Type Inheritance in Strongly Typed Genetic Programming

Thomas D. Haynes, Dale A. Schoenefeld and Roger L. Wainwright

December 22, 1996

This paper appears as chapter 18 of Kenneth E. Kinnear, Jr. and Peter J. Angeline, editors Advances in Genetic Programming 2, MIT Press, 1996.

Abstract

Genetic Programming (GP) is an automatic method for generating computer programs, which are stored as data structures and manipulated to evolve better programs. An extension restricting the search space is Strongly Typed Genetic Programming (STGP), which has, as a basic premise, the removal of closure by typing both the arguments and return values of functions, and by also typing the terminal set. A restriction of STGP is that there are only two levels of typing. We extend STGP by allowing a type hierarchy, which allows more than two levels of typing.

1 Introduction

> A program written in a language that does not support natural ways to express the required algorithms will necessarily use unnatural methods. Unnatural methods are less likely to be correct for all possible situations.
> [Sebesta 1993] (p. 17)

Strongly Typed Genetic Programming (STGP) [Montana 1995] is used to restrict the search space considered in the Genetic Programming (GP) paradigm [Koza 1992]. This is shown in both the paper in which Montana introduced STGP [Montana 1995] and in our research into multiagent behavioral strategies [Haynes et al. 1995]. STGP is able to reduce the search space by only allowing syntactically correct programs to be generated and produced by the crossover and mutation operators. Montana types both the function return value and the arguments, and requires that the typing restrictions be honored by all operations on the S–expressions. In order to allow for a minimal function set, generic types are introduced. Generic types must be instantiated during node construction.

In effect, generic types allow for a two level type hierarchy, as shown in Figure 1. From object oriented programming [Sebesta 1993], we know that it can be desirable to have more than two levels in a hierarchy. A simple example involving cars and numbers illustrates this desire. If we consider the class hierarchy shown in Figure 2, and assume the standard arithmetic operators of **addition**, **subtraction**, **division**, and **multiplication**, then we do not want to have specialized versions of these operators for `Reals` and `Integers`. The standard typing solution is to have a generic function for each of the operators, which can handle any type.

However, to type **addition** as `Generic`, we would have to ensure that **addition** is overloaded such that it makes sense in all contexts which can be instantiated from `Generic`. Failure to do so leads to the undesirable result that the program in Figure 3 is valid. What does it mean to add two `Fords`? Are we simply counting the cars, by type, that pass us on the highway? Or are we trying to add the qualities of one car to another to get a hybrid?

What we would like to do is to define the operator **addition** in class `Numbers`, appropriately overload it in class `Reals` and class `Integers`, and force the program in Figure 3 to be invalid. This further restricts the allowed inputs while still reducing the total number of functions. We are extending the concept of a generic type for the tree to generic types for subtrees.

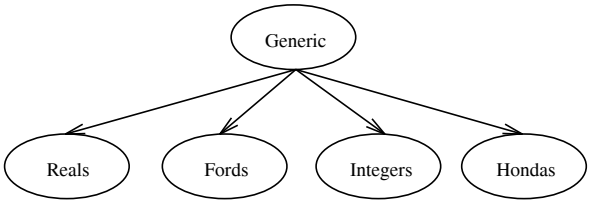

Figure 1: An example STGP type hierarchy.

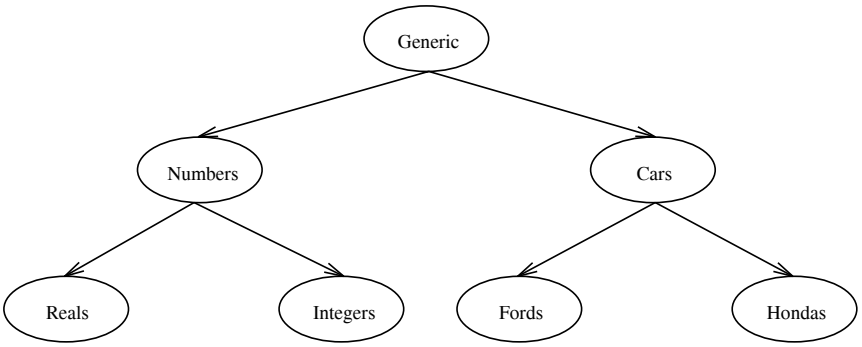

Figure 2: Example of a simple class/type hierarchy.

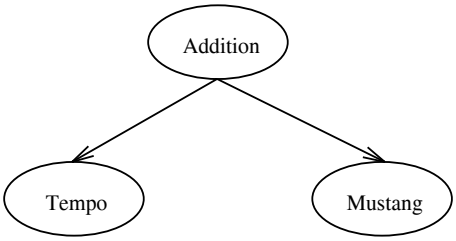

Figure 3: Addition of two `Fords`, which is not allowed.

A generic type can be thought as a variable for types and can be instantiated with any of the other allowable types. This property requires that the `Generic` type be the root node of the type tree, as is shown in both Figure 1 and Figure 2. The extension we present in this chapter is to allow multiple levels of generic types, i.e. the `Numbers` and `Cars` nodes in Figure 2. We derived the term *type hierarchy* from the fact that in the tree form of the class hierarchy, as shown in Figure 2, the generic types are inherited just like the other class properties. Thus class hierarchies illustrate the inheritance of type.

In this chapter we introduce such a mechanism to STGP, and we show the results of applying it to the clique detection problem. The rest of this chapter is organized as follows: Section 2 presents the basic concepts of STGP. Section 3 presents our modification to STGP to allow for type hierarchies. Section 4 describes the clique detection problem. Section 5 details our implementation of a STGP to solve the clique detection problem. Section 6 provides the results of using our new technique to solve the clique detection problem. Section 7 concludes our discussion on the type hierarchy. Section 8 discusses how this research can be extended.

2 Strongly Typed Genetic Programming

In GP the user must specify all of the functions, variables and constants that can be used as nodes in a parse tree or S–expression. Functions, variables and constants which require no arguments become the leaves of the parse trees and are called *terminals*. Functions which require arguments form the branches of the parse trees, and are called *non–terminals*. The set of all terminals is called the *terminal set*, and the set of all non–terminals is called the *non–terminal set*. Note the term *non–terminal* is what Koza [Koza 1992] calls a *function*. We follow this notation.

One serious constraint on the user–defined terminals and functions is *closure*, which means that all of the functions must accept arguments of a single data type and return values of the same data type. This means that all functions return values that can be used as arguments for any other function. Hence, closure means any element can be a child node in a parse tree for any other element without having conflicting data types. An example of closure is given in the prefix expression **div** *Ralph 0*. Not only must the division operator handle division by *0*, it must also convert *Ralph* into a numeric value.

In essence, standard GP has a "flat" type space of only one level. Montana [Montana 1995] claims that closure is a serious limitation to genetic programming. Koza [Koza 1992] describes a way to relax the closure constraint using the concept of *constrained syntax structures*. Koza used tree generation routines which only generated legal trees. He also only used operations on the parse trees which maintained legal syntactic structures.

Maintaining legal syntactic structures is at the heart of STGP. In STGP, variables, constants, arguments, and returned values can be of any type. The only restriction is that the data type for each element be specified beforehand. This causes the initialization process and the various genetic operations to only construct syntactically correct trees. A benefit of syntactically correct trees is that the search space is reduced. This has been shown in [Montana 1995, Haynes et al. 1995] to decrease the search time. One key concept of STGP is the idea of generic functions, which is a mechanism for defining a class of functions, and defining generic data types for these functions. Generic functions eliminate the need to specify multiple functions which perform the same operation on different types.

A classical example of a generic function is the **IFTE** function, which evaluates and returns its second argument if the first argument evaluates to true, otherwise it evaluates and returns the third argument. It can be typed as

$$\texttt{Generic IFTE}(\texttt{Boolean } A,\ \texttt{Generic } B,\ \texttt{Generic } C),$$

which allows **IFTE** to be reused by any type. Note that once B is instantiated, then both C and the return of **IFTE** must be instantiated to the same type.

Figure 4 illustrates the instantiation of the **IFTE** operator. The **IFTE** at the root of the tree has a return type of `Ford`, and the second **IFTE** has a return type of `Boolean`. Notice that each of the respective return types is typed the same as the respective B and C arguments.

The STGP algorithm differs from GP in that typed–based restrictions must be enforced at three points: initial tree generation, mutation, and crossover. As trees are generated, only child nodes with a return type

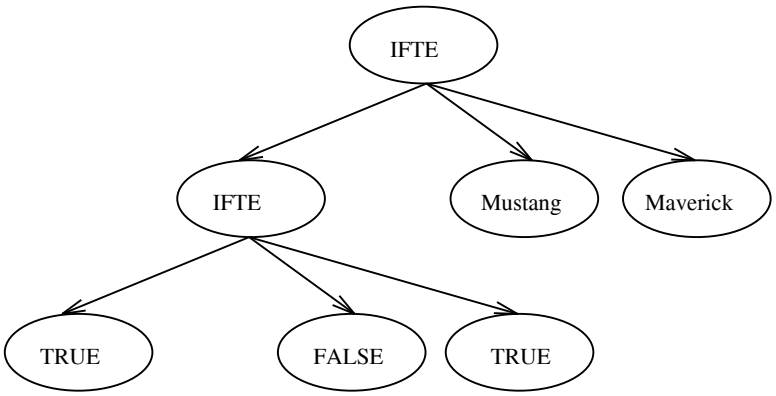

Figure 4: Instantiation of **IFTE**.

matching the argument type of the parent node can be instantiated into the tree. Also, even if a child node is type compatible with an argument, there has to be a check to make sure that the subtree represented by the child node does not violate the maximum depth restriction. A type table, showing valid types per depth level, is generated *a priori* to provide quick lookup to determine if this restriction has been violated. The expected type for the root node is an input parameter and domain dependent. It is during tree generation that generic functions are instantiated. Note that the standard mutation operator is a specialized case of tree generation.

During the crossover process, the return value type of the two subtrees selected for exchange must be tested to see if they are of the same type and if the resulting trees violate depth restrictions. If either check fails, then two new subtrees are selected. If, after a finite number of selections, no valid crossover points are found, then the two parent trees are copied into the pool for the next generation. Note that at this point there can not be any generic types in the S–expressions, as they must have been instantiated.

3 Modifying STGP

Abstract classes do not contain any instantiated objects in the class. An example of this is the `Generic` class of STGP. In STGP, only one abstract class is allowed. We propose to allow multiple abstract classes. A concrete class is one in which a class can have objects instantiated in the class. Note that all concrete classes can be made abstract by simply adding another class under the concrete class, and putting all objects that were originally in the concrete class into this new subclass.

The distinction can be illustrated by considering the `Cars` type in Figure 2:

1. Do we allow an instantiation of `Cars` to be the set of all automobiles minus the sets of `Fords` and `Hondas`?

2. Or is `Cars` the set of all automobiles including `Fords` and `Hondas`?

3. Or are there no other automobiles other than `Fords` and `Hondas`?

The last item reflects an abstract base class, and the first two are cases of concrete base classes.

An STGP algorithm which is modified for type inheritance has to perform additional checks during tree generation, crossover and mutation. During tree generation, base classes (types), both abstract and concrete, will also have to be instantiated. Abstract base classes do not have any objects that can be instantiated themselves, instead they serve as placeholders for collections of objects.

We construct a type tree to facilitate this checking. An example type tree is shown in Figure 2. Using the subtype principle, wherever a type may appear a descendant of it in the type tree may also appear. Following the example of generic functions, once an argument is instantiated to a specific type, then the remaining arguments are also typed to that instantiation. Thus an additional check must be performed to determine if a subtype is allowed.

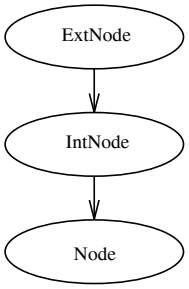

Figure 5: Non–branching type hierarchy for clique detection.

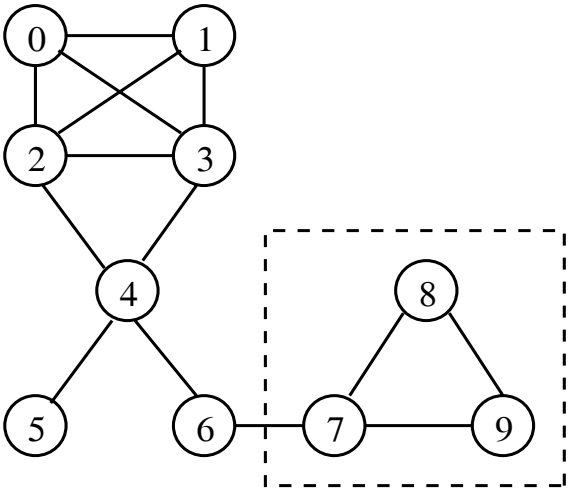

Figure 6: The 10 node graph for Example II. Note the candidate clique $\{7, 8, 9\}$ has been boxed off.

Type trees may be considered by two cases: *non–branching* and *branching*. In a non–branching type tree the tree has a branching factor of 1. This gives a linear type tree, as shown in Figure 5. A branching type tree has at least one node with a branching factor ≥ 2.

A non–branching type tree, as in Figure 5, is relatively simple to implement. Difficulties arise when we consider a branching type tree, as in Figure 2. In particular, assume we have a type tree with a type `T` which has left and right subtypes `STLC` and `STRC` respectively. If we have a function `T` $\mathcal{F}$ (`T` , `T`), and the first argument is typed as `STLC`, can the second be instantiated to type `STRC`? Using the subtype principle, this is permissible. But it is possible that `STLC` and `STRC` have some distinguishing characteristics that do not permit all relationships between them to be expressed.

4 Clique Detection

We have modified our STGP system, GPengine, to investigate research into type inheritance [Haynes 1995]. Specifically we consider the problem of finding all the cliques, i.e. the maximal subgraphs of a graph having an edge joining each pair of its vertices [Kalmanson 1986], in an undirected or directed graph. Figure 6 is a 10 node graph, with cliques:

$$C = \{\{0, 1, 2, 3\}, \{2, 3, 4\}, \{4, 5\}, \{4, 6\}, \{6, 7\}, \{7, 8, 9\}\}.$$

Finding all of the cliques in a graph is "more difficult" than finding the maximum clique of a graph[1]. To find the maximum clique, one can start at size $n = 2$, and determine if there exists a clique of size n. If

[1] We say "more difficult" because even though the maximum clique detection is NP complete, there is more work to be done to determine all of the cliques.

there is, then n is incremented by one, and the process is repeated. There is no need to "remember" previous cliques found, unless one takes the approach of first checking the corresponding clique of size $n-1$ to see if it can be incremented by one. The difficulty of finding all the cliques arises in having to remember all of the candidate cliques. In the maximum clique algorithm we are not concerned with identity, but, rather, with existence. When finding the maximum clique, we ask the question: is there a clique of size n? When finding all the cliques we ask: is this clique of size n the same as another one of size n?

Finding the maximum clique of a graph has been addressed within the genetic algorithm (GA) community [Bui and Eppley 1995, Chandraasekharam et al. 1993, Murthy et al. 1995]. The problem of finding the set of all cliques has not been addressed within the GA community. In part this is due to the fact that a variable length structure is needed to find all the cliques, while a fixed length chromosome can be used to find the maximum clique. Our preliminary work on the clique detection problem can be found in [Haynes 1995].

A variable length chromosome is necessary because, in general, there will be an unknown number of cliques per graph. Potential cliques, denoted as *candidate cliques*, need to be evaluated to determine if they actually are cliques. The chromosome is then interpreted as a collection of candidate cliques.

Candidate cliques need to be tested to determine if they

- contain duplicate nodes.
- are a duplicate of another candidate clique.
- are subsumed by another candidate clique.
- are completely connected.
- are maximal.

4.1 Representation Scheme

Each S–expression in a GP pool will represent sets of candidate maximal cliques. The function and terminal sets are respectively $F = \{ExtConnection,\ IntConnection\}$ and $T = \{1 \ldots \#nodes\}$. *ExtConnection* "separates" two candidate maximal cliques, while *IntConnection* "joins" two candidate cliques to create a larger candidate clique. Graphs are encoded in the DIMACS Challenge file format.[2]

4.2 Fitness Measure

The fitness evaluation is composed of two parts: a reward for clique size and a reward for the number of cliques in the tree. Since we want to gather the maximal complete subgraphs, we want the reward for size to be greater than the reward for the number of cliques. We also want to make sure that we do not reward for a clique either being in the tree twice or being subsumed by another clique. The first will falsely inflate the fitness of the individual, while the second will invalidate the goals of the problem.

The algorithm for the fitness evaluation is:

1. Parse the S–expression into a sequence of candidate maximal cliques, each represented by an ordered list of vertex labels.
2. Throw away any duplicate candidate maximal cliques and any candidate maximal cliques that are subsumed by other candidate maximal cliques.
3. Throw away any candidate maximal cliques that are not cliques.

The formula for measuring the fitness is:

$$F = \alpha c + \sum_{i=1}^{c} \beta^{n_i},$$

[2] ftp://dimacs.rutgers.edu/pub/challenge/graph/doc/ccformat.dvi

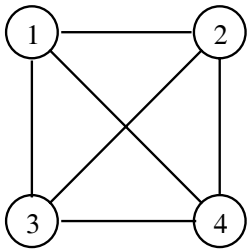

Figure 7: Example four node graph.

Table 1: Naive attempt at typing for clique detection.

Function/Terminal	Return Type	Argument Structure
Nodes	Node	
ExtConnection	Node	Node A, Node B
IntConnection	Node	Node A, Node B

where $c = \#\ of\ valid\ candidate\ maximal\ cliques$ and $n_i = \#\ nodes\ in\ clique_i$. Both α and β are configurable by the user. Note that β must be large enough so that a large clique contributes more to the fitness of one S–expression than a collection of proper subcliques contributes to the fitness of a different S–expression. For example, consider the graph in Figure 7. It is clear that there is only one maximal clique: $C_4 = \{1, 2, 3, 4\}$. However, there are four subcliques of cardinality three: $C_3 = \{\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}\}$.

5 Implementation

The S–expression is mapped into a list of list of nodes. An S–expression is shown in Figure 8, and corresponds to the set

$$C = \{\{1, 2\}, \{2, 1, 1\}, \{3, 1, 2\}\}.$$

Let L_e be a list of candidate cliques, and L_c be a candidate clique, i.e. a list of nodes. It is evident that each element of L_e is an L_c. We want to ensure that the members of L_e and L_c are not L_e, i.e. neither a list of candidate cliques nor a candidate clique can have as a member a list of candidate cliques. This is shown in Figure 9, where an *ExtConnection* is a child node of an *IntConnection* node.

If the typing is the same as in Table 1, then S–expressions of the form shown in Figure 9 can be generated. These S–expressions correspond to incorrect representations of the solution space. We do not want sets of cliques in which each clique can be a set of cliques.

From examining the desired form of the result, (see Figure 8) we derive the type system shown in Table 2. This system can not be implemented in a standard STGP package. It is representative of the non–branching type inheritance discussed in Section 3.

Note that the two type levels represented in Table 2 are sufficient to solve the clique problem. However the types can be confusing to someone trying to understand the problem. Both **IntConnection** and *Nodes* have a return type of `Node`. **ExtConnection** is a list building operator, **IntConnection** effectively performs a Union on its child nodes, and *Nodes* returns a singleton set. Since **IntConnection** is returning nodes, and *Nodes* is returning a node, the type of `Node` suggests they are returning the same object. Also, an **ExtConnection** can have the terminal *Nodes* as a child node, with no interconnecting **IntConnection**. This is the only way to specify a clique with only one member. Thus, `Node` is overloaded. To be correct, we should extend the type hierarchy to three levels, as shown in Figure 5.

Table 2: Successful attempt at typing for clique detection.

Function/Terminal	Return Type	Argument Structure
Nodes	Node	
ExtConnection	ExtNode	ExtNode A, ExtNode B
IntConnection	Node	Node A, Node B

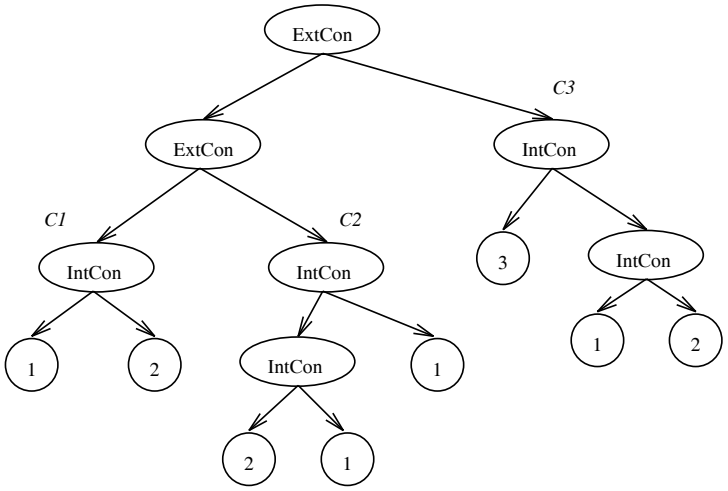

Figure 8: Example S–expression for the clique detection. Candidate cliques are labeled C1, C2, and C3.

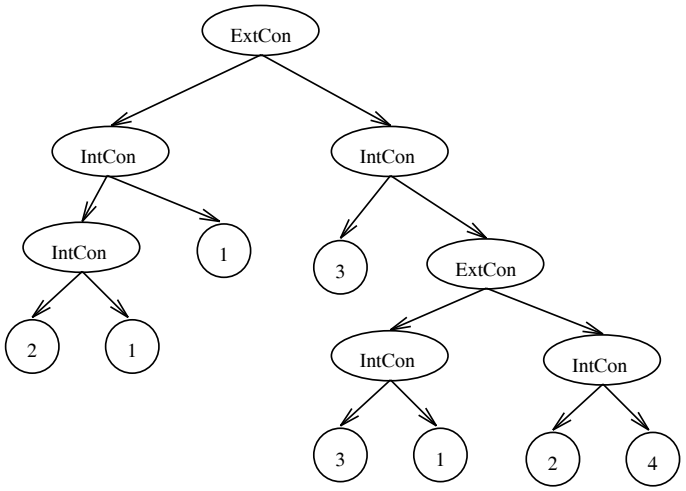

Figure 9: Bad S–expression corresponding to the first attempt at a type system.

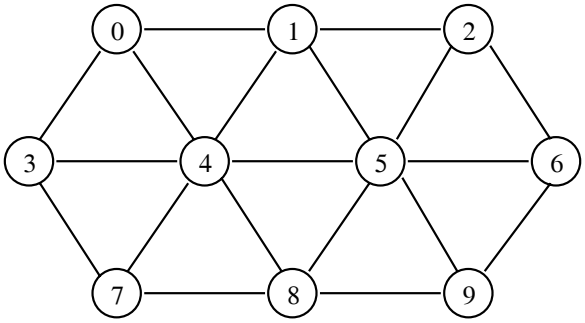

Figure 10: The 10 node graph for Example I.

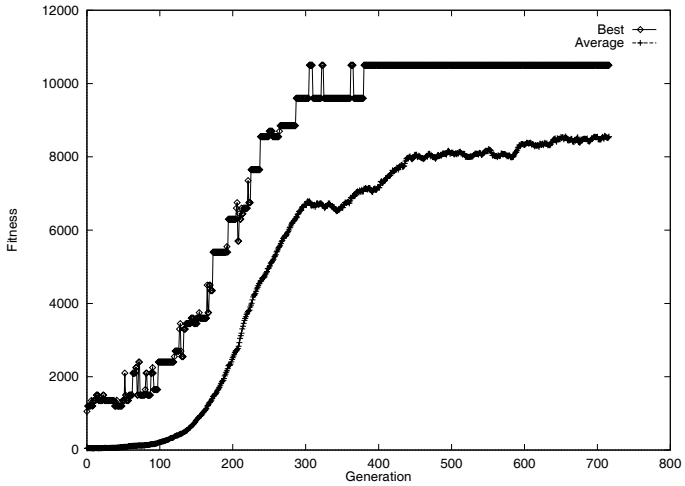

Figure 11: Best and Average fitnesses for Example I.

6 Experimental Results

The clique detection problem was implemented in a version of our STGP system, which has been modified to allow for a non–branching type tree as discussed in Section 3. Specifically, the problem of **ExtConnection** functions requiring either a parent node which is an **ExtConnection** function or be the root, was addressed.

In the next subsections, we will present some graphs to illustrate properties of our maximal clique detector. In all cases we set the population size to be 2000. We have also performed experiments with sample test cases from the DIMACS repository. For the *johnson16–2–4.clq* graph, with 120 nodes and a largest clique of size 8, we were able to detect cliques of size 7. For the *hamming8-2.clq* graph, with 64 nodes and a largest clique of size 4, we were able to detect cliques of size 4. It should be realized that these graphs are intended for a different problem, i.e. finding the largest clique.

6.1 Example Graph I

Figure 10 is a ten node graph that was used to test the clique detection system. The best and average fitness per generation are plotted in Figure 11. The system steadily improves after generation 100, and the global optimum is found around generation 375. By generation 500, it is evident that a plateau has been reached, and that the average fitness is only increasing slightly. S–expressions for the candidate maximal cliques corresponding to generations 0, 100, 200, 300, and 400 can be found in [Haynes 1995]. The resultant maximal cliques for generations 0, 100, 200, 300, and 400 are shown in Table 3. Notice the steady addition of cliques of cardinality three. Clearly by generation 400, all of the maximal cliques have been found.

Table 3: Cliques detected for 10 Node graph, Example I.

Generation	Fitness	# of Snodes	List of Candidate Maximal Cliques
0	1054	9	{{6}, {2}, {4, 7, 8}}
100	2400	481	{{0, 3, 4}, {3, 7}, {5}, {0, 1, 4}, {6}, {8}, {1, 2}}
200	6300	157	{4, 5, 8}, {0, 3, 4}, {1, 2, 5}, {9}, {3, 4, 7}, {4, 7, 8}, {0, 1, 4}}
300	9600	277	{4, 5, 8}, {0, 1, 4}, {1, 2, 5}, {2, 5, 6}, {1, 4, 5}, {0, 3, 4}, {4, 7, 8}, {6, 9}, {3, 4, 7}, {5, 8, 9}}
400	10500	439	{5, 6, 9}, {3, 4, 7}, {1, 2, 5}, {4, 7, 8}, {4, 5, 8}, {0, 3, 4}, {1, 4, 5}, {0, 1, 4}, {2, 5, 6}, {5, 8, 9}}

6.2 Example Graph II

Figure 6 is a ten node graph that was used to test the effect of varying β on clique detection. From the discussion on α and β in Section 4.2 and since $C_{max} = 4$, the proposed optimal value for β is 5. In this section, we examine values of $\beta = 10$ and $\beta = 5$ on the graph in Figure 6. We hold α constant at 4.

The best and average fitness per generation are plotted in Figure 12 for $\beta = 10$. The system is stuck in a local optimum until about generation 500, at which point the average fitness makes a dramatic jump, and then steadily improves. The largest maximal clique is found in generation 0. The resultant candidate maximal cliques for generations 0, 100, 200, 300, 400, 500, and 600 are shown in Table 4.

The best and average fitness per generation is plotted in Figure 13 for $\beta = 5$. The system does not get stuck in a local optimum, but has a steady increase in fitness. The shape of the curve is more like Figure 11 than Figure 12. The step–like increases between generations 100 and 300 indicate the incremental learning of different cliques. The resultant candidate maximal cliques for generations 0, 100, 200, 300, and 400 are shown in Table 5. With $\beta = 5$, it would have been nice if the candidate maximal cliques $C_{7,8,9} = \{\{7,8\},\{7,9\},\{8,9\}\}$, had coalesced into the clique $C = \{7,8,9\}$.

The difference between the two clique sets with $\beta = 10$ and $\beta = 5$ is that the larger value of β rewards for finding larger cliques. With too high a value of β, the clique detection system is actually hampered in the attempt to find all of the cliques in a graph.

7 Conclusion

The STGP as described by Montana [Montana 1995] deals with genericity but not with other aspects of object oriented methods. We propose an extension of STGP to deal with type hierarchies, and in particular, with polymorphism and dynamic binding aspects of the object oriented paradigm. In the two cases we presented, we have successfully implemented the non–branching case. In particular, we have shown how a non–branching type hierarchy can be used in clique detection, and how the standard STGP does not work for clique detection.

8 Future Work

Another problem domain to which this research is applicable is in determining the set of functional dependencies (FD) in a relational database. Given the relational database schema for a relational extension, the goal

Table 4: Cliques detected for 10 Node graph, Example II, with $\beta = 10$.

Generation	Fitness	# of Snodes	List of Candidate Maximal Cliques
0	10050	15	{{ 0, 1, 2, 3 }, { 5 }, { 6 }}
100	2400	353	{{ 1, 2, 3 },{ 0, 2, 3 },{ 4, 6 } { 4, 5 }, { 9 }, { 7 } }
200	10350	125	{{ 3, 4 }, { 8, 9 }, { 0, 1, 2, 3 }, { 5 } }
300	10050	47	{ { 0, 1, 2, 3 } }
400	10500	137	{ { 7, 8 }, { 0, 1, 2, 3 }, { 6 }, { 9 }, { 4, 5 }, { 3, 4 } }
500	11100	709	{{ 6 }, { 0, 1, 2, 3 } { 5 }, { 4 }, { 7, 8, 9 } }
600	12300	359	{{ 7, 8, 9 }, { 2, 3, 4 }, { 0, 1, 2, 3 } { 5 }, { 4, 6 } }

Table 5: Cliques detected for 10 Node graph, Example II, with $\beta = 5$.

Generation	Fitness	# of Snodes	List of Candidate Maximal Cliques
0	675	15	{{ 0, 1, 2, 3 }, { 5 }, { 6 } }
100	750	137	{{ 9 }, { 0, 1, 2, 3 } { 4, 5 }, { 7 } }
200	1125	233	{{ 4, 5 }, { 0, 1, 2, 3 } { 8, 9 }, { 6, 7 }, { 3, 4 } { 2, 4 }, { 7, 8 } }
300	1300	267	{{ 2, 3, 4 }, { 4, 5 } { 8, 9 }, { 7, 9 }, { 4, 6 } { 6, 7 }, { 0, 1, 2, 3 }, { 7, 8 } }
400	1300	267	{{ 4, 6 }, { 7, 9 }, { 6, 7 } { 2, 3, 4 }, { 7, 8 } { 0, 1, 2, 3 }, { 8, 9 }, { 4, 5 } }

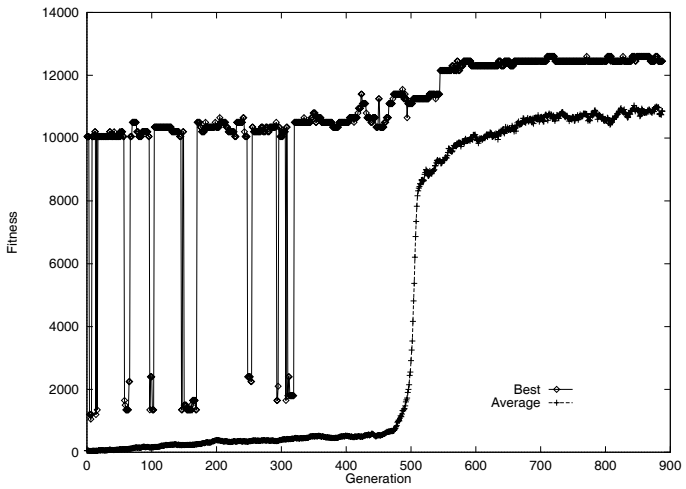

Figure 12: Best and Average fitnesses for Example II, with $\beta = 10$.

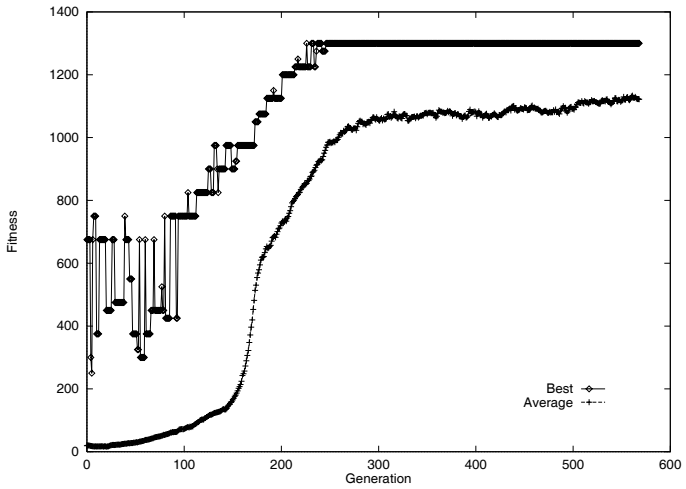

Figure 13: Best and Average fitnesses for Example II, with $\beta = 5$.

is to determine the FDs between the attributes [Elmasri and Navathe 1994, Fayyad and Uthurusamy 1995]. An example is with

$$R = \{A, B, C, D, E\},$$

$AB \rightarrow CDE$, i.e. given that the values for A and B are the same in two tuples, then so are the values for C, D, and E. The set of candidate FDs can be thought of like the set of candidate cliques.

In this chapter we presented an application of the non–branching type hierarchy to a domain. We are also looking for a suitable domain to illustrate the branching case.

9 Notational Conventions

A guide to the different notational conventions:

1. Functions are in boldface, **addition**.
2. Terminals are in italics, *19.27*.
3. Types are in the typewriter family, `Reals`.

10 Acknowledgments

This research was partially supported by OCAST Grant AR2–004 and Sun Microsystems, Inc.

References

[Bui and Eppley 1995] Bui, T. N. and Eppley, P. H. (1995). A hybrid genetic algorithm for the maximum clique problem. In Eshelman, L., editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 478–484, San Francisco, CA. Morgan Kaufmann Publishers, Inc.

[Chandraasekharam et al. 1993] Chandraasekharam, R., Subhramanian, S., and Chaudhury, S. (1993). Genetic algorithm for node partioning problem and applications in VLSI design. *IEE Proceedings, Part E: Computers and Digital Techniques*, 140(5):255–260.

[Elmasri and Navathe 1994] Elmasri, R. and Navathe, S. B. (1994). *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Redwood City, CA.

[Fayyad and Uthurusamy 1995] Fayyad, U. M. and Uthurusamy, R., editors (1995). *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95)*. AAAI Press, Menlo Park, CA.

[Haynes 1995] Haynes, T. (1995). Clique detection via genetic programming. Technical Report UTULSA-MCS-95-02, The University of Tulsa.

[Haynes et al. 1995] Haynes, T., Wainwright, R., Sen, S., and Schoenefeld, D. (1995). Strongly typed genetic programming in evolving cooperation strategies. In Eshelman, L., editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 271–278, San Francisco, CA. Morgan Kaufmann Publishers, Inc.

[Kalmanson 1986] Kalmanson, K. (1986). *An Introduction to Discrete Mathematics and its Applications*. Addison–Wesley.

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

[Montana 1995] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[Murthy et al. 1995] Murthy, A. S., Parthasarthy, G., and Sastry, V. U. K. (1995). Clique finding – a genetic approach. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 18–21, Piscataway, NJ. IEEE.

[Sebesta 1993] Sebesta, R. W. (1993). *Concepts of Programming Languages*. The Benjamin/Cummings Publishing Company.