining, the component representation gaining the most benefit and out-performing the line-based representation for the first time. When combined with the other methods used in these experiments, chaining increases performance in every case.

5.3 Isolating transformations

Separating transformations from data-producing subtrees via the type system seems to improve performance compared with keeping them of the same type. Compared to the ordinary component set we see a doubling of average best-of-run performance over the 20 runs, from 9.9 to 18.9. The best individual performance jumps from 26 to 34. This simple measure promotes the use of data-producing subtrees by increasing the chances of generating such a subtree during tree construction.

When combined with chaining, performance drops. Average best-of-run performance over 20 runs falls from

30.25 to 26.3. Maximum individual performance drops from 50 to 41. When combined with constant constraint (section 5.4) average best-of-run performance improves from 9.6 with constant constraint to 21.9 with both constant constraint and isolation of transforms. Best individual performance increases from 33 to 45.

5.4 Constraining constant generation

The use of strong typing to constrain the size of subtrees that calculate constant values has little effect on fitness when added to the plain component set. Average best-of-run fitness over 20 runs decreases from 9.9 to 9.6, with maximum individual performance increasing from 26 to 33. When added to the chaining method, constant generation constraint causes performances to drop from 30.25 to 28.1 (average over 20 runs) and from 50 to 39 (peak individual). When combined with isolation of transforms, the constraints increase performance slightly, from 18.9 to 21.9 (average, 20 runs) and 34 to 45 (peak). When added to both the chaining and transformation methods, constant generation constraint reduces average performance from 26.3 to 21.55 but increases peak performance from 41 to 44.

The use of constant constraint appears therefore to be a mixed blessing. When chaining is involved constraining the way constants are generated seems to have a negative effect. However, combining with transformation isolation improves performance. Combining with both has a mixed effect on performance. One explanation for the negative effect on performance when combined with chaining is that the increased complexity of the models resulting from the chaining means that a lot more constants need to be evolved and this could be made more difficult with the additional constraints, leading to less accurate solutions. When added to the transformation constraints, the small trees involved in constant calculation combined with lower complexity in terms of numbers of line segments, would divert more genetic operators to nodes involved in transformations, allowing good solutions to be found by manipulating the transforms of badly-placed line segments.

It is obvious that with a maximum subtree depth of 2, and only a few constants to choose from, it would be very difficult for GP to generate all possible values required to model the target image directly with un-transformed line segments. For a fifty pixel square image, fifty different values in the range 0.0 to 1.0 would be needed at regular intervals. The use of transformations would help to map line segments to positions which could produce positive fitness values. To see whether the lack of subtree depth was having a serious effect, 20 runs were carried out with an extra level of typing for the constant generation trees. This allows subtrees of depth 3 to be produced, whilst keeping all other factors constant, which allow a much wider range of calculations to occur within a single subtree. Results from these experiments seem to indicate a slight improvement, average performance moving from 9.9 (vanilla component set) to 10.25 and peak performance increasing from 26 to 34. Such figures however are only a marginal improvement over those gained by using depth-2 constraints. Further experimentation to see the effect of combining depth-3 constant constraint with other constraint methods would be necessary to see if this extra flexibility was beneficial across the board.

Using vanilla strong typing is a simple way to implement depth constraint on subtrees. Depending on the size of the primitive set being constrained, however, it can be quite inelegant. For depth 2, an additional 13 terminals are required to implement the ndouble type. For depth 3, on top of the ndouble primitives, another 13 terminals and 4 functions are required. This makes for large primitive sets which although easily coped with by the GP system cloud understanding for the designer. Some more explicit regime, perhaps extending the complexity of the tree representation to include depth constraints of subtrees of particular nodes, in the core GP system, might be more appropriate as a practical method.

6. Conclusions

These results show that the performance of a set of GP runs can vary widely if methods are used to constrain the architecture of the trees generated. The use of strong typing, a technique already available to the GP designer, is a flexible and powerful way of implementing many types of syntactic constraint of program trees. The experiments performed here show that with a small amount of effort, and a minimal amount of re-working of the problem definition and implementation, we can vastly increase program fitness over the levels achieved by the initial representation. More importantly, we have not changed the way that GP solves the problem but only the way in constructs the solutions - the underlying implementation and solution space remains the same.

The use of syntactic constraints via strong typing not only allows the search space to be reduced, but allows the paths taken within it to be 'funnelled' in particular directions. This has been shown to improve performance via three different examples.

Bibliography

[Angeline 1993] P. Angeline. *Evolutionary Algorithms and Emergent Intelligence.* PhD Thesis, Ohio State University. 1993.

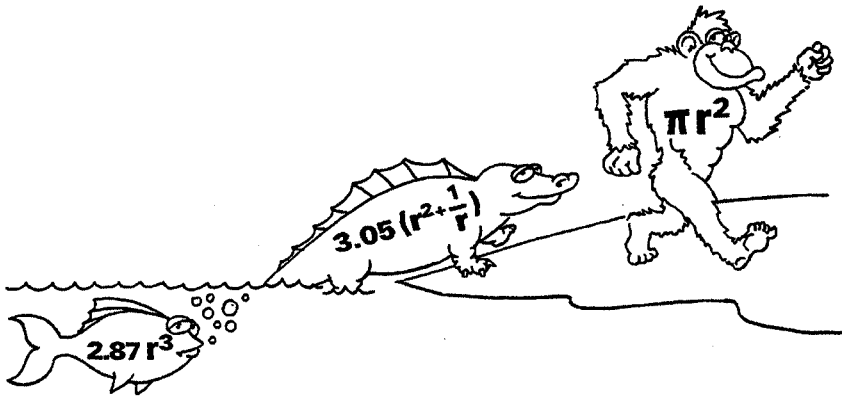
[Haynes, Schoenefeld & Wainright 1996] T. Haynes, D. Schoenefeld and R. Wainright. Type Inheritance in Strongly Typed Genetic Programming. In *Advances in Genetic Programming: Volume II*, K. Kinnear and P Angeline, editors. MIT Press. 1996.

[Koza 1994] J. Koza. *Genetic Programming II.* MIT Press. 1994.

[O'Reilly 1995] Unamay O'Reilly. *An Analysis of Genetic Programming.* PhD Thesis, Carleton University. 1995.

[Rosca 1996] J. Rosca and D. Ballard. Evolution-based discovery of hierarchical behaviours. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, AAAI/ MIT Press. 1996.

Late Breaking Papers at the Genetic Programming 1997 Conference Stanford University July 13 –16, 1997



Edited by
John R. Koza
Computer Science Department
Stanford University

Stanford Bookstore
Stanford University
Stanford, California 94305-3079 USA
415-329-1217 or 800-533-2670

ISBN

0-18-206995-8