A Depth Controlling Strategy for Strongly Typed Evolutionary Programming

Claire J. Kennedy
Department of Computer Science
University of Bristol
Bristol BS8 1UB, U.K.
kennedy@cs.bris.ac.uk

Christophe Giraud-Carrier
Department of Computer Science
University of Bristol
Bristol BS8 1UB, U.K.
cgc@cs.bris.ac.uk

Abstract

This paper presents a dynamic strategy for monitoring the depth of program trees evolved by STEPS (Strongly Typed Evolutionary Programming System). STEPS evolves higher-order functional programs in the form of trees, which are allowed to grow or shrink to fit the size of the problem, via specialised genetic operators. Thus, the need for arbitrary cut-off mechanisms is eliminated.

1 INTRODUCTION

Most evolutionary algorithms rely on fixed-length representations. Clearly, such representations simplify implementations. However, they often require the user to have some knowledge of the appearance and structure of the final solution. More recently, variable-length representations have been used to alleviate these limitations. One notable example is in the area of Genetic Programming where programs in the form of parse trees are evolved (Koza 1992). Although, more flexible and less demanding of prior knowledge, the variable- length representation can lead to a general increase in the depth and size of the individuals in the population over successive generations.

This increase in size is called *bloat* and is due to *introns* (Angeline 1994, Blickle and Thiele 1994), i.e., extra pieces of information/code that do not contribute anything to the fitness of the individual. Although not useful in terms of fitness, introns seem to protect fit pieces of code from the destructive effects of crossover (Angeline 1996). Unfortunately, introns tend to spread throughout the population by *Hitch-Hiking* along with the good pieces of code they are protecting during evolution (Tackett 1994). Hence, if left unchecked, introns may allow the individuals in the population to bloat uncontrollably, thus putting a considerable strain on the computational resources. Consequently, it is generally agreed that some form of restriction on the depth and/or size of the trees generated during evolution has to be incorporated into the system.

In general, size restrictions are implemented statically in a somewhat ad hoc way. For example, a size cut-off or maximum depth value are used in GP. These parameters are set a priori by the user. Within an evolutionary paradigm, there is something rather unnatural about such hard-coded bounds. It would be more elegant (and more in keeping with the philosophy of evolutionary computing) to let solutions grow or shrink according to the demands of the environment, as measured by the fitness function.

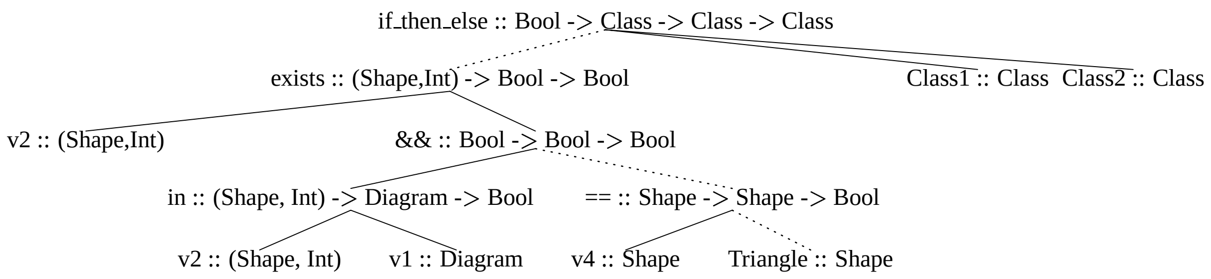
STEPS (Strongly Typed Evolutionary Programming System) provides such flexibility through the use of specialised genetic operators. Large/deep trees are allowed in the population, thus preserving potentially good genetic material that would otherwise be lost if a hard bound on tree size/depth were used. However, large trees have a higher probability of being pruned than smaller trees, whilst small trees have a higher probablity of being grown.

The paper is organised as follows. Section 2 describes the STEPS Learning algorithm. Section 3 reviews some of the existing depth controlling options available with variable length representations. Section 4 describes the alternative depth controlling strategy available with STEPS and reports and discusses some preliminary experiments. Finally, section 5 concludes the paper.

2 STEPS

2.1 REPRESENTATION

The evolutionary approach used in this study is that of STEPS - Strongly Typed Evolutionary Programming System (Kennedy and Giraud-Carrier 1999). STEPS evolves Escher programs in the form of parse trees. Escher is a strongly typed functional logic language that provides higher-order functionality such as set processing (Lloyd



if_then_else :: Bool -> Class -> Class -> Class
exists :: (Shape,Int) -> Bool -> Bool
Class1 :: Class
Class2 :: Class
v2 :: (Shape,Int)
&& :: Bool -> Bool -> Bool
in :: (Shape, Int) -> Diagram -> Bool
== :: Shape -> Shape -> Bool
v2 :: (Shape, Int)
v1 :: Diagram
v4 :: Shape
Triangle :: Shape

Figure 1: A program tree exhibiting variable consistency violation

1995). STEPS is mainly used for concept learning where the concept descriptions are of the form `if Cond then` C_i `else S`. Here the condition, `Cond`, is a boolean expression, C_i is a class label and `S` is either the default class label or another `if_then_else` statement.

The training examples provided to STEPS are represented as closed terms, which give a compact and self-contained description of each example (Flach *et al.* 1998). Selector functions in the form of subtrees are used to pull information out of the closed terms in order to make inferences about them. An algorithm has been designed to automatically generate the appropriate selector functions associated with a set of types (Bowers *et al.* 1999). This algorithm has been adapted to return selector functions in partially created subtree form. This adapted process is described in detail in (Kennedy and Giraud-Carrier 1999). These selector function subtrees form the alphabet for the problem along with some connective functions (conjunction, disjunction and negation) and the problem dependent constants.

2.2 EVOLUTIONARY APPROACH

Since Escher is a strongly typed language, an evolutionary paradigm that incorporates type information is necessary so that only type-correct programs are generated during learning. Traditional program tree based evolutionary paradigms, such as Genetic Programming (GP), assume the closure of all functions in the body of the program trees (Koza 1992). This means that every function in the function set must be able to take any value or data type that can be returned by any other function in the function set. While this characteristic simplifies the genetic operators, it limits the applicability of the learning technique and can lead to artificially formed solutions. In order to overcome this problem, a type system was introduced to standard GP to give Strongly Typed Genetic Programming (STGP) (Montana 1995). STGP helps to constrain the search space by allowing only type correct programs to be considered. STEPS extends the STGP approach to allow the vast space of highly expressive Escher concept descriptions to be explored efficiently.

2.2.1 Creation of Initial Population

Program trees in the initial population are formed by randomly selecting subtrees from the alphabet. However subtrees selected to fill a blank slot in a partially created program tree must satisfy certain constraints so that only valid Escher programs are created.These constraints are type and variable consistency. In order to maintain type consistency, each node in a subtree in the alphabet is annotated with a type signature indicating its argument and return types. A subtree selected to fill in a blank slot must be of the appropriate return type.

In order to maintain variable consistency, the local variables in a subtree selected to fill in a blank slot in the partially created program tree must be within the scope of a quantifier. In addition, all quantified variables in a program tree must be used in the conditions of their descendant subtrees to avoid redundancy. The program tree in Figure 1 provides an example of variable consistency violation.

The addition of the subtree rooted at `== :: Shape -> Shape -> Bool` in Figure 1 violates variable consistency as the variable `v4 :: Shape` is not within the scope of a quantifier. In addition variable consistency is violated by not using the quantified variable `v2 :: (Shape, Int)`.

2.2.2 Modified Crossover

The requirement for type and variable consistent program trees needs to be maintained during the evolution of the

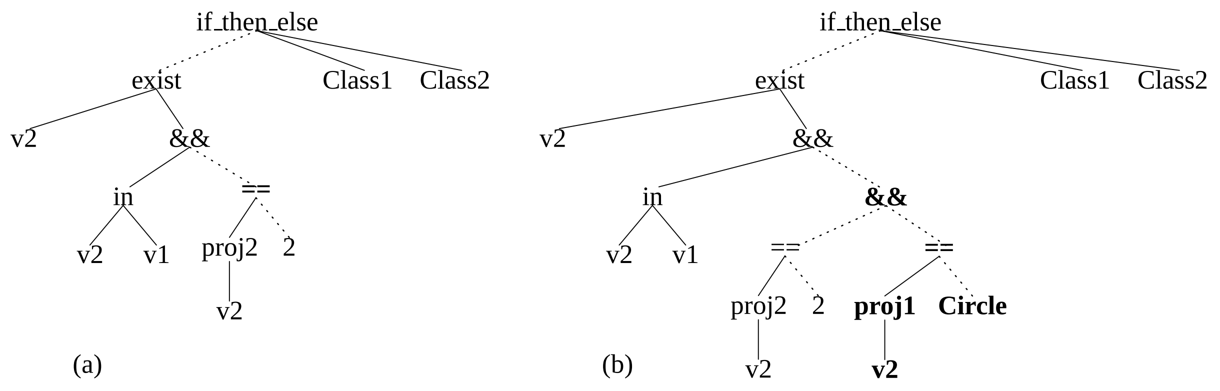


Figure 2: Sample AddConjunction Mutation

programs so that only syntactically correct programs are evolved. In addition to this, it is necessary to preserve the structure of the selector function subtrees. This results in a situation where crossover can only be applied to certain nodes within a program tree. These crossover points correspond to the roots of the subtrees in the function set. Once a crossover point has been randomly selected from the first parent, a crossover point that will maintain type and variable consistency can be randomly selected in the second parent. If no such crossover point is available in the second parent then an alternative crossover point is selected from the first parent and the process is repeated.

2.2.3 Specialised Mutation Operators

The constraints on the evolutionary process that are neccessary to ensure that only valid Escher programs are evolved can result in a decrease in the diversity of genetic material over successive iterations. In an extreme case, this can lead to the loss of genetic material that is essential to the search for an optimal solution and a method for reintroducing such lost material is required. STEPS ensures the preservation of genetic diversity through six distinct forms of mutation. These mutation operators are the terminal and functional mutation operators of conventional GP and four specialisations of functional mutation. The various functional mutations can only be applied at the crossover points in a program tree and must preserve type and variable consistency. The specialised functional mutations include AddConjunction, DropConjunction, AddDisjunction and DropDisjunction. AddConjunction and AddDisjuction involve inserting an && or `||` respectively at the node to be mutated. Its first argument is the subtree originally rooted at that node and its second argument is randomly grown. For example, if we apply the AddConjunction operator to the `==` node in the tree of Figure 2(a), then we could obtain the tree of Figure 2(b).

The DropConjunction and DropDisjunction operators involve randomly selecting an && or `||` crosspoint respectively, replacing it with the subtree that makes up its first argument.

2.2.4 Specialised Crossover Operators

In addition to the specialised mutations, some specialised crossover operators are available during evolution. The first operator AndCrossover involves randomly selecting a crossover node with a return type `Boolean` in the first parent and a crossover node that preserves type and variable consistency in the second parent. The subtrees rooted at the crossover points in both parents are combined as the arguments to an && node and this new subtree is used to replace the subtree selected from the first parent. The second operator OrCrossover works in the same way, except the subtrees selected from both parents are combined as the arguments to an `||` node.

3 DEPTH RESTRICTIONS

There are three main types of restrictions for preventing the unbounded growth of individuals in the evolving population. These include aborting offspring if they are greater than a specified depth or size, editing the extra non-contributing code in the individuals, and penalising large individuals through the fitness function.

3.1 MAX-DEPTH/SIZE CUT OFF

This is the most common approach to restricting the growth of trees during evolution in GP. It works by putting a limit on the depth (the longest path from the root of a tree to any of its leaf nodes) or size (the number of nodes) of an individual. Once an offspring has been obtained, its depth (or size) is measured. If its depth (or size) exceeds the maximum allowed depth (or maximum allowed size) then it is considered to be illegal and is not placed into the next generation. Instead a copy of its parent is placed into the population or the genetic operator is re-applied until a legal offspring is produced.

3.2 EDITING

Another method for restricting the size of the variable length individuals is to remove or edit from the individual the extra pieces of code that are not contributing to its fitness. These extra pieces of code can be calculated a priori from properties of the function set (Soule *et al.* 1996). Deleting Crossover is a variation of this method (Blickle 1996). It involves marking the parts of the code traversed during evaluation and removing the unmarked parts of the code as they are redundant and do not contribute to the individual's fitness.

3.3 PARSIMONY PRESSURE

The idea behind parsimony pressure is to penalise large programs. A penalty proportional to the size of an individual is incorporated into the fitness function. Therefore larger trees will have a lower fitness value providing a bias towards smaller solutions.

3.4 DISCUSSION

When the cut-off depth control method is used in conjunction with crossover as the main genetic operator, it can lead to a loss of diversity of genetic material which can cause the population to converge on a suboptimal solution (Gathercole and Ross 1996). In addition to this, it is difficult to identify which value to set the cut-off limit at. If it is too big then you will be wasting memory and CPU time - too small and the solution will never be found. It is an unnatural and harsh way to control depth and is not in keeping with the theme of natural evolution.

Editing explicitly removes the introns that are protecting the good pieces of code from the destructive effects of crossover and other genetic operators. The removal of the introns is computationally expensive and removes the protection exposing the good code and thus making it vulnerable to destruction. In addition to this, these ineffective blocks of code can be modified during the evolutionary process into useful pieces of code.

Parsimony pressure favours smaller solutions but selecting the correct pressure bias to apply is difficult to determine. In addition to this it is not always possible to use parsimony pressure without an explicit depth restriction such as maximum depth cut off (Gathercole 1998).

4 THE STEPS DEPTH CONTROLLING STRATEGY

4.1 THE STEPS APPROACH

During evolution, as an alternative to picking a genetic operator according to a particular distribution, STEPS allows the choice of the genetic operator to be based on the genetic material of the individual that is randomly selected from the population. The depth controlling strategy works by allowing a randomly selected program tree to select its own genetic operator according to its depth. If the depth of the program tree is greater than the specified maximum depth, then the tree is considered to be too big so a mutation operator that is likely to reduce the size of the tree (i.e., by dropping a disjunction or a conjunction) is chosen to modify the tree. If the depth of the selected tree is less than the minimum specified depth, then the tree is considered to be too small so a mutation operator that is likely to increase the size of the tree (i.e., by adding a disjunction or a conjunction) is chosen to modify the tree. If the depth of the program tree lies within the specified depth constraints then any genetic operator can be randomly selected to modify it.

This strategy is a depth controlling strategy used to keep the size of the program trees under control rather than allowing the the trees to grow in an unconstrained manner. This provides a more flexible method for controlling the depth of the tree. If a tree is considered too big it is not thrown away, but its size is reduced giving any good genetic material that it may contain a chance to survive.

4.2 EXPERIMENTS

In order to evaluate the STEPS depth controlling strategy (Depth) its performance on a number of simple problems is compared to that of GP (i.e., crossover as the sole genetic operator) and the STEPS' basic learning strategy (Basic - where any of the specialised genetic operators can be randomly selected by each individual). Both the GP and Basic approaches use max depth cut off in order to prevent the uncontrollable growth of the program trees. The Basic and Depth approaches both apply mutation operators to newly created offspring to ensure that all program trees in a population are unique.

During the experiments the use of solution structure knowledge such as minimal required alphabet for optimum solution was avoided. This leads to a sub-optimal performance of the algorithm, but gives more realistic conditions as for real world problems where such information is not available. Therefore for each problem the alphabet consists of the selector function subtrees, all available connectives (conjunction, disjunction and negation) regardless of whether they were necessary for the solution, and the problem dependent constants.

For each problem, each learning approach was carried out with a large maximum depth parameter and then with a small maximum depth parameter. The large maximum depth parameter was set to 15 - sufficiently big enough to find the known optimal solution for each problem. The small maximum depth parameter's value varied with each problem. It was set to be small enough so that it was smaller than the depth of the known optimal solution but large enough so that complete program trees could be generated. The experiments were carried out 30 times for each of the three learning approaches. For each run, Tournament selection was used to select individuals from a population of size 300. Predictive accuracy was used as the fitness evaluation and for each problem the optimal solution was defined as a program tree with an error of 0.0.

The next section gives a description of each problem with their associated results followed by a discussion of the results.

4.2.1 Playing Tennis

The objective of the Tennis problem is to generate a concept description that distinguishes weather conditions that allow you to play tennis from weather conditions that do not (Mitchell 1997). For this problem there are fourteen training examples.

The experiments were first carried out with the maximum depth parameter set to 15 and then repeated with the maximum depth parameter set to 5. The results are expressed as the average number of generations taken to find an optimal solution, and the average size of the solution, in Table 1 for the maximum depth of 15 and Table 2 for the maximum depth set to 5. The number in brackets indicates the percentage of runs in which an optimal solution was found within the maximum number of generations (set to 60). A '-' indicates that an optimal solution was not found in any of the runs carried out.

Table 1: Average No. of Generations for Tennis, Max-Depth = 15

STRATEGY	**AV. GENS**	**AV. SIZE**
GP	16.69 (87%)	30.19
Basic	11.17 (100%)	31.067
Depth	11.40 (100%)	30.17

Table 2: Average No. of Generations for Tennis, Max-Depth = 5

STRATEGY	**AV. GENS**	**AV. SIZE**
GP	- (0%)	-
Basic	- (0%)	-
Depth	39.41 (57%)	27.7

4.2.2 Michalski's Train

The objective of the Michalski's train problem is to generate a concept description that distinguishes trains that are travelling East from trains that are travelling West (Muggleton and Page 1994). For this problem there are ten training examples.

The experiments were carried out first with the maximum depth parameter set to 15 and then repeated with the maximum depth parameter set to 6. The results are expressed in Table 3 for the maximum depth set to 15 and in Table 4 for the maximum depth set to 6.

Table 3: Average No. of Generations for Trains, Max-Depth = 15

STRATEGY	**AV. GENS**	**AV. SIZE**
GP	8 (100%)	34.57
Basic	5.6 (100%)	35.37
Depth	6.8 (100%)	35.67

4.3 Animal Class

The objective of the animals problem is to generate a concept description that distinguishes between four classes of animals. For this problem there are sixteen training examples.

The experiments were first carried out with the maximum depth parameter set to 15 and then with the maximum depth parameter set to 5. The results are expressed in Table 5 for the maxmum depth set to 15 , and in Table 6 for the maximum depth set to 5.

Table 4: Average No. of Generations for Trains, Max-Depth = 6

STRATEGY	AV. GENS	AV. SIZE
GP	- (0%)	-
Basic	21.5 (20%)	29
Depth	12.8 (100%)	32.8

Table 5: Average No. of Generations for Animals, Max-Depth = 15

STRATEGY	AV. GENS	AV. SIZE
GP	7.167 (100%)	27.8
Basic	9.133 (100%)	30.433
Depth	10.1 (100%)	34.167

4.4 DISCUSSION OF RESULTS

Results show that for a large maximum depth there is no real difference between the performance of the Depth and Basic approaches. Both perform slightly better than GP on the Tennis and the Michalski's train problem. However for the Animals problem, it is the GP approach that performs slightly better than the other two approaches, finding a solution on average in fewer generations.

But on runs with a small maximum depth the Basic and GP approaches never find a solution for the Tennis problem, whereas the STEPS' Depth control approach finds a solution in over half the runs for the Tennis problem and in every run for the Animals and the Michalski's train problems. The Basic approach is able to find a solution in 6 out of 30 of its runs for the Michalski's train problem with the maximum depth threshold set to 6, and 4 out of 30 of its runs for the animals problem with the maximum depth threshold set to 5. This is due to the application of the mutation operators to newly created offspring. The GP approach never finds the solution if the depth bound is set too small.

The basic idea behind the STEPS depth controlling strategy is to grow or shrink trees to fit the problem. Consequently, no real advantage is gained by using this strategy when a large maximum depth is specified. On the other hand, such large bounds often lead to inefficiency. The STEPS depth controlling strategy allows experienced users to retain efficiency without jeopardising the chances of finding a (larger) optimal solution by specifying conservative depth restrictions. The system will compute efficiently within these restrictions and only incur additional computational costs if these are strictly necessary to produce a better solution. In addition, the system also becomes more robust since the uninformed guesses of inexperienced users do not hinder its capacity to find an optimal solution.

Table 6: Average No. of Generations for Animals, Max-Depth = 5

STRATEGY	AV. GENS	AV. SIZE
GP	- (0%)	-
Basic	27.125 (27%)	20.75
Depth	10.166 (100%)	30.55

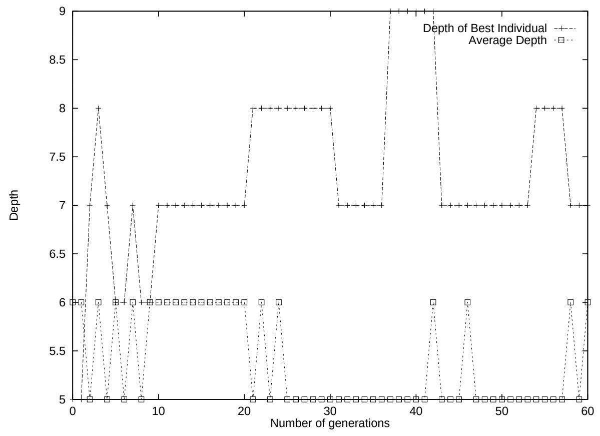


Figure 3: A graph of the depth statistics from a run using the STEPS' depth monitoring strategy with maximum depth set to 5

5 CONCLUSION

This paper details the dynamic depth monitoring strategy of STEPS. The strategy allows the depth of individuals to grow as necessary while keeping the average depth of the population under control. Figure 3 illustrates that while the average depth of the program trees are kept to (approximately) within the depth constraints, the depth of the fittest trees are allowed to grow beyond this threshold as necessary. The preliminary experiments demonstrate that the depth approach performs as well as the other two approaches when a more than sufficient maximum depth is set, and outperforms the other approaches when a small maximum depth threshold is set. Further experimentation is necessary to determine the full implications of the approach. In particular experiments will be carried out on some more substantial, real world problems.

Acknowledgements

This work is funded by EPSRC grant GR/L21884. The authors would like to thank Tony Bowers for his implementation of the Escher interpreter.

References

P. J. Angeline (1994). Genetic programming and emergent intelligence. In: K. E. Kinnear Jr. (ed.) *Advances in Genetic Programming*, Chapter 4, 75-89. MIT Press.

P. J. Angeline (1996). Two self-adaptive crossover operators for genetic programming. In: P. J. Angeline and K. E. Kinnear Jr. (eds.) *Advances in Genetic Programming 2,* Chapter 5, 89-110. Cambridge, MA: MIT Press.

A. F. Bowers, C. Giraud-Carrier and J. W. Lloyd (1999). Higher-order logic for knowledge representation in inductive learning. In preparation.

T. Blickle (1996). Theory of Evolutionary Algorithms and Application to System Synthesis. PhD thesis. Swiss Federal Insistute of Technology, Zurich.

T. Blickle (1996). Evolving compact solutions in genetic programming: A case Study. In: H. Voigt, W. Ebeling, I Rechenberg, and H. Schwefel (eds.), *Parallel Probelm Solving from Nature IV . Proceedings of the International conference on Evolutionary Computation*, Vol. 1141 of *LNCS*, 564-573. Berlin, Germany: Springer Verlag.

T. Blickle and L. Thiele (1994). Genetic programming and redundancy. In J. Hopf (ed.) *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, 33-38. Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany: Max-Planck-Institut für Informatik (MPI-I-94-241)

P. Flach, C. Giraud-Carrier, and J. W. Lloyd (1998). Strongly typed inductive concept learning. In *Proceedings of the International Conference on Inductive Logic Programming (ILP'98)*, 185-194.

C. Gathercole (1998). An Investigation of Supervised Learning in Genetic Programming. PhD thesis. University of Edinburgh.

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

C. J. Kennedy and C. Giraud-Carrier (1999). An evolutionary approach to concept learning with structured data. In: A. Dobnikar (ed.) *Proceedings of the Fourth International Conference on Artificial Neural Networks and Genetic Algorithms* To appear.

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

W. B. Langdon and R. Poli (1998). Fitness causes bloat. In: P. K. Chawdhry, R. Roy and R. K. Pant (eds.), *Soft Computing in Engineering, Design and Management*, 13-22. Springer.

J. W. Lloyd (1995). Declarative programming in escher. Technical Report CSTR-95-013. Department of Computer Science, University of Bristol.

T. M. Mitchell (1982). Generalisation as search. *Artificial Intelligence* **18**:203-226.

T. M. Mitchell (1997). *Machine Learning*. McGraw-Hill

D. J. Montana (1995). Strongly typed genetic programming. *Evolutionary Computation* **3**(2):199-230.

S. Muggleton and C.D. Page (1994). Beyond first-order learning: Inductive learning with higher-order logic. Technical Report PRG-TR-13-94. Oxford University Computing Laboratory.

J. P. Rosca and D. Ballard (1996). Complexity drift in evolutionary computation with tree representations. Technical Report NRL5. University of Rochester, Computer Science Depatment, Rochester, NY, USA.

T. Soule, J. A. Foster and J. Dickinson (1996). Code growth in genetic programming. In: J. R. Koza, D. E. Goldberg, D. B. Fogel and R. L. Riolo (eds.), *Genetic Programming 1996: Proceedings of the First Annual Conference* 291-296. Stanford University, CA: MIT Press.

W .A. Tackett (1994). Recombination, Selection, and the Genetic Construction of Computer Programs. PhD thesis. Department of Electrical Engineering Systems, University of Southern California.