THE PROGRAM STRUCTURE OF GENETIC PROGRAMMING TREES

MICHAEL A. FOSTER

Supervisor: VIC CIESIELSKI

Honours Thesis

School of Computer Science and Information Technology
RMIT University
Melbourne, AUSTRALIA

October, 2005

Abstract

Recent work has started to examine the structure of trees on a large scale to determine if they have an impact on the solutions found in genetic programming (GP) and also as measures of diversity and convergence. Much of the focus has been on the shape of evolved trees. Our work changes this focus onto the *symbol space*, an indication of what program functions and terminals exist, at what depth, and in what order. We create a new data structure, the *Symbol–Tree*, that stores the symbol space information and that can scale from single program trees to an entire population. Motivated by a need for measures such as the number of genetic nodes, number of nodes in the symbol space, and the similarity between programs, our data structure provides an efficient means of providing these measures. Our results using the measure of similarity between populations clearly showed change, convergence and stagnation within a run – important properties for a successful outcome.

Contents

1 Introduction

Evolutionary algorithms are typified by having a population of potential solutions spread over a search space. This is distinct from traditional optimisation methods where it is only a single solution adapted over a run. The biological evolution metaphor is further extended by having genetic operators such as crossover and mutation that create new results through either the sharing of genetic material (crossover) or through random change (mutation). A central idea to this theme is that of *survival of the fittest* – individuals that are better solutions are more likely to pass on their genes to their children. This biasing of selection for recombination helps the algorithm in a guided random search.

The potential solutions in evolutionary algorithm can take many forms. In this work we examine solutions that take the form of computer programs, otherwise known as Genetic Programming (GP). This method of problem solving was made famous by John Koza in *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza, 1992). One of the key concepts is that both the structure of the derived programs and their genetic material in the form of functions and terminals are guided by the fitness of the solutions. Thus a variable length structure is needed to store solutions. This most often takes the form of a tree structure although other representations are possible (Banzhaf et al., 2001).

The GP domain is however a fairly complex one. There are many reasons to perform analysis including the search for errors, real–time monitoring and guiding, examining phenomena such as code bloat (the excessive growth of programs without a clear improvement in fitness), or to investigate the relationship between population diversity and the ability to find good solutions (Daida et al., 2005). Recently (Daida, 2004) suggested a tiered view of how GP works consisting of three layers, *lattice*, *network*, and *content*. The level of lattice is concerned with the size of programs based on their representation. It predicts a region of size where trees of various depths are likely to fit based on building trees from smaller ones. The second level, network, is concerned with the diversity of programs that form a solution. It has been shown that only a small percentage of the population contribute to a final solution (McPhee and Hopper, 1999). The final level, content, relates to the actual program code that is being evolved and the structures it forms. This is the layer that our work will be attempting to analyse.

In particular we are motivated by the work of (Ekárt and Gustafson, 2004) and (Daida et al., 2005). In the former a data structure, the information hyper–tree or *iTree* for short, was developed that in its simplest form maintained a count of how many times a particular node position was sampled in a collection of GP programs. Using the iTree a number of measures relating to a population could be calculated including the number of node positions sampled, the total number of program nodes used to build the iTree (*genetic nodes*), the fullness of the tree, a number of measures of tree imbalance and a distance based on structure. Using a more complex representation of the data structure they were also able to calculate the entropy of a particular node position and the edit distance between two populations. We discuss the iTree further in section 2.2.1.

Whilst the visualisation of the iTree was discussed in (Ekárt and Gustafson, 2004) it was fully developed by (Daida et al., 2005). They describe a method for drawing binary trees that used a circular grid and is able to scale to many thousands of nodes. While not being explicitly mentioned, these visualisations could utilise a data structure very similar to the iTree. We discuss this method of visualisation further in section 2.3.2 as it is now an accepted form for visualising GP program trees.

There are, however, representations of a population that are not concerned so much with the structure (shape) of programs. We propose a program space that considers the functions and terminal (*symbols*) used in the population and the space of their children without taking account of the parameter order. We refer to this as the *symbol space*. This is a simplified representation compared to

maintaining every branch of every genetic program but that can still provide meaningful information. For example, how many nodes are in the symbol space? How many genetic nodes are in the population? Is there a correlation between the size of the symbol space and the number of generation nodes? How similar are two symbol trees? How similar is the initial population to the final population?

Our research questions are:

1. Is there a data structure that can efficiently maintain and calculate measures related to the symbol space?

2. How can we use this data structure and the associated metrics to create visualisations that can aid in the understanding of the GP search process?

3. Can these visualisation be used to explore the dynamics of a run?

The next section covers the relevant literature relating to our work. This includes an introduction to evolutionary computation and genetic programming, diversity and visualisation of technical data and in particular trees. Section 3.1 explains the Symbol–Tree data structure and works through the algorithm for creating it. This data structure is our encoding of the symbol space for a problem. In section 3.2, we formally define a number of metrics that can be calculated using the structure and in section 3.3 suggest a visualisation method that is better suited to viewing the Symbol–Tree. In section 4, we use the data structure and associated elements to examine a number of test problems. We end with a conclusion and some recommendations for further work.

2 Literature Review

There is a wide array of literature that we found relevant to this work. We start with a brief overview of evolutionary computation before examining the sub–field of genetic programming (GP) that is the focus of this work. The second section deals with diversity in evolutionary algorithms and GP and a number of metrics, measures and data structures that can be used to calculate them. The final section deal with visualisation of technical data. We note some of the principles of effective visualisation before giving a summary of visualisation in GP and focusing on the ways in which trees have been visualised. This final section is relevant since it seems intuitive that visualising the Symbol–Tree data structure could be beneficial.

2.1 Evolutionary Computation

Evolutionary computation (or algorithms) refers to the class of problem solving techniques that are inspired by evolution, biology and the natural world. These are typified by using a population of potential solutions that are improved in parallel and share information rather than improving on a single point as is typically done in traditional methods. It has become widely used due to its suitability for solving difficult optimisation problems, robust performance, flexibility and adaptability to a wide range of problems (Bäck et al., 1997). Part of its success is due to its conceptual simplicity starting with population initialisation and then repeating the steps of randomly varying the individuals, evaluating the *fitness*, and application of selection methods to choose the individuals that will contribute to the next population (Fogel, 1998). Among the first of these techniques to be developed was genetic algorithms (Holland, 1962; Holland, 1975). Genetic algorithms are characterised by generally having a fixed length chromosome that is able to encode a potential solution. This is usually done as a

binary string (Michalewicz and Michalewicz, 1997). Individuals are then changed through crossover between two selected parent strings and mutation.

Other techniques include evolutionary strategies (Beyer and Schwefel, 2002), evolutionary programming (Fogel and Fogel, 1995) and genetic programming GP (Koza, 1992). Our work involves this last field and it is discussed in the next section.

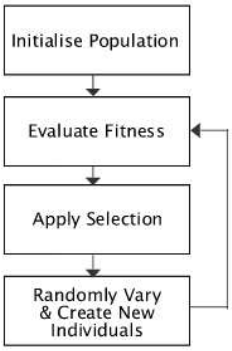
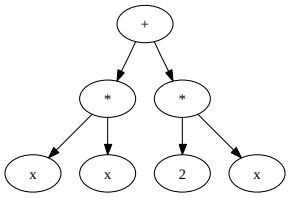


Figure 1: Evolutionary Computation Flowchart

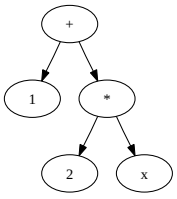
2.1.1 Genetic Programming

One of the key features of genetic programming (GP) is that it uses computer programs to solve problems. Using computer programs allows flexibility in taking on various sizes and shapes as well as using different functions to build a solution. To do this, some form of hierarchical structure is needed. This often takes the form of a linear list or tree structure (Banzhaf et al., 2001, p.5). In this work the programs will take the form of tree structures. From here on we will assume that this is the structure that has been chosen to represent solutions. We now follow the evolutionary computation flowchart from Figure 1 to describe how GP operates.

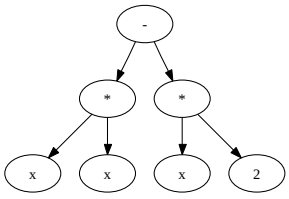
The first step is to create an initial population. In effect this does a blind random search of the space to create a random sample. Though there are many methods to achieve this, one of the most popular is the ramped half–and–half method by (Koza, 1992, p.93) which contains a mixture of full trees and randomly generated trees of a specific height. Other methods include ramped uniform (Langdon, 2000) which generates a uniform range of program sizes. An example of a population of programs can be seen in Figure 2.



(a) Program A : $x^2 + 2x$



(b) Program B : $1 + x^2$



(c) Program C : $x^2 - 2x$

x	f(x)
1	3
2	8
3	15
4	24

(d) Function Values

Figure 2: An Example GP Population

Given an initial population, a selection of genetic material is available for evolution. We now want

to determine how the fitness of a program can be measured. This is done using a ***fitness function***, a user defined function that when given a program, will determine how suitable it is for the environment. For example we may wish to determine a function, $f(x)$, that when given an x value will produce an $f(x)$ value as seen in Figure 2d. This function would describe the relationship between the data sets. This is also known as symbolic regression. One way of measuring the fitness of a program in describing this relationship is to evaluate the program at each x value and determine how different the returned values is to that of the provided points. Thus the total error will be the sum of difference at every test point. A program will be said to have solved the problem if it has an error of zero. We assume that lower values indicate better fitness as this is a common representation for optimisation problems. In the genetic programming literature it is also know as *standardized fitness* (Koza, 1992, p.96).

As a concrete example of fitness we evaluate the programs from our example population in Figure 2 a – c against the values in our table. It turns out that individual A is the actual solution to the problem with an error of zero. In spite of this, what is the fitness of the other programs? Program B has an error of 17 and program C has an error of 41. Over these values program B is clearly better than program C. As program A is the solution to the problem it is best solution.

The crux of Darwinian evolution is survival of the fittest. In GP, this is implemented at the selection stage. It works by providing the fitter individuals with a greater chance of passing on their genes to their children. We now have three individuals with varying fitnesses, how are they selected to participate selected as participants for evolution? We examine two methods: roulette selection and tournament selection. Roulette selection is achieved by splitting a notional roulette wheel into segments proportional to the fitness of the individual. The wheel can is then spun once for every individual needed but biasing the selection of the better programs. The second method, tournament selection, is also common. In this method a number of individuals are selected at random and then compete in a mini fitness tournament with the highest fitness program being the one selected. The size of the tournament in this method determines how likely the best individuals are to be chosen. This is also known as the selection pressure. In some cases it may not be possible to determine a numeric fitness. In this case it must be possible to at least rank solutions so that tournament selection can still proceed.

Selection by itself allows for the identification of the best individuals in a probabilistic way. In biological evolution, new genetic material is created through the breeding of two parents and the possible mutation of the DNA. This process needs to be simulated in the GP tree structures where the child elements should be similar to their parents so that the genetic material is passed on through lineages. There are a number of ways this can be achieved including reproduction, sub–tree crossover and sub–tree mutation. Reproduction is the simplest of the operators. It occurs by copying a single individual to the next generation. This simulates a strong individual surviving to the next breeding season. Sub–tree crossover occurs by selecting two parents and then choosing random points in their program trees to be swapped over. This creates two new children. The final common method is that of mutation. Here, an individual is chosen, a branch randomly chosen from the program, and then replaced with a randomly create one. These operators are often used to build a portion of the population. Once done, the population can again be evaluated for fitness. An example of new individuals being created using mutation and crossover can be seen in Figure 3. The crossover of information from program C to program B shows how better solutions can be created through the sharing of genetic material. Using the same fitness function as before, Program B would now be a match for our function.

The cycle ends when a program reaches some predetermined fitness, or reaches some other termination criteria such as having evaluated a certain number of generations. Other methods of ending a run check for a lack of diversity in the population as it can indicate that the run is trapped in a local

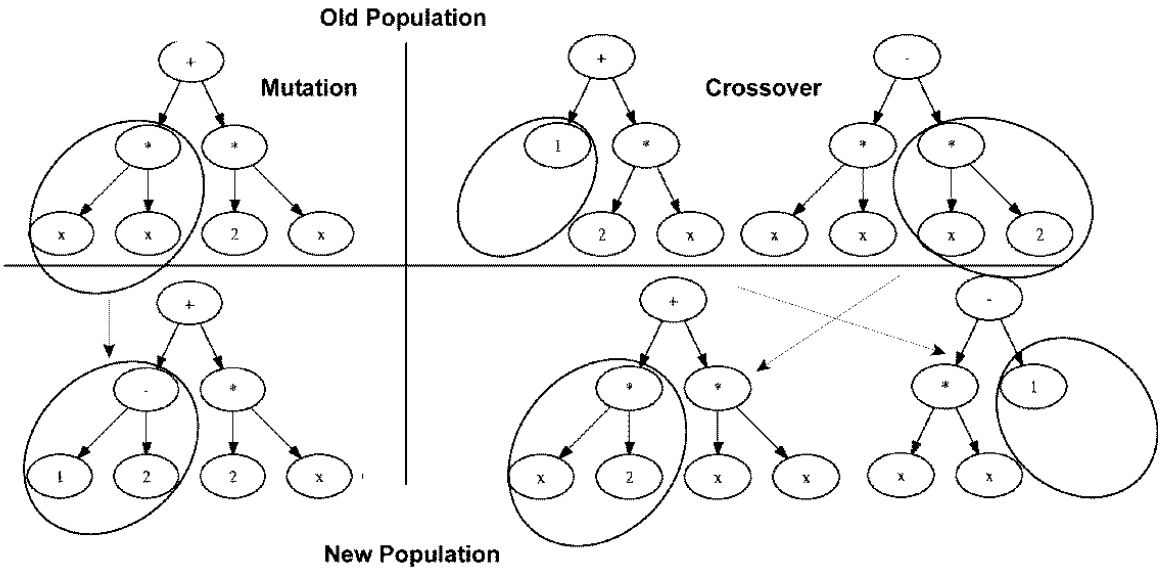


Figure 3: GP Operators Creating New Individuals in the Next Generation

optima that it is unlikely to escape from.

2.2 Diversity in Evolutionary Algorithms

(Wineberg and Oppacher, 2003a) give an overview of diversity measures in evolutionary computation and find that most of these are based on a comparison of all possible pairs. These have a comparison cost of $O(n^2)$ where n is the population size. The develop an implementation for genetic algorithms that can be achieved in $O(n)$ times. Whilst not following their reformulation of the problem to reduce the complexity, we note that it is important that comparisons can be done in less than $O(n^2)$ cost.

The value of a distance between populations is shown in (Wineberg and Oppacher, 2003b). The note that it is not the fitness of a potential solution that determines where the genetic algorithm will search next, it is actually the ratio of fitness to every other potential solution in the population. Because of this it is not enough to examine the diversity of single solutions, the diversity of the entire population must be considered. We note that our examination should compare the diversity between populations to take advantage of the population effects of the search.

2.2.1 Diversity in Genetic Programming

In this section we review the ways that large numbers of programs have been analysed using diversity in the genetic programming literature. We start with a number of summary pieces that allow us to classify the various techniques.

In both (Gustafson, 2004) and (Burke et al., 2002b) the diversity measures are categorised as either reflecting on the structures used, the fitness values, or a combination of the two. In the latter study it was found that a diversity of fitness values had a greater correlation with improved performance than a simple measure of genotype diversity. A further study (Burke et al., 2002a) expanded on the work and showed that a more complex metric for genotype diversity, edit distance, has a larger correlation with good performance.

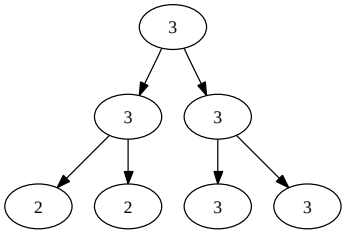


Figure 4: iTree For Example Population from Figure 2

Diversity Relating To Structure

One of the first measures of diversity was known as variety (Koza, 1992). This is the percentage of unique programs in a population. Variety was again used in (Langdon, 1996), however, Langdon noted that the measure ignores a number of issues including: that individuals may be similar but not identical, the extent of difference is not taken into account, differences between individuals may occur as introns (segments of code that have no effect on the program), different individuals may have similar or identical behaviour (especially if the functions are associative or commutative), and that the number of fitness values is often discrete so programs may end up with the same fitness despite their differences.

Edit distance is a common metric used to determine how many operations are required to change one tree into another. This was initially defined in (de Jong et al., 2001) which took account of the number of nodes that were dissimilar when two trees were overlaid. A second method was used in (Ekárt and Németh, 2002) which also took into account the depth of the change and scaled it accordingly. In the former study, they investigated using a multi–objective approach to minimise bloat (the excessive growth of programs) that also included a metric on diversity called Average Squared Distance. Ekárt and Németh extended this metric much more thoroughly, using it to analyse a fitness sharing approach and its impact on program diversity.

In line with the approach to examining the actual programs, recent work has begun to analyse the shape of evolved programs. In (Ekárt and Gustafson, 2004) a new data structure is developed called the *iTree* that records this information. Figure 4 shows the iTree for the example population from Figure 2. To create this, imagine that each of the program trees from the population are overlaid on top of each other. The value of each node in the iTree is then the number of trees that sample a particular node position. The iTree in this case shows that all nodes apart from the two leaf nodes of the left sub–tree were sampled by all programs.

Using this data structure a number of measures can be calculated. These include the number of node positions sampled in the tree search space (Equation 1), the total number of program nodes (genetic nodes) in a population (Equation 2), and the fullness of the iTree (Equation 3). All these require only the count of individuals that sample a particular node position.

$$M_1(P) = \sum_{A \in iTree} 1 \tag{1}$$

$$M_2(P) = \sum_{A \in iTree} n_a \tag{2}$$

$$M_3(P) = \frac{1}{size(P)} \sum_{A \in iTree} \frac{1}{2^{depth(A)}} n_a \tag{3}$$

Also, using the minimal information, the authors define two measures of imbalance. They suggest that an unbalanced iTree indicates a biased sampling of nodes. This imbalance may suggest that there is a high number of individuals with a common shape indicating low diversity. On the other hand a balanced iTree is more likely to occur when there is a large range of structures in the population. This indicates a high level of diversity. The two measures differ in that the first only takes into account a node existing whilst the other takes into consideration the number of times a node was sampled.

As a final measure using only the minimal iTree, they define an edit distance based on program structure. This is equivalent to an edit distance that ignores symbol information. The measure can be used as a distance between populations by using the cumulative iTree of each population for comparison.

If the distribution of symbols is also recorded, a number of extra measures can be calculated. The first of these was the entropy of a node, which measures the uniformity of distribution. Entropy gives a measure of node content diversity. As a final measure, they note that it is possible to calculate the diversity of average edit distances for a population.

Ekárt and Gustafson also note that it would be possible to visualise their data structure. This has effectively been done in (Daida et al., 2003a) and (Daida et al., 2005). We discuss their work further in section 2.3.2 but note that a visual measure of the structural diversity is possible. This type of measure may be more appropriate for online analysis of a run.

Also looking at the structure of programs, (Ciesielski and Li, 2004) developed a heuristic for commutative distinct individuals, that converted a string representation of a program by removing functions and replacing terminals with # characters. For example, the string $(\times(1+x)(x-(x-3)))$ would be converted to $((\#\#)(\#(\#\#)))$. This offers more granularity than Koza's *variety* since there are many trees that can share the same structure. However, it does not differentiate between trees that are similar but not quite the same. They discovered that there is a higher rate of duplicate evaluation than expected, that only a tiny fraction of the possible tree shapes were being examined, and that there was a large variance between the problems investigated.

Possible Improvements To Current Work

The fundamental flaw with the iTree data structure is the extensions needed for many of the metrics – the recording of the distribution of symbols within a node. Their case study does not use any of the measures that require this information for this very reason. It adds a great deal of complexity to a useful data structure without a significant increase in usefulness.

What is needed is a data structure that maintains information regarding the symbols whilst still retaining some of the structural significance.

It is also noted that visualising the data structure could reveal significant aspects of a population, run, or series of runs. This visualisation, again, appears limited to the basic data structure. Whilst Ekárt and Gustafson did not visualise the structure they noted some of the aspects that could be covered.

2.2.2 Requirements of a Metric

In section 2.2.1, a number of measures and metrics (or distances) were defined. It is worth noting that there is a difference between these two terms. A metric must satisfy the following conditions (Schweizer and Sklar, 1960):

$$d(p,q) = 0 \text{ if, and only if, } p = q.(Identity) \tag{4}$$

$$d(p,q) \geq 0.(Positivity) \tag{5}$$

$$d(p,q) = d(q,p).(Symmetry) \tag{6}$$

$$d(p,r) \leq d(p,q) + d(q,r).(TriangleInequality) \tag{7}$$

However, a measure does not have to meet any of these criteria. It is judged on its usefulness to the problem.

2.3 Visualisation of Technical Data

There are many methods for visualising information. The method chosen for a particular data set is often determined by the type of data. If the data is structured with the connections between data already known then these connection should be taken advantage of by visualising both the data and the links between them as edges. Much of our data is of this type and we further examine how it can be visualised in section 2.3.2. If the data is unstructured then the aim of the visualisation is often to find links between the data (Herman et al., 2000).

Evaluating the effectiveness of a visualisation is often not easy. Tufte (Tufte, 1983) is one of the most authoritative sources on the topic. He has set out a series of guidelines for the display of quantitative information. These include (1) show the data, (2) induce the viewer to think about the content rather than how it was created, (3) avoid distortion of the data, (4) present many numbers in a small space, (5) make large data sets coherent, (6) encourage the eye to make comparisons, (7) reveal the data at several levels of detail, (8) serve a reasonably clear purpose, and (9) be closely integrated with statistical and verbal descriptions. We will use these principles to evaluate later visualisation methods. Although these are mostly subjective criteria, there are some more practical ways to critique a visualisation including looking at the amount of ink used to display information compared to that of the graph itself.

2.3.1 Visualisation in Evolutionary Computation

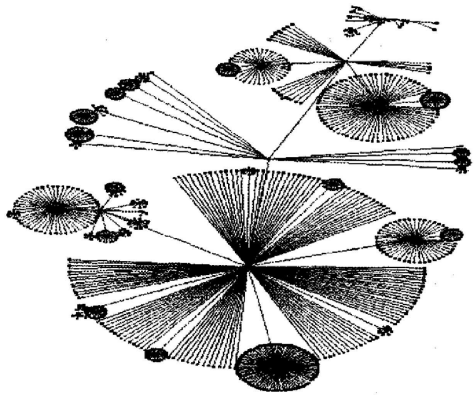
It has been noted in (Barlow et al., 2002) that understanding how evolution acts is a key problem and that there has been a significant number of attempts to visualise the behaviours. They classify the visualisations as relating to the information regarding the pedigree of solutions, information regarding convergence, or information about the fitness landscape. A set of standard techniques for visualising evolutionary computation has also been defined in (Pohlheim, 1999). These includes plots such as time versus best fitness and also mean fitness, fitness values of all individuals, and the distance between individuals in a population.

The visualisation of individuals in GP is more complex than in other representation in evolutionary computing. We next review the literature relating to the visualisation of trees since individuals in GP are program trees.

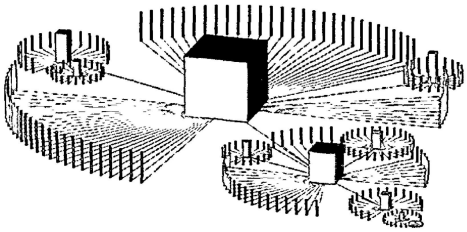
2.3.2 Visualisation of Trees

We begin the review of ways in which individual programs have been visualised with an introduction to the ways in which trees have been visualised. Trees are a subset of graphs and (Herman et al., 2000) have surveyed much of the relevant literature. They note that increased information is increasingly

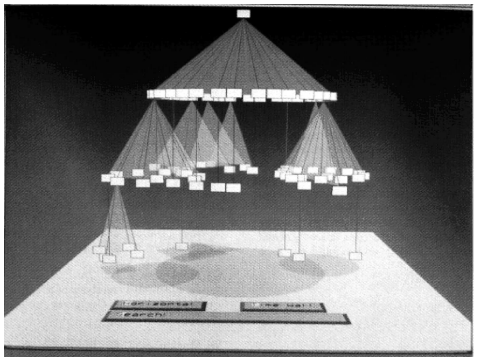
hard to plot due to the limitations of many drawing platforms. These include both screen real estate and the amount of time that it takes to draw a graph. They note that trees have generated the most interest and have a number of extra aesthetic rules applied to them. These include (1) nodes of the same depth should be drawn on the same horizontal line, (2) distance between siblings should be fixed, (3) nodes and edges should be evenly distributed, (4) edges should have the same length, (5) edges should be straight lines, and (6) edge crossing should be kept to a minimum or eliminated.



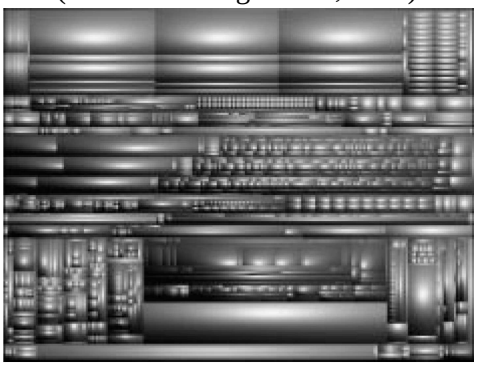
(a) Top View Cone Tree
(Carrière and Kazman, 1995)



(b) Cityscape Metaphor
(Keskin and Vogelmann, 1997)



(c) Robertson Cone Tree
(Robertson et al., 1991)



(d) Cushion Tree Map
(van Wijk and van de Wetering, 1999)

Figure 5: Various Methods of Visualising Trees

Trees can be represented in many ways. (Johnson and Shneiderman, 1991) have developed the Tree–Map. This operates like a Venn diagram but only the innermost elements are shown. The region is sub–divided given the number of children with each region being further sub–divided recursively. Whilst being quite effective in showing large amounts of information, it is hard to compare the various branches of the tree due to the moire effect [1] from the large amount of parallel lines. This way of visualising trees disregards many of the aesthetic rules that Herman et al. noted in their work.

The Tree–Map method has been extended in (van Wijk and van de Wetering, 1999) to eliminate some of the moire effect that makes it difficult to look at. This is done by varying the shading so that each rectangle gradually fades towards white as it nears the centre. An example of this can be seen in Figure 5d. Whilst this eliminates much of the moire effect, it still has the drawback that most of

[1] A shimmering pattern that makes a visualisation very hard to focus on.

the links are not visualised. It is much easier to see the structure of a tree than it is to see the actual structure of a Tree-Map.

Computing power has allowed for many visualisations to be done in 3D space. This 3D space was used by (Robertson et al., 1991) in their 3D cone trees as seen in Figure 5c. Their methods works by arranging the children in an outward extending cone below the parent. One problem with the cone tree is that an amount of user interaction that is required to fully explore that data set since elements can be hidden behind other elements due to this 3D layout. In respect to our work, this method is computationally too expensive to calculate and view.

The idea of a 3D layout has also been used by (Keskin and Vogelmann, 1997) using a cityscape metaphor (Figure 5b). This translates a circle layout of the tree into a 3D model. This better exploits the human perception system which is daily confronted with depth. The drawbacks of this method are that it is computationally more difficult and still requires interaction to investigate all the data.

Still in the 2-Dimensional realm, a hyperbolic geometry can be used rather than a Euclidean one (Lamping et al., 1995). Hyperbolic geometry is an idea that the information space is shaped like a globe. Thus as information reaches the edge of the globe the edges become smaller. It also has an impact on how parallel lines are drawn since they were no longer on a flat plane. This method can help extend the drawing of visualisation on a circular plane. It recognises that the space available for visualisation increases exponentially as we move out of the circle. One of the best known models for defining the hyperbolic plane in a Euclidean space is the Klein model. The main difference is that the length of a line will be a function of the position in relation to the perimeter of the disk (Herman et al., 2000). This indicates that while lines edges are of equal length as for requirement (4), they can be shown in the hyperbolic plane as lines of varying length.

Whilst Ekárt and Gustafson identified some cases of GP trees that could be interesting to visualise, it was Daida et al.(Daida et al., 2003a; Daida et al., 2005) who have set the standard for visualising the structure of GP trees. In particular we focus on (Daida et al., 2005) as it is the longer and more comprehensive work. In it their aim is to investigate GP through the visualisation of tree structures. They note that most visualisations of tree structure have been for example purposes only.

The main contribution from the work is a method of visualising using a circular grid. To do this each level is given a ring that the nodes can be plotted on and then this ring is divided up into a suitable number of points. The Daida et al. equation for calculating the ring radius of a node is:

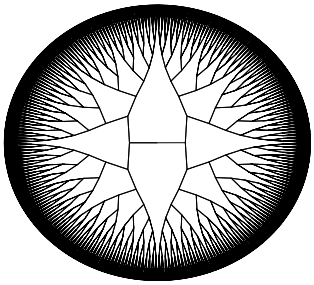
$$\rho(k) = \begin{cases} 0, & k = 1. \\ \lfloor \log_2 k \rfloor & k > 1. \end{cases} \tag{8}$$

and the node position on the radius by

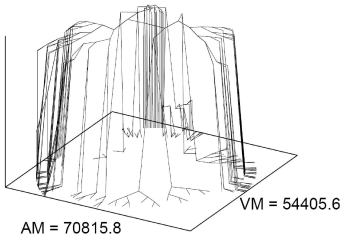
$$\phi(k) = \begin{cases} 0, & k = 1. \\ \pi\left(\frac{1}{2} + \frac{1}{2^{\rho k}} + \frac{k mod 2^{\rho(k)}}{2^{\rho(k)-1}}\right), & k > 1. \end{cases} \tag{9}$$

An example of this type of graph can be seen in Figure 6a. The method can be extended in a number of ways to investigate populations of programs. They give two main approaches. The first involves displaying each individual in a matrix type representation. The second method involves the use of some form of data structure that may have been similar to the iTree (Ekárt and Gustafson, 2004). Again, it can be imagined that every individual program tree is overlaid one on top of the other so that every branch used in a population can be seen. To also include information about the amount that each branch is sampled, the amount of shading has been varied so that the more common regions are darker in colour.

The method developed by Daida et al. has been extended by (Krasnogor and Gustafson, 2003). They take the circular lattice and transform it into a 3D representation. To further enhance the visualisation they allow the height of a branch to indicated the number of nodes that sampled that point. Whilst this method has not been used elsewhere it does allow an effective visualisation of the shape of programs in a population without the need to vary intensity of colours and shades. An example of this type of visualisation can be seen in Figure 6b.



(a) Example of Daida et al. method on a depth 10 tree.



(b) Root–labelled Lattice (Krasnogor and Gustafson, 2003)

(c) Daida et al. method on iTree from Figure 4.

Figure 6: Various Methods of Visualising Trees

The key point of the circular grid method of visualising trees is that similar shaped programs will be result in similar shaped visualisations. This is an important property when the visualisations are being used for online analysis and a general feel of the data is more important than a thorough examination. The limitation of this method, however, is that it is quite limited in the flexibility to deal with programs of a higher arity. Daida et al. note that it would be possible to redefine the method to deal with different arity, but as we will show later, there are cases where the theoretical arity may not be known before and may also be very large. In this case flexibility would be an important issue.

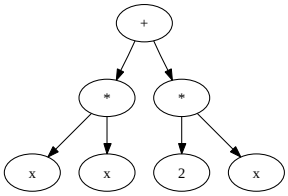
3 The Symbol–Tree

In the introduction we described the symbol space of a genetic program. This is concerned with the functions and terminals (symbols) used, their children, and the order in which they occur without taking into account the parameter order. Removing the parameter order specification simplifies the representation and creates a new space that can be examined. As an example and as an idea of what is being created throughout the description of the algorithm, Figure 7 shows the conversion from a program tree to that of a symbol tree. We can now see that in the Symbol–Tree there will be at least one node at every level of the Symbol–Tree with an identifier equal to that of every program element (symbol) at the equivalent level in the program tree.

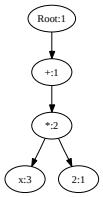
This section is divided into three parts. We start with the formal algorithm for creating the Symbol–Tree data structure. Following this we re–examine some of the metrics identified in the introduction as being important and show how they could be calculated using the Symbol–Tree. Lastly we review the ways in which the data structure and the metrics could be used to create visualisations.

3.1 Data Structure

Before formally introducing the algorithm for creating a Symbol–Tree it is worthwhile briefly examining the structure of nodes that make up the tree. As can be seen in Figure 7b, each node contains



(a) Program A From Example Population, Figure 2.



(b) Resultant Symbol–Tree of Program A.

Figure 7: Resultant Symbol–Tree From Program A

two pieces of data. The first of these is the *identifier*. The second element separated by the colon is the *count*. The only other information that need be maintained by a node is a list of its *children*. We will explain the usage later of these elements throughout the algorithm description.

There are three basic steps in creating a Symbol–Tree. They are:

1. Initialise Symbol–Tree
2. Add New Tree
3. Add Tree To Node (Recursive)

Given the complexity of the algorithm, it is necessary that we work through each of the three steps in detail.

1. Initialise Symbol–Tree

   A Symbol–Tree containing more than one individual's information has the potential for a number of different root nodes to be present in the population of programs. Unlike in the iTree where the root nodes of the tree will be matched for all individuals, the Symbol–Tree needs multiple nodes to record the root information of its constituent individuals. Thus an artificial root node is required to maintain the structure of the tree.

   The Initialise Symbol–Tree step consists of only:

   (a) Initialise Root Node

   This creates a root node of the same type as the rest of the nodes in the tree. It has an artificial symbol identifier that marks it as the root node of the tree. The count of this node corresponds to the number of trees that have been added to the structure.

   An example of this initialisation can be seen in Figure 8. In this worked example, we will show the adding of program A from the example population in Figure 2 to a Symbol–Tree.

   Once this step has been completed, the root node need not be re-initialised. Further program additions can begin directly at the next step.

2. Add New Program

   Once the Symbol–Tree has been initialised a program can be added. This step performs two functions:

(a) Set $current \leftarrow$ Symbol–Tree.*RootNode*

We initialise a new pointer that determines which node in the tree we are currently accessing. This will change as we move down the tree. Since the root node of the Symbol–Tree is a special case. This step is needed

(b) Increment *count* to indicate new tree sampled.

This increments the counter on the root node of the Symbol–Tree every time a new tree is added. This allows for the quick calculation of metrics requiring the number of trees held in the structure.

At this stage the root node of the Symbol–Tree has been initialised and can be located using the *current* pointer. The root node counter has been incremented to reflect that a program is being added to the structure. The current configuration can be seen in Figure 9. The next step recursively adds the program to the Symbol–Tree.

3. Add Tree (Recursive)

So far, neither step has had to deal with the actual program we are adding to the structure. We assume that the program is in some form of tree structure similar to our example population from Figure 2. One of the important properties that a tree structure has is that each child forms a sub–tree that is itself a tree. Using this property, the recursive addition of nodes is possible on sub–trees of the whole program.

(a) Set $current \leftarrow current.getChild(ProgramTree.RootNode)$

This step attempts to retrieve a child of the node pointed to by *current*, that has the same symbol identifier as that of the root node in the program tree that is being added. If it is found then it is assigned to the *current* pointer. If it does not exist, then a new node is created with a count of zero and a symbol identifier equal to that of the root node in the program tree. An example of this can be seen in Figure 10 part A.

It is at this step that the Symbol–Tree process is most different to that of the iTree. In the iTree data structure. the child would be selected on the parameter number of the program node. Ekárt and Gustafson encode extra information regarding the frequency of symbol types inside the nodes as a list although this adds extra complexity to their structure. Our algorithms could be changed so that the identifier was a combination of symbol and parameter number to also encode this information. It is possible that this information could significantly increase the size of the Symbol–Tree and is thus left out of this study.

(b) Increment *count* of $current$ to indicate node sampled.

$Count$ represents the number of times a symbol has been accessed. In this case it is the frequency of a symbol in some order of a program. This order ignores the parameter order but takes into account parameter existence. This is a simplified measure that allows the Symbol–Tree to be more compact. This step can be seen in Figure 10b.

(c) Add sub–trees of ProgramTree to $current$

This step activates the recursive quality of the algorithm. Each child node of the program tree root can be split into a sub–tree. These sub-trees are in themselves trees and can thus be used in the Add Tree step of the algorithm. One of the important properties is that the algorithm will work with program trees of any arity.

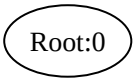


Figure 8: Initialising a Symbol–Tree. Consists of creating an artificial root node that maintains the tree structure. It also contains a count of the number of trees used to build this structure.

Figure 9: Adding a New Program Tree to the Symbol–Tree. This step increments the count in the root node and sets a pointer, *current*, to the root node so that the adding of a program can proceed.

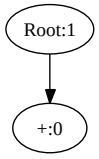
The Symbol–Tree now has two nodes as shown in Figure 10; the artificial root node that records the number of trees held and the root node of program A from our example population. The process of adding the remaining nodes from program A continues recursively at the Add Tree step. We will continue to trace through the addition of nodes as an example of how a Symbol–Tree is built from scratch.

Figure 11a–f indicate each step. The next symbol added is the multiplication operator that is the left child of the program root node. In the Symbol–Tree it gets added as a child of our addition node. For the duration of this walk–through we simplify the steps of the algorithm so that the finding of a child, increment of the child's count, and recursive adding of sub–trees is seen in a single diagram.

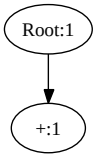
We have shown above how a Symbol–Tree is created from a collection of programs. The cost of creating the data structure is bound by the number of nodes in the collection or population. Our next question is what range of time complexity can we expect? Our collections so far have been mostly of full trees. These can be expected to be the worst performing with the overall time cost of $O(NumberOfTrees \times (MaxTreeArity^{Depth+1} - 1))$. The best case is where there is a population of sparse trees. The time complexity of visiting every node in the population will then be $O(NumberOfTrees \times (MinArity \times Depth + 1))$. Thus the actual complexity is likely to fall somewhere in between

$$O(NumberOfTrees \times (MinArity \times Depth + 1)) \geq$$
$$ActualComplexity \geq$$
$$O(NumberOfTrees \times (MaxTreeArity^{Depth+1} - 1))$$

The value of this actual complexity will often be problem dependent. For example in the Max

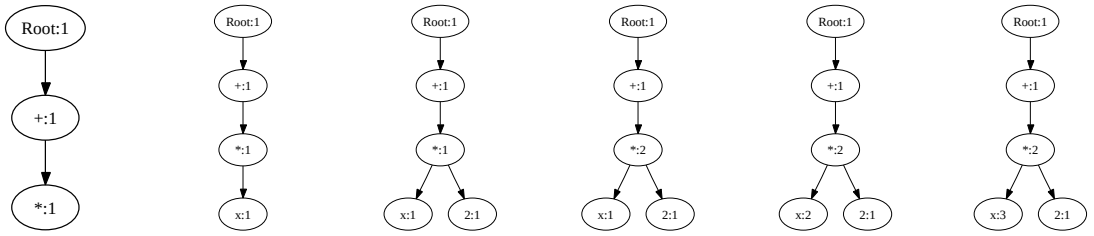


(a) Retrieving child. In this case it has to be created since that node position has not been accessed previously.



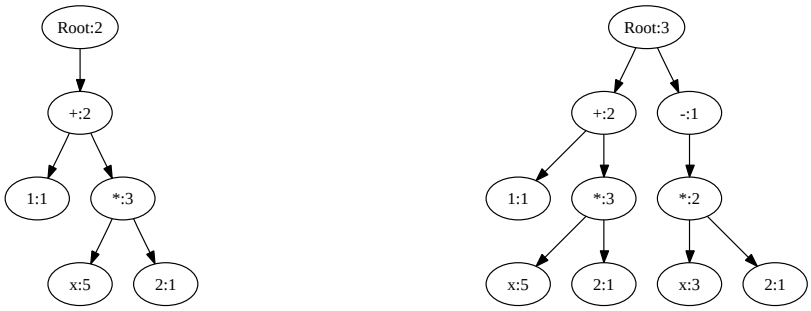
(b) Incrementing the new current node.

Figure 10: Building of a Symbol–Tree – 1



(a) Addition 2 (b) Addition 3 (c) Addition 4 (d) Addition 5 (e) Addition 6 (f) Addition 7

Figure 11: Building of a Symbol–Tree – 2



(a) Symbol–Tree after adding Program B. (b) Symbol–Tree after also adding Program C.

Figure 12: Building of a Symbol–Tree – 3

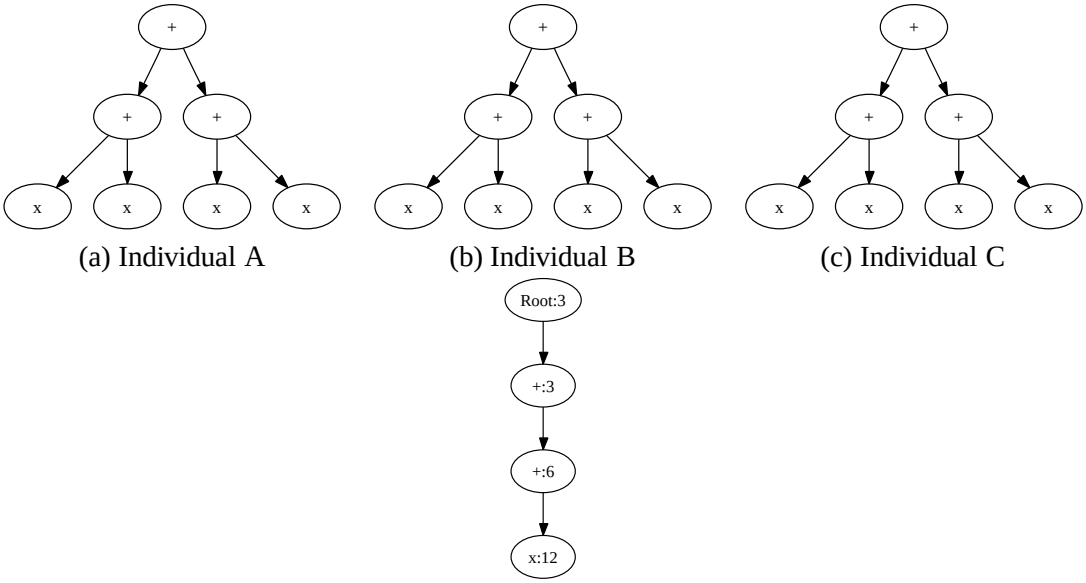
problem (see section 4.1 for further details) the aim is to build a full tree since it is the only tree shape that can solve the problem. This is not a common attribute for GP problems though. Recent work in (Daida and Hilss, 2003) and (Daida et al., 2003b) suggests that there is a preferred size for GP programs and that this falls somewhere in the middle of the best and worst cases as identified above.

Whilst the time complexity of creating a Symbol–Tree is interesting it does not reflect the variance in size the data structure may have. So far we have shown general cases that do not show all the properties of the data structure. We now examine the most and least compact representations of a population possible using the data structure?

We start with a case in which the Symbol–Tree will be highly efficient. Figure 13 shows a population with a high degree of structural similarity, in fact each individual is identical with a low distribution of symbols. The greater the correlation near the root of the tree the more efficient the resulting Symbol–Tree should be. The resulting Symbol–Tree from the population is shown in Figure 13d. It contains only four nodes compared to the twenty one required by the population – a significant space saving. This represents the best efficiency of the data structure, a space complexity of $O(depth + 1)$ where depth represents the depth of the programs in the population.

What is described above is the ideal case. It is also possible that the space of programs could create an inefficient Symbol–Tree. Figure 14 shows an example population where the Symbol–Tree will do poorly. This is surprising at first glance since every program has the same distribution and layout of nodes in every level apart from the root. What is more surprising is that the efficiency of the Symbol–Tree ends up *worse* than that of just storing each program individually needing twenty two nodes to twenty one. At its worst the Symbol–Tree could have a size complexity of $O(|F \cup T|^{depth+1})$, a large value given there are possibly many different functions and terminals possible in a program.

Given this result, should we expect a typical GP population to perform as poorly as this example population? The answer is probably not. As pointed out in (Gathercole and Ross, 1996) and (Langdon



(a) Individual A (b) Individual B (c) Individual C

(d) Resultant Symbol–Tree

Figure 13: An Efficient Symbol–Tree

and Poli, 1998) the root sections of a population converge quite quickly. Further, (McPhee and Hopper, 1999) found that in the majority of runs the final population often shared as much as the top five levels of nodes. Whilst probably not being as optimised as the population in Figure 13, we can expect a fair amount of similarity particularly in the latter generations of a run.

These examples do give rise to further uses of the Symbol–Tree. Both are made from the same number of genetic nodes yet there is a large difference in the number of nodes in our Symbol–Trees. We investigate this further in the next section.

3.2 Measures, Metrics and Their Complexity

In the introduction we suggested a number of measures that would be useful to calculate in regards to the symbol space and also GP in general. We now list these and formally define the measures, what they can tell us about a population, how they can be calculated using the Symbol–Tree and finally a comparison of how this would be completed without the data structure and the complexity of the operation.

The measures we wish to calculate are:

1. How many nodes are in the Symbol–Tree?
2. How many program nodes are in the population?
3. Is there a correlation between the size of the Symbol–Tree and the number of generation nodes?
4. How similar are two symbol trees?

For each of these measures we assume that the Symbol–Tree has already been created and that it will not contribute anything extra to the complexity. This is a simplified case but seems fair if we are going to calculate a number of measures.

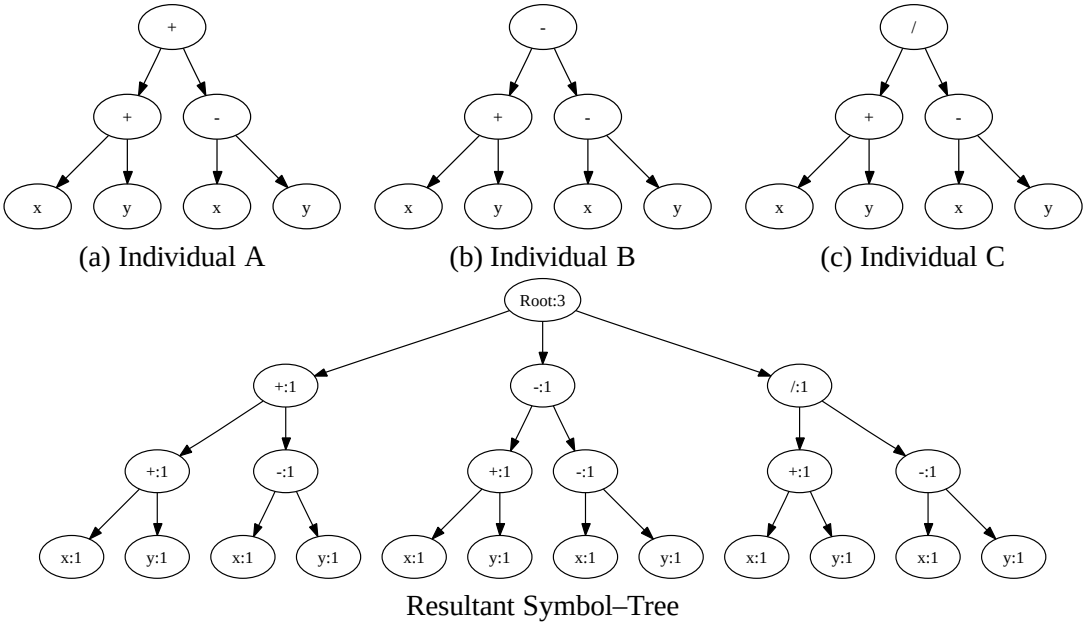


Figure 14: An Inefficient Symbol–Tree

Measure number one is quite easy to calculate. In fact no new material needs be developed since equation 1 covers it. When applied to the Symbol–Tree it calculates the number of nodes in the symbol space. We continue to refer to this measure as M_1.

To calculate the this measure it is necessary to traverse the entire tree. This has a worst cost of $O((|F|^{D-1} - 1) + ((|F|^D \times |T|) - |T|))$. The tree size is slightly minimised since we can determine that a number of the symbols will be terminals and will therefore only act as leaf nodes.

It would be very difficult to calculate this without the Symbol–Tree. It would be likely that a Symbol–Tree or similar data structure would need to be created every time it was to be calculated. We have examined the costs of this previously.

The second measure relating to the number of genetic nodes in a population can also be adapted from equation 2 of (Ekárt and Gustafson, 2004). It traces through each node in the population and calculates the sum of *count*. When not including the artificial root node this value is a measure of nodes used to build the tree. We continue to refer to this measure as M_2.

This measure is important because it can indicate if there is a steady program size, a number of bands, or random movement of tree sizes within the population. It can also be used as a measure of the computing power needed to evaluate the entire population.

The complexity of this operation is the same as that of the last measure. It requires that the Symbol–Tree is traversed. The two measures could be computed in parallel to increase the performance.

Without the symbol tree, this measure needs be calculated by traversing every program tree in the population.

The correlation between the symbol space and the number of genetic nodes is a measure that combines the last two measures. In this we are interested in the ratio of nodes in the Symbol–Tree to those in the entire population. To make these measure more comparable between various runs the measure can also be normalised by turning the M_2 measure into an average number of program nodes per program. We can retrieve the number of program trees in the population from the root node of the

Symbol–Tree using,

$$NT = RootNode \rightarrow count \tag{10}$$

The measure can then be calculated as:

$$F_1 = \frac{M_1 \times NT}{M_2} \tag{11}$$

Given that we can calculate both M_1 and M_2 in parallel the complexity of the measure is the same as was previously shown.

The final measure, and probably the most interesting, is that of similarity. This counts the number of genetic nodes that have similar sub–trees. A sub–tree can only be similar if the symbol identifiers of those nodes are the same as that of a compared sub–tree. Since each symbol identifier can be allocated only once per node (for example there can only be one child node of type "+" at any node in the Symbol–Tree). The equations to calculate this can be seen in equation 12.

$$S_1(P_1, P_2) = \begin{cases} \sum S_1(i,j), i \in P_1.Children, j \in P_2.Children, & Root \\ count + \sum S_1(i,j), i \in P_1.Children, j \in P_2.Children, & P_1.id = P_2.id \\ 0, & P_1.id \neq P_2.id \end{cases} \tag{12}$$

We can then convert this number to a percentage using

$$S_2(P_1, P_2) = \frac{S_1(P_1, P_2)}{\max(M_2(P_1)M_2(P_2))} \tag{13}$$

We use the maximum genetic nodes size of the two trees because otherwise two trees that overlap but are of different sizes would be equal. This is not the case.

This equation does not, however, meet the criteria for becoming a metric since it fails in identity (Equation 4) and also the triangle inequality (Equation 7). To address these problems we can turn it into a dissimilarity percentage by

$$D_1(P_1, P_2) = 100 - S_2(P_1, P_2) \tag{14}$$

Now trees that are the same will have a dissimilarity of zero. While being a metric is an important property, we find it conceptually simpler to use the similarity percentage, S_2, for the remainder of this work. Other uses of the metric are however possible that require the formal properties of a metric.

The worst cost of this operation occurs when the two trees are the same. In this case both trees must be fully traversed. In the best case it will be only the root node that is compared.

3.3 Visualisation

It was shown in the literature review that (Daida et al., 2004) have set the standard for visualising GP program trees. Their method works well for binary trees but is limited in that it does not scale well to larger arity trees. It was mentioned that the method could be modified for other arities but this still does not allow for optimisations based on nodes with a smaller arity. In one respect this is good since it gives a consistent visualisation. On the down side it can quickly run out of space. It also does not fit well in the GP domain where there are many functions with an arity greater than two or with only one parameter. Thus we have chosen a method that is much more flexible.

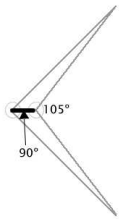


Figure 15: Effects of Hyperbolic Plane

Klein's model uses a hyperbolic plane to increase the amount of space that can be used to visualise without a great increase in complexity. This lends itself well to cases where we want to visualise in real time. More importantly it is scalable to any node arity. It allows for more emphasis to be placed on the nodes near the root since they naturally share less space. It can then be mapped to the Euclidean plane.

We take one aspect of this layout, the realisation that space increases as we move away from the circle centre, and that optimisations can be developed using this. This aspect is shown in Figure 15. We show how this is used when stepping through the algorithm for visualisation.

The algorithm again takes advantage of a recursive approach due to the tree structure of data. The basic algorithm is shown below.

There are two global variables, *length* and *maximum depth*. Length determines how long each branch in the tree will be. If the algorithm was to extend the hyperbolic metaphor further this could be changed to reflect the correct geometry. Having a constant length, however, meets requirement (4) of (Herman et al., 2000, Literature review, page 11) rules for aesthetically pleasing visualisations of trees. The *maximum depth* is more a requirement of the optimisation method. If the maximum depth is not known there is a potential for edge crossings. Noting the maximum depth allows the most efficient calculation of allowed space.

The algorithm can then be executed recursively:

The four steps of $PlotNode(LineAngle, RangeAllowed)$ are:

1. Perform hyperbolic optimisation
2. Split graph area into segments based on the number of children
3. Recurse down the tree using $PlotNode(LineAngle, RangeAllowed)$ as just calculated.
4. Plot line representing node

We now trace these steps through in further detail for the $PlotNode(LineAngle, RangeAllowed)$ function.

1. Perform hyperbolic optimisation

   This step works as seen in Figure 15. Given that we know the length of our move, the range we were previously allowed to plot in and the maximum depth of the tree, it is possible to calculate the extra area that can be used for plotting.

   The equation for this is:

$$ExtraArea = 2 \times \arcsin\left\{ \frac{maxdepth - depth}{\sin(\frac{Range}{2})} \right\} \tag{15}$$

2. Split graph into segments

   We have now been given an extra amount of range through moving outwards from the centre of the circle. This range is then split with each child being given an equal portion. This is only one method that could be used. A method that allocated range according to the size of the sub–tree was not used in this study because it felt that this could compromise the shape and its consistency.

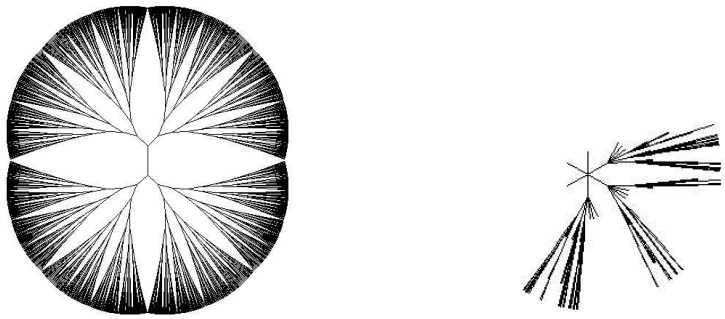
   There are a number of extra considerations at this step. These mainly deal with the special cases near the root node. For example we chose to limit the range by half if there is only one child of the root node. This is a major loss in information space but again maintained the consistent shape of programs.

3. Recurse down

   This method can again be called for each child. It calculates a new angle based on the angle provided and the range allowed.

4. Plot line

   Plot the actual line of this node based on the information given.



(a) A Full Binary Tree of Depth 10 (b) Example of a Real Population

Figure 16: Example Visualisation of a Symbol–Tree

In section 2.3.2 (page 11) we gave a list of criteria for the aesthetics of trees. How well does this method respect those criteria? The first criterion now needs to be changed for this method. We now define it as, “nodes of the same depth should be plotted on the same concentric circle”. Whilst our method does not meet this exactly it is quite close, enough so that the discrepancy is visually insignificant. All other criteria are met; the distance between siblings is fixed, nodes and edges are evenly distributed, edges have the same length, edges are straight lines and there are no edge crossings.

More importantly the method can easily scale to trees with any arity. This was a major problem with (Daida et al., 2005) and their method of graphing.

3.3.1 Similarity Matrix

We have called a new visualisation that we have created using the similarity measure, a *similarity matrix*. In this visualisation we calculate the similarity of every program to every other. If the results were to be displayed in tabular format they would be similar to Table 1. Each cell is a percentage value.

Table 1: Similarity Matrix for Four Generations

Population	1	2	3	4
1	100	98	93	80
2	98	100	95	85
3	93	95	100	90
4	80	85	90	100

Of course many of these values are redundant. There is no need to compare a Symbol–Tree against itself since the value will always be 100%. There is also redundancy in half the table since the comparison has already been made once already. When calculating the actual values for this table we only compare population n with populations $n+1, n+2, ..., m$. This removes the redundancy and provides a more meaningful display since a population is now compared for similarity to those succeeding it. This in effect shows a gradient curve of how the diversity changes throughout a run. More examples are shown in section 4.

3.3.2 Visualising Measures

For the other measures we use a more standard approach – generation versus measure. This has been a widely used representation in EC.

4 Experiments and Results

In the previous section we have shown that program trees can be simplified using our new data structure. From this a number of metrics regarding population diversity can be calculated to reveal general information about the type of search that is occurring. Using the data structure we can also visualise the order of program segments from an entire population or run. The question at this point is how can this new information be used to analyse and aid the genetic programmer?

To show this usage we perform experiments on three test problems. The first is the MAX problem. It is an interesting problem due to its low number of terminals and functions and the impact this will have on the Symbol–Tree. The second problem is that of the Artificial Ant. It was chosen due to its moderate number of terminals and functions and also being a well known test problem for GP. The last problem is an instance of symbolic regression similar to that described in section 2.1.1. This one was used in (Ekárt and Gustafson, 2004) and it is useful to continue the analysis of this problem since their work most closely resembles ours. With each of these experiments we give examples of each measure in graphical format. We also relate the visualisations of the Symbol–Tree data structure to the similarity matrix of various runs.

4.1 Max Problem - Small Space of GP Terminals and Functions

The MAX problem has been well studied in GP (Gathercole and Ross, 1996; Langdon and Poli, 1997; McPhee and Hopper, 1999). It represents a case where a mathematical function is to be maximised. This is made slightly more complex by only allowing multiplication, addition and a numeric value of one half. Given a maximum program depth, D, the optimal solution can be calculated as $4^{2^{D-4}}$. Our parameter settings for these runs can be seen in Table 2.

Table 2: Parameter Values For Max Problem

Parameter	Value
Type	Maximise Function Value
Fitness	The largest value that can be found.
Normalised Fitness	The largest value possible can be calculated using the function, $4^{2^{maxDepth-4}}$. The normalised fitness will therefore be the best possible value minus the value found.
End Condition	Difference between the best and found value being zero or the algorithm running for 500 generations.
Functions	+,*
Terminals	0.5
Crossover Rate	89%
Mutation Rate	10%
Elitism	1%
Selection Method	Roulette
Population Size	200
Number of Generations	500
Number of Runs	20
Min/Max Depth	3 – 7

The problem is an interesting one because it contains few symbols and requires a full tree to find the maximum solution. This maximum solution is also comprised of fairly regular blocks since the solutions are built from segments of addition that reach a value of two and then the successive multiplication of these segments to reach the global optimum value. It can also be achieved by using an additional plus instead of a multiplication to reach the required building block.

4.1.1 Motivation For Experiment

The MAX problem has a very small set of symbols used in the GP process. It is assumed that the resulting Symbol–Tree will be very compact. This may have an impact on informativeness of the data structure. Also, the functions used in the problem are associative and commutative. The implications of this will be discussed in the results.

4.1.2 Methodology

After completing the twenty runs of the problem we calculate each of the metrics, M_1, M_2, F_1, and create a visualisation of the similarity between each population in the run. This last visualisation is only done for two sample runs that have been found to be interesting through looking at the other measures. These usually correspond to the best and worst runs. These metrics are integrated with the visualisation of the data structure to incorporate the space being used by the data structure.

4.1.3 Results

One of the foremost aims of analysis is to not only learn more about the process, but also to associate changes with an improvement in performance. For each of the experiments we begin the discussion with a description of the changes in best fitness for the collection of runs.

In this case Figure 17 shows that the problem can be quite difficult since a number of runs are unable to find the optima and that the best run takes approximately 24,000 evaluations. The runs that failed can be said to have become trapped in local optima and most likely have converged on a Symbol–Tree that contains an addition node too high in the program tree.

Figure 18 is a composite measure relating both Figures 19 and 20 and the number of trees in the population. There are a number of interesting outliers in this graph. Runs 11 and 12 are significantly higher around generation 180 and Run 6 is also higher than the norm at generation 275. Interestingly all of these runs managed to find a solution indicating that a higher average F_1 value can indicate the movement out of a local optima.

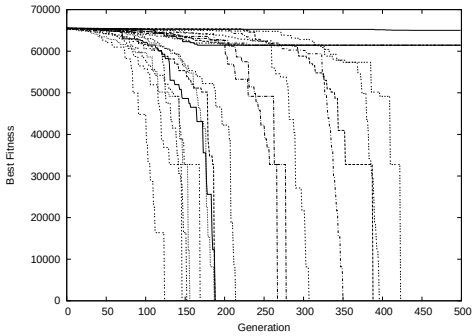


Figure 17: Best Fitness of 20 Runs– Max Problem

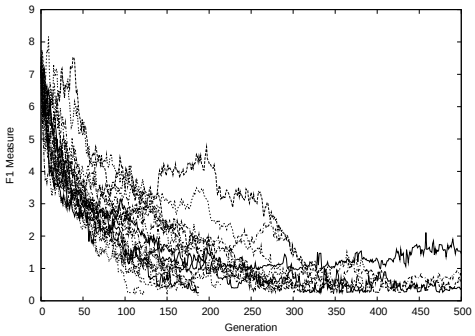


Figure 18: F_1 Measure of 20 Runs– Max Problem

Due to the nature of the MAX problem only full trees can be solutions. It is because of this property that many of the runs find a solution soon after the total number of genetic nodes in a population exceeds 25,000 (Figure 19). Runs that failed to converge towards a population of full trees were much less likely to succeed as can be seen by Run 19 which by the final generation only had an M_1 measure of around 17,000.

The opposite effect can be seen in the number of Symbol–Tree nodes that occur in populations that find solutions. These are usually around or below 50 Symbol-Tree nodes for the population when a solution is found. This low number of nodes indicates that there is very little diversity in the run near the end and that the run has converged.

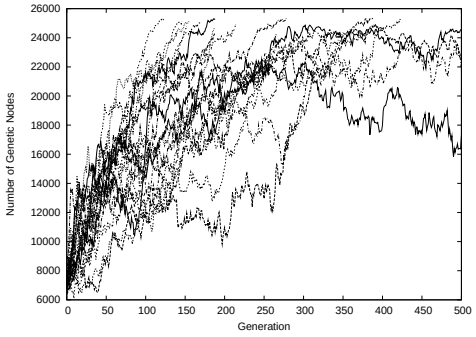


Figure 19: Number of Genetic Nodes of 20 Runs– Max Problem

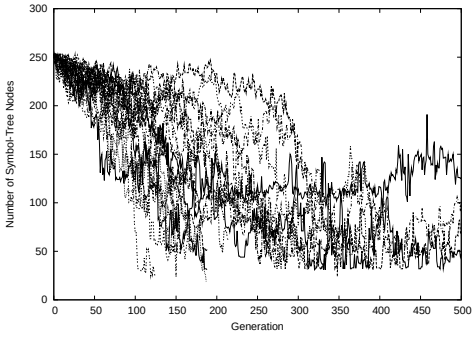


Figure 20: Number of Symbol–Tree Nodes of 20 Runs – Max Problem

We now focus on selected runs and a visualisation of their similarity matrices to examine the

progression of a run in detail. The visualisations are quite complex so we will initially discuss them in detail. We start with a discussion of Figure 21a which shows the similarity matrix for run one. Each line on the graph represents a string of comparisons. For example, the line that runs along the bottom is the comparison of the initial population against every other population other than itself. The second lowest line on the graph is the comparisons of generation one and every other generation after it. The first point of each line is one of the key indicators of convergence. It represents the similarity of the generation to the one preceding it, telling us how much change there is between consecutive populations. Thus, between the initial population and generation one there is roughly 85% similarity. The values slowly rise as the run progresses indicating a steady convergence to a particular Symbol–Tree.

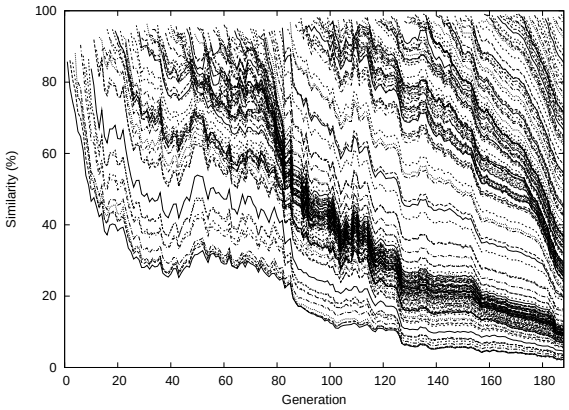
The gradient of the lines is also quite informative. They indicate, that although the similarity between consecutive runs may not be large, the similarity between runs after this drops quickly. This can be due to two factors: the populations are indeed quite dissimilar or the number of program nodes making up the Symbol–Trees is growing quickly. In this problem given the information from Figure 19 where most runs are increasing in the total number of genetic nodes, we suggest the latter. This is especially evident in the final 10 generations of Figure 21 since it was previously noted that most runs show a spike in the number of genetic nodes used to build a Symbol–Tree just before finding a solution.

The visualisations of the Symbol-Tree structure for a number of generations can be seen in Figure 21b–e. The first two visualisations show that there is still a large amount of variety in the symbol space. The tree's fullness indicates that a large number of trees are still being trialled. Visualisation (d) starts to see some convergence on the symbol space. There are now no trees that consist of a single 0.5 terminal. There are, however, still a large number of trees that start with the sub–optimal symbol, addition. The last tree indicates the fast convergence seen in the final few generations of (a). There are now no trees starting with the sub–optimal addition operator and the first two levels have converged on multiplication nodes. There are still some trees that sample sub–optimal configurations but many others have settled on correct branches.

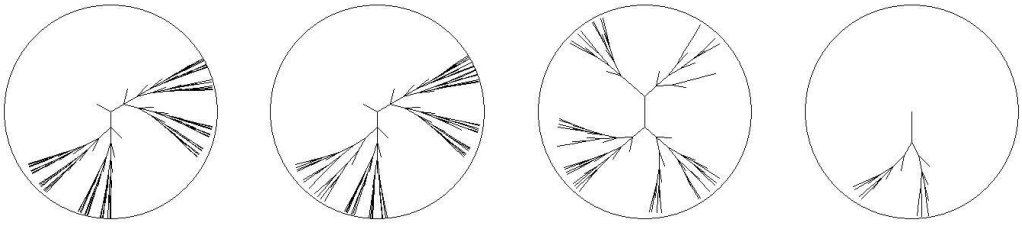
The similarity for a second run (run two) can be seen in Figure 22. Due to the large number of generations in this run we only graph the comparison of every 5th generation as the graph becomes very crowded and hard to read otherwise. The gradients of this similarity matrix is slightly different to that seen in Figure 21. After generation 250 the lines become much more horizontal and their similarity percentage rises as the run progresses. Since Figure 19 shows that most of the runs that failed reached a steady M_2 value, it can be assumed that the number of program nodes in the population is only a small influence. Thus, the similarity rise shows that there is a slow convergence on a Symbol–Tree type and that by generation 350 there is very little change to this. This may be an indication that the run has converged to a point where this will be very hard to break out of especially if the mutation operator rate is low.

One of the characteristics of this problem that is not shared with the other test problems is that the functions used in the program are both commutative and associative. In relation to how the Symbol–Tree is built this is a positive since the parameter order is inconsequential to the solutions derived.

A slight variant on this run was tested with tournament selection as the selection operator. This showed that runs were much more likely to find a solution in the first 60 generations or to never find a solution. This suggests that the diversity through using roulette selection is higher but that it is also less likely to capitalise on good programs in a population. We omit these visualisations from this work due to space considerations.



(a) Similarity Matrix – Run 1



(b) Gen. 0 (c) Gen. 60 (d) Gen. 100 (e) Gen. 188

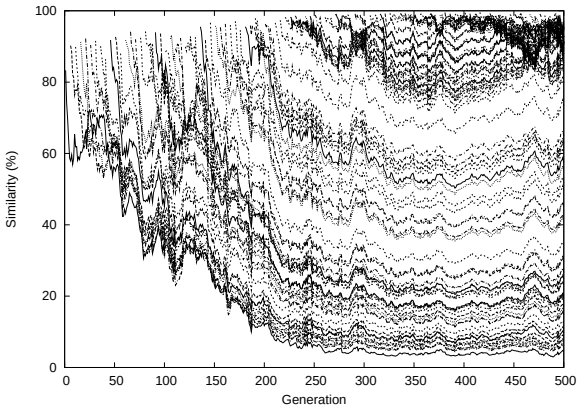
Figure 21: The Similarity of Populations Throughout A Run – Max Problem

4.2 Artificial Ant Problem – Moderate Number of GP Symbols

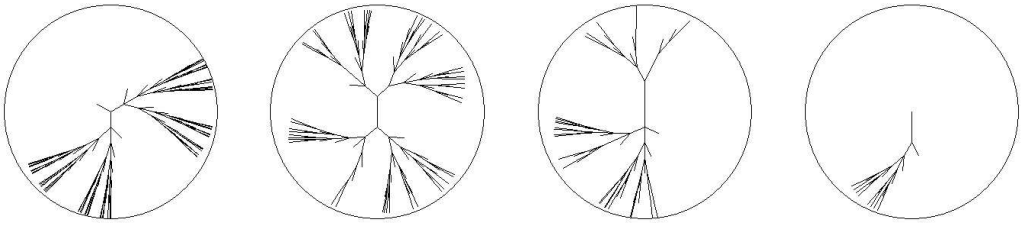
The Artificial Ant problem is an example of a robotics and planning domain. It involves creating a program that can navigate an ant over an irregular trail of food placed on a 32×32 grid. It has been described in (Koza, 1992, p.147-162) and has been termed a hard problem for GP in (Langdon and Poli, 1998) due to the ruggedness of the landscape and the lack of performance relationship between similar programs. We use the Santa Fe Trail (Koza, 1992, p.55) in this work since it is the most widely used test case. Programs are built through using the Prog2 and Prog3 functions which take two and three parameters respectively which will be evaluated in order. The function, IfFoodAhead, takes two parameters and will execute the first of these if there is food directly ahead of the ant or execute the second otherwise. Each of the terminals, moveForward, turnLeft, and turnRight takes a single move of which 500 are allowed. If a program does not use all 500 steps it will be re–run until they have been used.

4.2.1 Motivation For Experiment

The Artificial Ant is a problem that contains a moderate number of terminals and functions. Unlike the max problem it is not expected that the structure of a solution will take a certain shape. We experiment to see how the lack of expected structure affects the derived Symbol–Tree and what information can be gained about the problem using the measures available.



(a) Similarity Matrix – Run 2



(b) Gen. 0 (c) Gen. 150 (d) Gen. 210 (e) Gen. 400

Figure 22: The Similarity of Populations Throughout A Run – Max Problem

4.2.2 Methodology

The methodology of this experiment is similar to that of the previous one. We perform twenty runs of the artificial ant problem although this time only for 50 generations and with a population size of 500. From this we calculate our various measures across each of the runs and select two runs for further analysis.

4.2.3 Results

The graph of all runs best fitness values for this problem (Figure 23) shows that for these parameter settings the problem can be quite difficult. Only seven of the twenty runs found a solution to the problem, with some of the runs that remained trapped in local optima only able to find less than half the pieces of food on the trail (eg. run 17). The runs are also interesting in that some remained trapped in a local optimum for a long time whilst others are able to slowly improve.

The F_1 measure of this problem does not reveal a great deal (Figure 24). There is a pattern that runs leading up to a solution experience a sudden drop in value which is most likely caused by a convergence on a good solution and therefore a drop in the number of Symbol–Tree nodes as can be seen in Figure 26. A final point about this graph is that it has a much larger scale than in the MAX problem. This is not surprising since it indicates that a more complex symbol space is needed to solve the problem. It may also indicate that introns are able to exist.

Measures M_1 and M_2 (Figures 26 and 25) are also somewhat interesting. There is a large range of genetic nodes in the various runs from 43,000 nodes at the end of run five to around 10,000 in run nine. Whilst neither of these runs solved the problem there was also a spread in the sizes of the trees in runs that did solve the problem although these were typically less that 25,000 nodes.

Table 3: Parameter Values For Artificial Ant Problem

Parameter	Value
Type	Robotics and Planning
Fitness	The number of pieces of food eaten along the trail in 500 moves.
Normalised Fitness	The difference between the number of food pieces eaten and the total number of pieces on the trail.
End Condition	All the food being eaten or 50 generations being reached.
Functions	Prog2, Prog3, IfFoodAhead
Terminals	moveForward, turnLeft, turnRight
Crossover Rate	89%
Mutation Rate	10%
Elitism	1%
Selection Method	Roulette
Population Size	500
Number of Generations	50
Number of Runs	20
Min/Max Depth	3 – 7

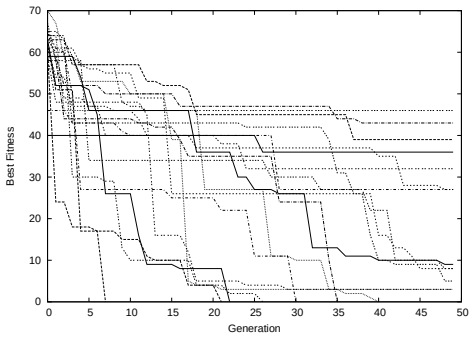


Figure 23: Best Fitness of 20 Runs – Ant Problem

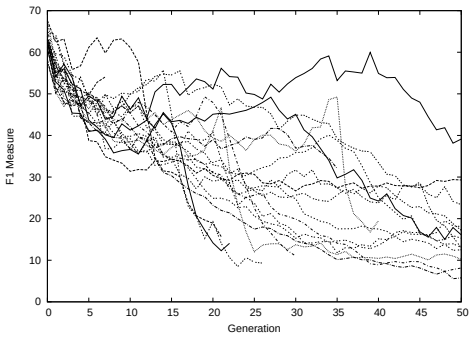


Figure 24: F_1 Measure of 20 Runs – Ant Problem

As already mentioned the number of nodes in the Symbol–Tree fell dramatically leading up to a solution being found. More than this, most runs dropped from 1,800 nodes to around 600 before the end of the run. This indicates that the runs converged quite quickly on a program type. In this problem it may best to restart the run if after dropping below 600 Symbol–Tree nodes and progressing ten generations a solution is not found.

We now further examine two of the runs using the similarity matrix and visualisation of the Symbol–Tree data structure. Run number four was an interesting one because it was the best performing run to not find the solution. From the fitness graph in Figure 23 it can be seen as the lowest line at generation 50. Interestingly it had a low fitness from around the 17th generation where it had made a big improvement from the generation before with a fitness of 39. Soon after this improvement there was a steep drop in the number of Symbol–Tree nodes from 1807 to 293 from generation 19 to 25.

This drastic change is also reflected in the similarity matrix in Figure 27. The run is converging slowly between the first to 20th generation before this lucky solution is found in generation 16. By three generations later there is a massive change in the population as seen by the 60% similarity

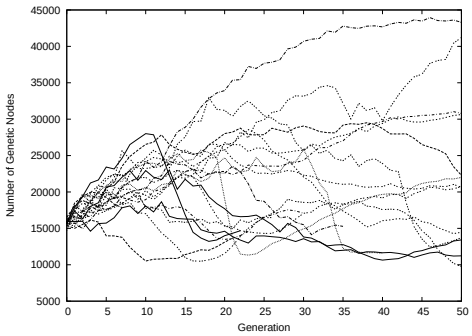


Figure 25: Number of Genetic Nodes of 20 Runs – Ant Problem

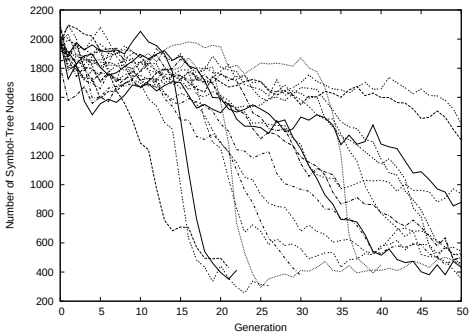
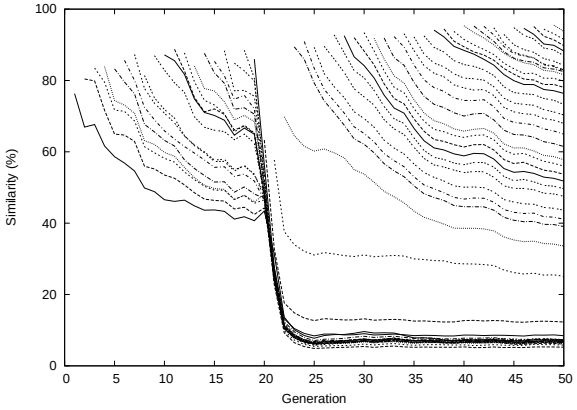


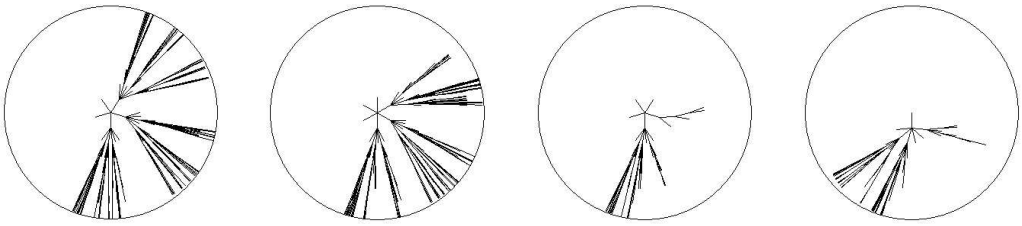
Figure 26: Number of Symbol–Tree Nodes of 20 Runs – Ant Problem

between 20 and 21 and the 70% similarity between 21 and 22. After this there is a slow convergence as can be seen in the slow rise of similarity in consecutive populations. This can be seen by the slow rise of the first point in each line after the change.

The problem in this run is that there seems to be a loss of diversity after the jump in fitness. Since roulette selection uses the best program, that best program can come to quickly dominate the population, which afterwards is very hard to improve. The use of tournament selection in this case may have stopped this since it would have been less influenced by the scale of the difference in best fitness.



(a) Similarity Matrix – Run 4



(b) Gen. 17 (c) Gen. 21 (d) Gen. 25 (e) Gen. 37

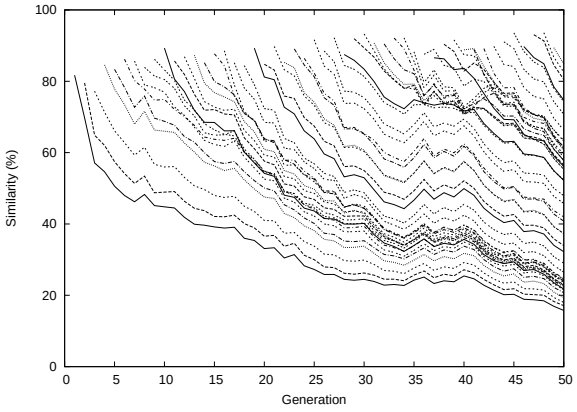
Figure 27: The Similarity of Populations Throughout A Run – Ant Problem

The second run to be analysed is run 17. It was the worst performing of the runs and did not find even half the food after evolving over 50 generations. In fact it had evolved a program to collect 43

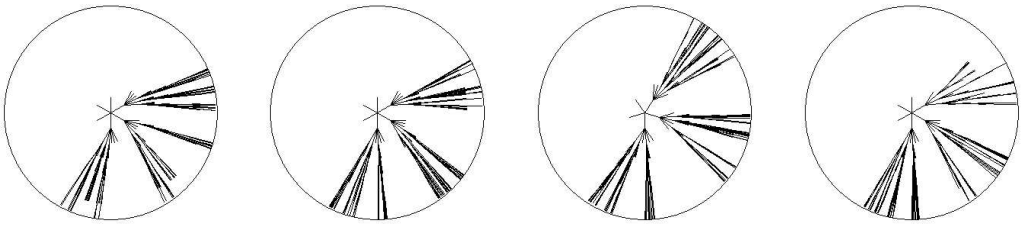
pieces of food by the 12 generation and never improved on this. It is also interesting due to having both large numbers of genetic nodes and Symbol-Tree nodes in its populations.

The similarity matrix (Figure 28) is interesting because it shows very few characteristics of converging on a solution. The trend is similar throughout the entire run with no regions of increased change or large changes in the similarity of consecutive populations. This suggests that convergence is needed to find a solution and that a lack thereof is a characteristic of runs that do very poorly on this problem. More work would have to be done to confirm this suspicion.

The Symbol–Tree visualisations (Figure 28b–e) show that there is a large space of symbols being sampled. This is dissimilar to other runs we have seen where the programs converge on a fairly small range of symbols. It either suggests that there is a full tree of varied symbols being tested in the most promising program or else the run was still conducting a blind random search. This second point seems possible.



(a) Similarity Matrix – Run 17



(b) Gen. 0 (c) Gen. 19 (d) Gen. 27 (e) Gen. 50

Figure 28: The Similarity of Populations Throughout A Run – Ant Problem

The results of this run showed that the measures were again quite informative. The detailed analysis showed a large difference in the similarity matrix of a one run that very quickly converged and another that seemed to never home in on a solution. Being able to see this kind of behaviour could be used to help guide a run in an interactive way.

4.3 Symbolic Regression - High Number of GP Symbols

We have often drawn parallels and made comparisons between our work and that of Ekárt and Gustafson and their iTree data structure (Ekárt and Gustafson, 2004). Whilst they only gave a minimal case study it seems reasonable that we should repeat and compare our work to theirs. Thus we use their instance of a symbolic regression that attempts to evolve through minimising the error of a function to a series of points generated by the equation $(x - 0.2)^2(x + 0.5)(x - 0.5), x \in [-1, 1]$.

Table 4: Parameter Values Symbolic Regression Problems

Parameter	Value
Type	Symbolic Regression - Error Driven Evolution
Function	$(x-0.2)^2(x+0.5)(x-0.5)$ with $x \in [-1,1]$
Fitness	$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(gp(x_i)-y_i)^2}$. The sum of squared error over 50 randomly generated x values.
Functions	+,*,/ (Protected Division)
Terminals	X, random constants $\in [-1,1]$
Crossover Rate	90%
Mutation Rate	10%
Elitism	None
Population Size	100
Number of Generations	50

Many researchers have studied instances of symbolic regressions beginning with Koza (Koza, 1992) to (Chaudhri et al., 2000; Daida et al., 2001) with their tunably difficult problem, binomial–3. This is an important variety of test problem because there are many real world applications where it is useful to find a mathematical expression that can describe a series of points.

The parameters used in our run can be seen in table 4. We set the fitness of the problem to be the square root of the average squared error. This instance of the problem only has to use the functions, $+$, $\times$, and $/$ (protected division). This make the problem significantly easier than if we had included a greater suite of mathematical operators. In this cases the operators are a very close fit with the actual problem. The terminal set consists of x, and *random constants* in the range [-1,1]. These terminals create a random value when they are first evaluated and then maintain this value as a constant throughout the rest of their lifetime. The minus operator is omitted since it can be found through using negative values. In line with Ekárt and Gustafson we use tournament selection with a size of ten and a population size of 100 that is evolved for 50 generations.

Ekárt and Gustafson's main experiment was to explore the program trees of programs that were either very good ($fitness < 0.03$) or very bad ($fitness > 1$). These were split up into two sub–sets and values taken for M_1, M_2, and a measure of imbalance. We take this idea a step further by investigating runs that succeed and fail in much the same way as we have for the previous experiments.

4.3.1 Motivation For Experiment

As mentioned previously this experiment was used in (Ekárt and Gustafson, 2004) to test their iTree data structure. On top of this it is a very common style of test problem. As a final motivation, the random constant terminal type allows for a large number of different terminals in regards to the Symbol–Tree. This is a factor that has not been investigated in the previous two experiments where the symbol space was limited.

4.3.2 Methodology

We perform 20 runs of the problem and then visualise the runs using best fitness, M_1, M_2, and F_1. We then choose two interesting runs and expand on them using the similarity matrix and a selection of visualisations of the Symbol–Tree data structure at various stages in the run.

4.3.3 Results

The results of all runs best fitness (Figure 29) shows that this is a problem which GP finds difficult due to the number of local optima. Most of the runs have reached their best value or close to it by the 12th generation. The most identifiable exception to this was run 1 which took much longer to converge and also obtained the best result over the 50 generations.

Run 1 is again prominent in Figure 30 of the F_1 measure. It is a significant outlier before generation 20. The majority of the runs are fairly tightly constrained to a range between a value of 1 and 5. This may indicate that in comparison to the Ant problem there is a fairly small number of Symbol–Tree nodes per number of genetic nodes in the population.

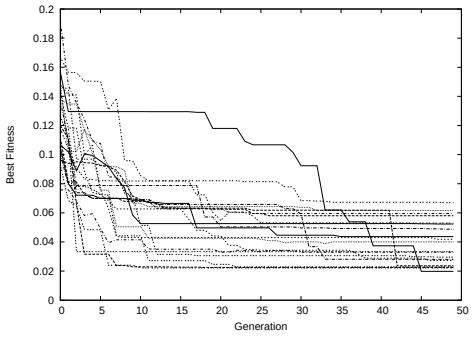


Figure 29: Best Fitness of 20 Runs – Symbolic Problem

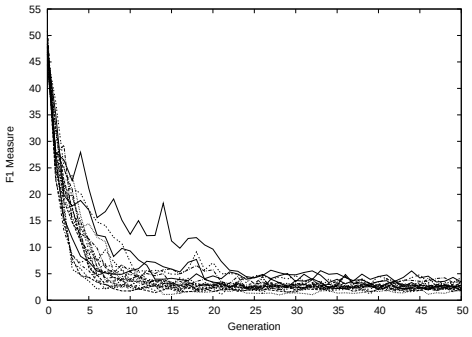


Figure 30: F_1 Measure of 20 Runs – Symbolic Problem

Figure 31 is interesting in that it appears there are fairly constrained levels of program size that fall in the local optima for this problem. It appears very ordered with runs sticking to a single number of genetic nodes per population before occasionally jumping up or down a range. This movement is usually in an upward direction possibly indicating that better solutions can more often be found in larger trees. Run 1, which was of interest in Figures 29 and 30 has a final value of 2288, falls near the middle of the ranges.

Despite the varying levels of genetic nodes in the runs the number of Symbol–Tree nodes (Figure 32) is fairly constant for all runs. Most runs had converged on the range of 30 to 80 nodes per population. This may indicate that there are only a small amount of symbols used in the evolved programs or that there are frequent common patterns in the programs.

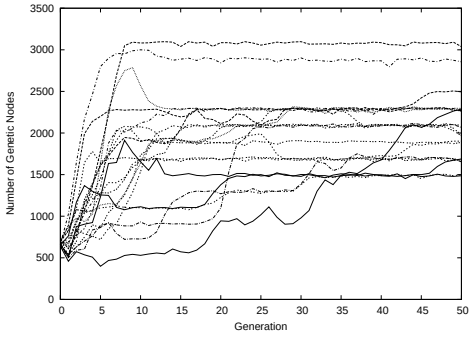


Figure 31: Number of Genetic Nodes of 20 Runs – Symbolic Problem

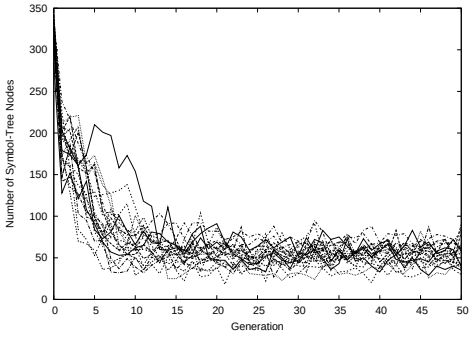
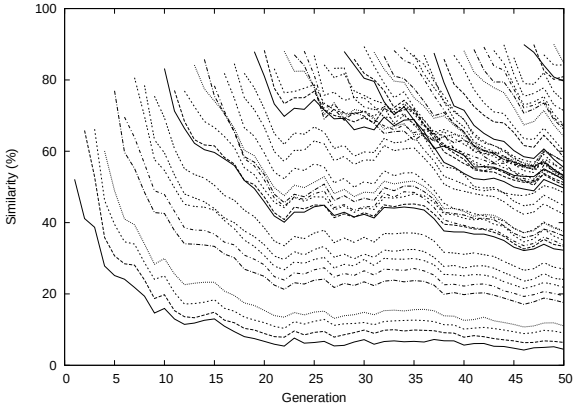


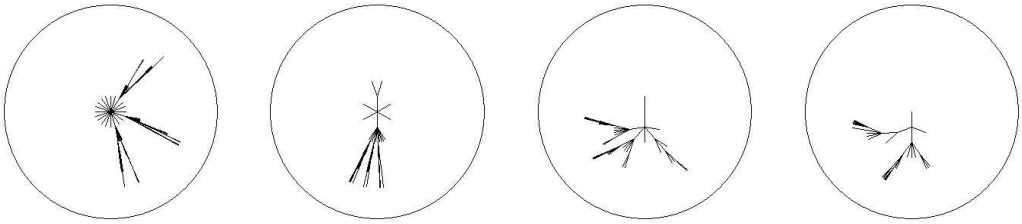
Figure 32: Number of Symbol–Tree Nodes of 20 Runs – Symbolic Problem

Run 1 was the most interesting run from the previous plots. It seemed to converge at a much slower rate. The similarity matrix (Figure 33) is not so striking however. It can be seen by the slow rise in similarity of consecutive populations (the first point on each line) that there is some convergence, there is no striking increase as we have seen in some previous visualisations. This indicates that the run is still improving and could possibly perform better still.

The visualisations of the Symbol–Tree structures indicate that there is a large space of symbols being evaluated, particularly in the early generations. For example in the initial generation (b), there is a circle of depth one branches. These are most likely random constants. The large numbers cause the graph to be very hard to read since the space is very constrained when it is split up evenly as we do in our current method. Whilst this is eased in the latter generations where the runs have converged on solutions that do not contain early constants, there existence still causes crowding in the visualisations. This run may be a candidate for either allowing the create tree algorithm to identify all random constants as the same node type or to ignore them when visualising the structure. Both methods lose information about the symbol space but may make the plots clearer and easier to understand particularly if the functions are having a major impact on the solutions.



(a) Similarity Matrix – Run 1



(b) Gen. 0 (c) Gen. 9 (d) Gen. 37 (e) Gen. 50

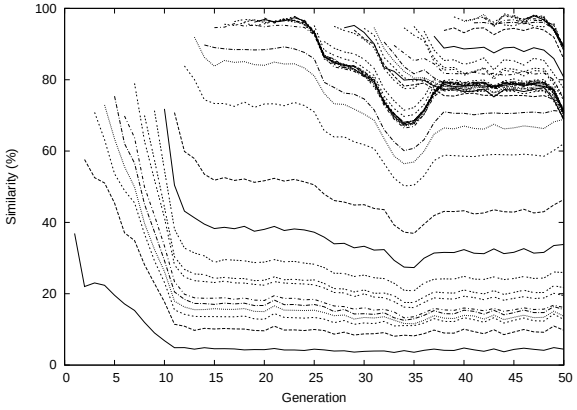
Figure 33: The Similarity of Populations Throughout A Run – Symbolic Regression Problem

Run 16 has initially a much more striking similarity matrix (Figure 34). It was one of the worst performing runs. After starting off poorly it found a good solution and seemed to have converged on this by the 10th generation. After this time it made very little improvement and seemed to be stuck in a local optima. In all of the other measures it seemed fairly typical.

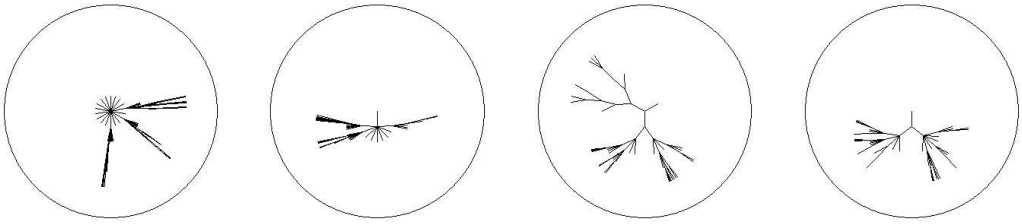
This search for a solution can be seen in the first 10 generations. After the 10th generation however we see a very quick convergence represented by the rising similarity of consecutive populations. By generation 15 consecutive populations are almost identical indicating that the population has converged on a common program or symbol space. The run manages to break out of this at the 25th

generation and this is probably the cause in an increase in fitness between the 25th and 30th generations. Soon after the 35th generation the similarity of consecutive populations begin to rise again. The run again appears to have converged by generation 40. The final few generations show another change possibly indicating that the run was about to move on to another optima.

The constant converge – break out pattern was quite interesting in this run. The fast convergence this time cannot be caused by roulette selection. It is positive to note that the run was still working towards a new solution at the final generation and that it could have benefited from further evolution.



(a) Similarity Matrix – Run 16



(b) Gen. 0 (c) Gen. 10 (d) Gen. 15 (e) Gen. 37

Figure 34: The Similarity of Populations Throughout A Run – Symbolic Regression Problem

The two runs analysed in detail provided quite interesting analysis. Run 1 was very slow to converge and at the 50th generation seemed still to have not settled on a particular program type. Run 17 on the other hand was very quick to converge on potential solutions, converging twice and yet managing to break out of these local optima. It would be interesting to examine on a large scale which style of evolution performs better.

4.4 Discussion of Results

We have performed runs on three different problems; MAX, Artificial Ant, and an instance of symbolic regression. For each of the problems 20 runs were completed and plots of best fitness, M_1, M_2, and F_1 were calculated for each generation. Using these results two runs from each problem were selected for further analysis using a similarity matrix and some selected visualisation of the Symbol–Tree data structure. From these visualisations predictions were made about the state of the run and how it progressed in its search.

The similarity matrices in particular led to a detailed analysis of the various runs. The rise in similarity between consecutive generations could clearly be seen which could allow a GP practitioner

to stop a run when they believed generations had converged and were unlikely to find new results. A drop in convergence could also be seen in a number visualisations. This probably indicates that a new potential solution has just been found and a run should be extended to allow time to explore this new region thoroughly.

Whilst informative, the other measures were not felt to have revealed as much as the similarity matrix. In particular the F_1 measure often revealed that there was not much difference between the runs. Further analysis of this measure on different problems in regards to the scale of the values might indicate that it is more useful on determining the entropy of solutions. For example the result on the Ant problem (Figure 24) had much larger values than the other problems. This may have indicated that there was a much greater spread of functions and terminals throughout the programs than there was in the other two experiments. This is an area for further research.

5 Conclusion

Genetic programming is a complex domain and there is a need for analysis to further understand the dynamics of a run. We proposed a *symbol space*, a simplified representation of the space of programs trees that takes into account what symbols exist, at what depth, and in what order. We were motivated with a need to be able to calculate measures relating to the symbol space and thus proposed the following research questions:

1. *Is there a data structure that can efficiently maintain and calculate measures related to the symbol space?*

   We developed a new data structure, the *Symbol–Tree*, that was inspired by (Ekárt and Gustafson, 2004) and their iTree data structure. We explored both the time and space complexity of the Symbol–Tree to show that it would be no worse than maintaining the entire population with the addition of the artificial root node. Using the data structure we were able to show that it could calculate a number of measures more efficiently than without the data structure.

2. *How can we use this data structure and the associated metrics to create visualisations that can aid in the understanding of the GP search process?*

   Our visualisations took three forms. The first was a visualisation of the Symbol–Tree data structure using a circular layout. This was inspired by (Daida et al., 2005) method of visualisation using a circular grid. As their method is limited to only trees with a small program arity (the number of parameters a particular function can have), we developed a new method that worked by sub–dividing the available space and was optimised using a hyperbolic optimisation that realises that there is extra space available as a line moves away from the center of the circle.

   Our second method involved a more orthodox plot of the F_1, M_1, and M_2 measures against generation.

   The third method involved creating a similarity matrix that compared every generation to every other. From this the similarity values of a generation compared to every other generation following it could be plotted.

3. *Can these visualisation be used to explore the dynamics of a run?*

   The visualisation were used to explore data from three test problems, MAX, Artificial Ant, and an instance of symbolic regression. In each of these problems 20 runs were performed and visualisation done.

Visualisation of the data structure showed that most runs converge on a compact Symbol–Tree. It is unclear at this stage what impact this has on the fitness of a run.

Analysis of the visualisation of the F_1, M_1, and M_2 measures against generation showed that:

(a) There is usually a sudden drop in the number of nodes in a Symbol–Tree shortly before finding a solution.

(b) Various problems show quite different characteristics in regards to program growth.

(c) There was a large variety in the F_1 measure but that lower values typically indicated a solution was about to be found.

The similarity matrix visualisation turned out to be very effective in exploring the dynamics of a run. It was used on selected runs that had interesting characteristics in the previous visualisations. It was able to reveal:

(a) Similarity of consecutive generations slowly rises.

(b) Convergence on a local optima could be seen. This information could be used to stop a run when the chance of improvement are low.

(c) Lack of convergence could be seen. This information could be used to extend a run since there is a strong chance of improvement.

There are a number of avenues that this work could be extended. Firstly, we only investigated a small number of problems of what could be called "toy" problems. Some real world problems should be reviewed to extend the analysis. Results so far using the visualisations showed it is possible to identify cases where a run should be stopped and also where a run should be given more time to converge. To test this, a real–time analysis would need to be conducted as a run was progressing to see if the predictions hold true without apriori knowledge of what occurs later in the run. If this approach was successful the process could be taken to a real–time guiding of the GP search with a practitioner being able to increase various parameters based on predicted state of the run.

In the literature review we showed that there are a number of quite complex methods for visualising trees. These were not chosen in favour of a quicker and simpler representation. Given the complexity of some of the trees, particularly with a problems that have a large symbol space, it may be useful to investigate these approaches further. Although they may require more interaction than the method developed the nuances that could be further examined may be revealing.

Acknowledgements

I would like to thank my supervisor, Vic Ciesielski for his invaluable feedback and guidance and David and Anne Foster for their diligent proof reading and economic support throughout the year.

References

Bäck, T., Hammel, U., and Schwefel, H.-P. (1997). Evolutionary computation: Comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1(1):3–17.

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (2001). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications (3rd edition)*. Morgan Kaufmann, dpunkt.verlag.

Barlow, M., Galloway, J., and Abbass, H. A. (2002). Mining evolution through visualization. In Smith, T., Bullock, S., and Bird, J., editors, *Beyond Fitness: Visualising Evolution — Work. of Artificial Life VIII: 8th Int. Conf. Simulation and Synthesis of Living Systems*.

Beyer, H.-G. and Schwefel, H.-P. (2002). Evolution strategies: a comprehensive introduction. *Natural Computing*, 1:3–52.

Burke, E. K., Gustafson, S. M., Kendall, G., and Krasnogor, N. (2002a). Advanced population diversity measures in genetic programming. In *PPSN*, pages 341–350.

Burke, R., Gustafson, S., and Kendall, G. (2002b). A survey and analysis of diversity measures in genetic programming. In et al., W. B. L., editor, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 716–723, New York. Morgan Kaufmann Publishers.

Carrière, J. and Kazman, R. (1995). Research report: Interacting with huge hierarchies: Beyond cone trees. In Gershon, N. D. and Eick, S., editors, *Proc. IEEE Symp. Information Visualization, InfoVis*, pages 74–81. IEEE Computer Society Press.

Chaudhri, O. A., Daida, J. M., Khoo, J. C., Richardsons, W. S., Harrison, R., and Sloat, W. J. (2000). Characterizing a tunably difficult problem in genetic programming. In *GECCO*, pages 395–402.

Ciesielski, V. and Li, X. (2004). Analysis of genetic programming runs. In McKay, R., editor, *Asia-Pacific Workshop on Genetic Programming*.

Daida, J. (2004). Considering the roles of structure in problem solving by a computer. In O'Reilly, U.-M., Yu, T., Riolo, R. L., and Worzel, B., editors, *Genetic Programming Theory and Practice II*, chapter 5. Kluwer, Ann Arbor.

Daida, J., Ward, D., Hilss, A., Long, S., Hodges, M., and Kriesel, J. T. (2004). Visualizing the loss of diversity in genetic programming. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, volume 2, pages 1225–1232, Portland, Oregon. IEEE Press.

Daida, J. M., Bertram, R. R., Stanhope, S. A., Khoo, J. C., Chaudhary, S. A., Chaudhri, O. A., and Polito II, J. A. (2001). What makes a problem GP-hard? analysis of a tunably difficult problem in genetic programming. *Genetic Programming and Evolvable Machines*, 2(2):165–191.

Daida, J. M. and Hilss, A. M. (2003). Identifying structural mechanisms in standard genetic programming. In et al., E. C.-P., editor, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1639–1651, Chicago. Springer-Verlag.

Daida, J. M., Hilss, A. M., Ward, D. J., and Long, S. L. (2003a). Visualizing tree structures in genetic programming. In et al., E. C.-P., editor, *Genetic and Evolutionary Computation – GECCO-2003*, pages 1652–1664, Berlin. Springer-Verlag.

Daida, J. M., Hilss, A. M., Ward, D. J., and Long, S. L. (2005). Visualizing tree structures in genetic programming. *Genetic Programming and Evolvable Machines*, 6(1).

Daida, J. M., Li, H., Tang, R., and Hilss, A. M. (2003b). What makes a problem GP-hard? validating a hypothesis of structural causes. In *GECCO*, pages 1665–1677.

de Jong, E. D., Watson, R. A., and Pollack, J. B. (2001). Reducing bloat and promoting diversity using multi-objective methods. In et al., L. S., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 11–18, San Francisco, California, USA. Morgan Kaufmann.

Ekárt, A. and Gustafson, S. (2004). A data structure for improved GP analysis via efficient computation and visualisation of population measures. In Keijzer, M., O'Reilly, U.-M., Lucas, S. M., Costa, E., and Soule, T., editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 35–46, Coimbra, Portugal. Springer-Verlag.

Ekárt, A. and Németh, S. Z. (2002). Maintaining the diversity of genetic programs. In Foster, J. A., Lutton, E., Miller, J., Ryan, C., and Tettamanzi, A. G. B., editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 162–171, Kinsale, Ireland. Springer-Verlag.

Fogel, D. B. (1998). An introduction to evolutionary computation. In Fogel, D. B., editor, *Evolutionary Computation: the fossil record*, pages 1–2. IEEE Press, Piscataway, NJ.

Fogel, D. B. and Fogel, L. J. (1995). An introduction to evolutionary programming. In *Artificial Evolution*, pages 21–33.

Gathercole, C. and Ross, P. (1996). An adverse interaction between crossover and restricted tree depth in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 291–296, Stanford University, CA, USA. MIT Press.

Gustafson, S. M. (2004). *An Analysis of Diversity in Genetic*. PhD thesis, The University of Nottingham.

Herman, Melançon, G., and Marshall, M. S. (2000). Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43.

Holland, J. H. (1962). Outline for a logical theory of adaptive systems. *J. ACM*, 9(3):297–314.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press.

Johnson, B. and Shneiderman, B. (1991). Tree maps: A space-filling approach to the visualization of hierarchical information structures. In *IEEE Visualization*, pages 284–291.

Keskin, C. and Vogelmann, V. (1997). Effective visualization of hierarchical graphs with the cityscape metaphor. In *Workshop on New Paradigms in Information Visualization and Manipulation*, pages 52–57.

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

Krasnogor, N. and Gustafson, S. (2003). Visualising populations of rooted labeled trees on a lattice.

Lamping, J., Rao, R., and Pirolli, P. (1995). A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proceedings CHI'95*. ACM.

Langdon, W. B. (1996). Evolution of genetic programming populations. Research Note RN/96/125, University College London, Gower Street, London WC1E 6BT, UK.

Langdon, W. B. (2000). Size fair and homologous tree crossovers for tree genetic programming. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119.

Langdon, W. B. and Poli, R. (1997). An analysis of the MAX problem in genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 222–230, Stanford University, CA, USA. Morgan Kaufmann.

Langdon, W. B. and Poli, R. (1998). Why ants are hard. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 193–201, University of Wisconsin, Madison, Wisconsin, USA. Morgan Kaufmann.

McPhee, N. F. and Hopper, N. J. (1999). Analysis of genetic diversity through population history. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1112–1120, Orlando, Florida, USA. Morgan Kaufmann.

Michalewicz, Z. and Michalewicz, M. (1997). Evolutionary computation techniques and their applications. In *Intelligent Processing Systems*, volume 1, pages 14–25, Beijing.

Pohlheim, H. (1999). Visualization of evolutionary algorithms – set of standard techniques and multidimensional visualization. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proc. of the Genetic and Evolutionary Computation Conf. GECCO-99*, pages 533–540, San Francisco, CA. Morgan Kaufmann.

Robertson, G. G., Mackinlay, J. D., and Card, S. K. (1991). Cone trees: Animated 3D visualizations of hierarchical information. In et al., R., editor, *Proceedings of the CHI '91 Conference*, pages 189–194.

Schweizer, B. and Sklar, A. (1960). Statistical metric spaces. *Pacific Journal of Mathematics*, 10(1):313–334.

Tufte, E. R. (1983). *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Connecticut.

van Wijk, J. and van de Wetering, H. (1999). Cushion treemaps: visualization of hierarchical information. In *1999 IEEE Symposium on Information Visualization (INFOVIS '99)*, pages 73–78, Washington - Brussels - Tokyo. IEEE.

Wineberg, M. and Oppacher, F. (2003a). Distance between populations. In *GECCO*, pages 1481–1492.

Wineberg, M. and Oppacher, F. (2003b). The underlying similarity of diversity measures used in evolutionary computation. In *GECCO*, pages 1493–1504.