Comparison of Tree and Graph Encodings as Function of Problem Complexity

Michael Schmidt

Computational Synthesis Laboratory
Cornell University

Ithaca, NY 14853, USA

mds47@cornell.edu

Hod Lipson

Computational Synthesis Laboratory
Cornell University
Ithaca, NY 14853, USA

hod.lipson@cornell.edu

ABSTRACT
In this paper, we analyze two general-purpose encoding types, trees and graphs systematically, focusing on trends over increasingly complex problems. Tree and graph encodings are similar in application but offer distinct advantages and disadvantages in genetic programming. We describe two implementations and discuss their evolvability. We then compare performance using symbolic regression on hundreds of random nonlinear target functions of both 1-dimensional and 8-dimensional cases. Results show the graph encoding has less bias for bloating solutions but is slower to converge and deleterious crossovers are more frequent. The graph encoding however is found to have computational benefits, suggesting it to be an advantageous trade-off between regression performance and computational effort.

Categories and Subject Descriptors
I.1.1 [**Symbolic and Algebraic Manipulation**]: Expressions and their representations – *representations (general and polynomial), simplifications of expressions*

General Terms
Algorithms, Performance, Design

Keywords
Symbolic Regression, Expression Trees, Expression Graphs

1. INTRODUCTION
In this paper, we analyze the differences between a tree and graph encoding in genetic programming. The choice of solution encoding in genetic programming can have dramatic impacts on the evolvability, convergence, and overall success of the algorithm [7]. Algorithms and encodings are often described by their bias-variance trade-off – error introduced by predisposed structure (bias), and error introduced by representative power and accommodation (variance) [9,10,11,12]. In this paper, we examine such trade-offs more precisely, considering their

representations, solution bloat, overfitting, and convergence over a range of complexity problems. In contrast with previous research, we examine these performance trends across problems with a systematically-generated range of complexities.

Tree encodings are well-known for their representative power and used heavily in genetic programming [1]. Tree encodings are generally rooted with each branch describing a unique or isolated sub-structure. In contrast, graph (or network) encodings describe groups of interacting or re-used structures.

Graph encodings allow direct re-use of subcomponents components, and can thus promote modularity and regularity in solutions. Graphs can also have a computational advantage by reducing the evaluation frequency of commonly reused structure within the solutions. However, the inherent tradeoff between modularity and regularity [14] suggest that reuse of modular substructures also creates internal coupling that may sometimes hinder evolvability. As a special case of graphs, tree encodings can often be adapted to graph encodings which may be more natural to the problem being solved when latent features are commonly reused.

We compare these two encoding approaches systematically using the symbolic regression problem [1,3]. Symbolic regression is a well-known genetic programming benchmark problem with precise definitions of performance and convergence. Additionally, symbolic regression provides a natural measurement of problem complexity and difficulty, allowing us to explore performance trends as problem complexity increases,

2. BACKGROUND
2.1 Symbolic Regression
Symbolic Regression is the problem of identifying the exact mathematical (analytical) description of a hidden system from experimental data [1,2,3,8]. Unlike polynomial regression or related machine learning methods which also fit data, symbolic regression is a system identification method which explicates behavior. Symbolic regression is an important problem because it is related to general machine learning problems but is an open-ended discrete problem that cannot be solved greedily, thus requiring non-standard methods.

2.1.1 Evolutionary Operations
For experiments in this paper, we represent algebraic expressions (candidate solutions) as both tree and graph encodings (we describe these encodings explicitly later). Both variants consist of connected operations, input variables, constant values, and output root.

Operations can be unary operations such as *abs*, *exp*, and *log*, or binary operations such as *add*, *mult*, and *div*. If some a priori knowledge of the problem is known, the types of operations available can be narrowed ahead of time [2,3]. The terminal values (leave nodes) available consist of the function's input variables and the function's evolved constant values.

Mutation in a symbolic expression can change an operation type (eg. change *add* to *sub*), change the arguments of an operation (eg. change $x+1$ to $x+x$), delete an operation (eg. change $x+x$ to x), or add an operation (eg. change $x+x$ to $x + (x*x)$).

Crossover of a symbolic expression exchanges sub-trees or sub-graphs from two parents. For example, crossing $f_1(x) = x^2 + 1$ and $f_2(x) = x^4 + sin(x) + x$ could produce a child $f_3(x) = x^2 + sin(x)$. In this example, the leaf node $+1$ was exchanged with the $sin(x)$ term.

2.1.2 Fitness
The fitness objective in symbolic regression, traditionally, is to minimize error [1,2,3,8] or to maximize correlation [13] on the training set. For experiments in this paper we use correlation fitness since it is a naturally normalized metric that translates well between multiple experiments.

3. THE TREE ENCODING

3.1 Structure
The tree encoding is a popular structure in genetic programming [1], particularly in symbolic regression. Tree encodings typically define a root node that represents the final output or prediction of a candidate solution. Each node can have one or more child nodes that are used to evaluate its value or behavior. Nodes without children are called leaf nodes (or terminals) that evaluate immediately from an input, constant, or state modeled within the system.

Tree encodings in symbolic regression [1,13] are termed expression trees. Nodes represent algebraic operations on children, such as *add*, *sub*, *mult*, *div*. Leaf nodes represent input values (eg. $x_1 = 1$) or evolved constant values (eg. $c_1 = 3.14$). An example expression tree is shown in Figure 1(a).

Evaluating an expression tree is a recursive procedure. Evaluation is invoked by calling the root node, which in turn evaluates its

children nodes, and so on. Recursion stops at the leaf nodes and evaluation collapses back to the root. Recursion can be computationally expensive, particularly in deep trees

3.2 Evolutionary Considerations
Crossover of expression trees swaps two sub-trees from two parent individuals. The crossover points are typically chosen at random in each parent [3,13].

An immediate consequence of this procedure is that offspring can become extremely large by chance. For example a leaf node swapped with the root node of another parent could double the depth of the child's tree. Therefore, it is common practice to crop children or avoid crossovers that produce trees over some threshold depth.

A second consequence is repeated or duplicate structure. For example if the individual encodes the function $f(x) = (x-1)^4$, the sub-expression $(x-1)$ must exist four times in the tree. The duplicate expressions can dominate the crossover point selection focusing recombination on $(x-1)$ sub-trees.

Along the same line from the previous example, duplicate expressions make mutation more difficult. To produce $f(x) = (x - 1.23)^4$ (from the previous example), the constant must be mutated 4 times.

4. THE GRAPH ENCODING

4.1 Structure
The graph encoding is similar to the tree, but child nodes are no longer unique – multiple nodes may reference the same node as its child.

Graph encodings in symbolic regression are termed expression graphs, or operation lists. Each node in the graph can represent algebraic operations, constant values, or input variables. An example graph expression is shown in Figure 1(b).

A useful feature of graph encodings is that they lend well to efficient non-recursive representations. For experiments in this paper, we use a list of operations that modify a set of internal variables, *R*. Local variable represent internal nodes in the graph and are necessary to build-up non-trivial expressions

In the list encoding, each operation in the list can reference one or

Tree: $f(x) = (x + 1)^4$

Graph: $f(x) = (x + 1)^4$
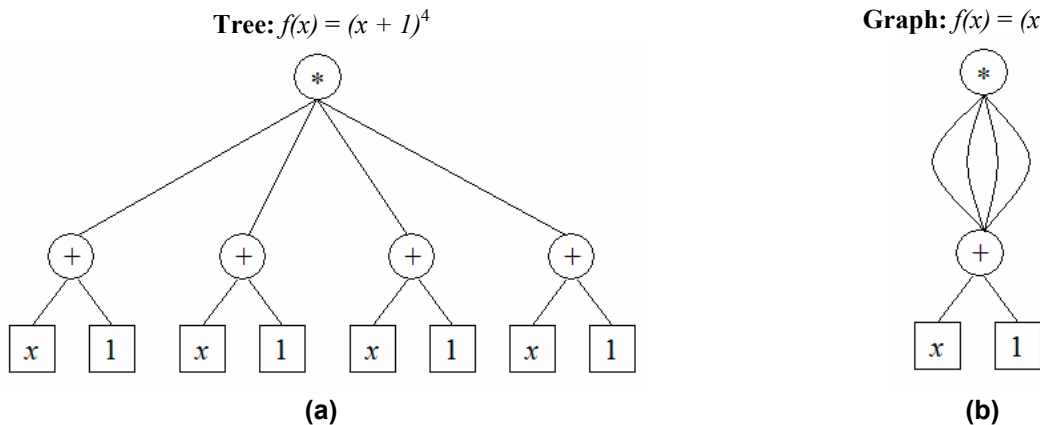

(a)

(b)

Figure 1. Example expressions of $f(x) = (x + 1)^4$ in the tree encoding (a) and graph encoding (b). The graph encoding reuses redundant sub-expressions but is more susceptible to deleterious variation.

more input variables, evolved constants, or internal variables. The result from each operation is then stored in an internal variable. After all operations are completed, the final local variable is returned as output.

Avoiding recursion, without the need to cache or compile a tree expression, provides significant speed up computationally. We will analyze this improvement later in the paper.

4.2 Evolutionary Considerations

Crossover in the graph encoding exchanges two sections of the operator list to form a child. For experiments in this paper we use single point crossover that is chosen randomly.

The graph encoding reuses sub-expressions (multiple operations can reference the same sub-expression). Unlike the tree, crossovers in the graph are less likely to focus on redundant structure since it can be represented in a single operation (or internal variable).

For the same reason, crossover and mutation can be significantly more deleterious. An alteration to an operation producing a reused internal variable will effect all other operations which reference it. In contrast, variation in the tree encoding is localized to individual branches.

5. EXPERIMENTS

5.1 Experimental Setup

The symbolic regression algorithm and past experiments on scaling complexity can be found in [3]. For experiments in this paper, we have simply swap out the tree and graph encodings described earlier and hold all evolutionary parameters fixed.

Table I. Summary of Experiment Setup

Solution Population Size	64
Selection Method	Deterministic Crowding
P(mutation)	0.05
P(crossover)	0.75
Inputs	1
Operator Set	$\{ +, -. *, /, sin, cos \}$
Terminal Set	$\{ x, c_1, c_2, c_3, c_4 \}$
Graph Encoding	
List Operations	16
Internal Variables	4
Evolved Constants	4
Crossover	variable, single point
Tree Encoding	
Initial Depths	1-5
Crossover	single branch swap

Parameters for all experiments are summarized in Table I. In deterministic crowding, offspring replace their most similar parent if they have equal or higher fitness and are discarded otherwise. Population size, mutation probability, and crossover probability are the same used in [3].

For experiments in this paper we use correlation fitness [13] since it is a naturally normalized metric that translates well between multiple experiments and different target functions.

Evolution is stopped after the best candidate solution has converged on the training set (convergence defined later), or after a maximum of one million generations.

5.2 Target Complexity

We define complexity as the number of nodes in a binary tree needed to represent the function [3,8]. Target functions are generated randomly, and then simplified algebraically (eg. collecting terms, canceling quotients, and evaluating constants) to give a more accurate representation of the targets minimum size.

This metric for complexity does not perfectly match problem difficulty. For example, $f(x) = x_1*x_2*x_3$ is most likely more difficult to regress than $f(x) = x_1 + x_2 + x_3 + x_4$ for combinatorial reasons. However, as seen in Section 7, the correlation with problem difficulty is strong and larger target functions take longer to regress symbolically on average for random functions. Random

5.3 Random Target Functions

A key focus of this paper is to examine performance trends between the two encoding schemes over a range of different complexity problems. We collect results on randomly generated functions to get sufficient samples over several complexity targets.

Random targets are generated by randomizing a tree encoding. The target first simplified algebraically before measuring its complexity. Each encoding is then run on the same target functions.

The training data is generated by sampling the target function randomly over the range $\mathbf{R}^n \in [0, 2]$ for all input variables 200 times. The test set is generated similarly by sampling over the range $\mathbf{R}^n \in [0, 4]$.

Results are collected over 500 randomly generated target functions, divided evenly among complexities { 1, 3, 5, …, 19 }, or 50 random targets per complexity. Additionally we test on two input feature sizes: single variable and 8-variable.

5.4 Convergence Testing

Convergence is defined as having greater than 0.9999 correlation on the training set. Evolution is stopped if the best candidate solution reaches this correlation.

Note that convergence on the training set may not mean the target function has converged; the solutions may have overfit to the training data. For this reason we report convergence on the test set (test set correlation greater than 0.9999) in experimental results.

6. SOLUTION COMPLEXITY AND BLOAT

A challenging problem in many genetic programming domains is dealing with bloat. Bloated solutions are those which are excessively complicated. In machine learning, bloat is synonymous with "overfitting" where solutions contain complex structures that do not exist in the target function to explain the fitness objective.

We measure bloat as the complexity of the regressed solution minus the complexity of the target function:

Bloat = (# nodes in solution) – (# of nodes in target)

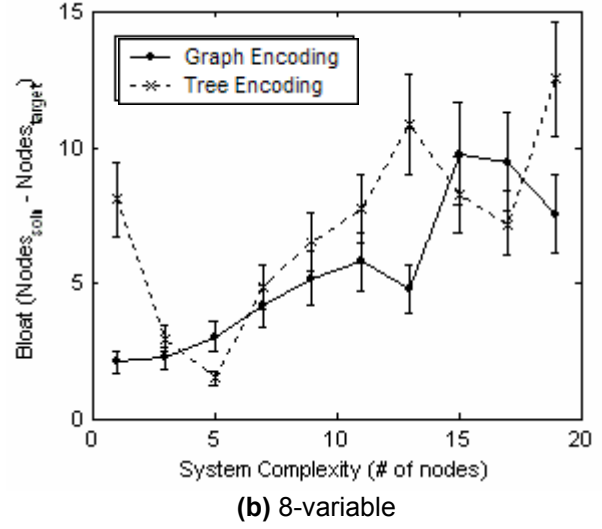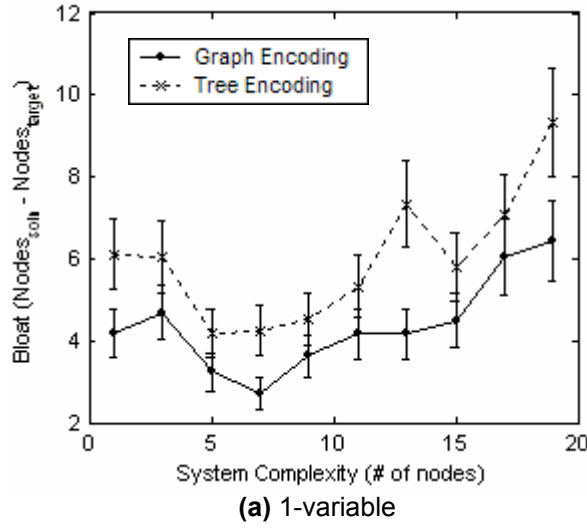**(a)** 1-variable **(b)** 8-variable

Figure 2. Bloat of converged solutions for 1-variable functions (a), and 8-variable functions (b). Each point is averaged over 50 randomly generated target functions. Error bars show the standard error.

This definition of bloat will be zero if the evolved solution is the exact same size as the target (perfect case) or positive it is larger. In rare cases, converged solutions may use fewer nodes if further simplification on the target function is possible but not caught by our algebra library.

We measure the effective number of nodes in the graph encoding by converting it to a binary tree. This always increases the number of nodes but allows better comparison with the tree encoding results.

The mean bloat of each encoding type is shown in Figure 2 at each target function complexity. In the 1-variable case, the tree encoding has higher average bloat over all complexities. The amount of bloat (for both encodings) tends to increase with target complexity. Bloat is also higher on average in the 8-variable targets than the single variable targets.

7. CONVERGENCE

In this experiment we measure the convergence rate for each encoding over target function complexity – the percent of runs where the best solution achieves greater than 0.9999 correlation on the withheld test set.

Figure 3 shows the test set convergence for each complexity target function. Both encodings drop in convergence with higher complexity target functions. Each encoding is run on the same target functions.

The tree encoding achieves slightly higher convergence than the graph encoding over medium sized targets. However, their general trends in both the 1-variable and 8-variable cases appear to be comparable.

8. EVALUATIONS

In this experiment we measure the number of point evaluations
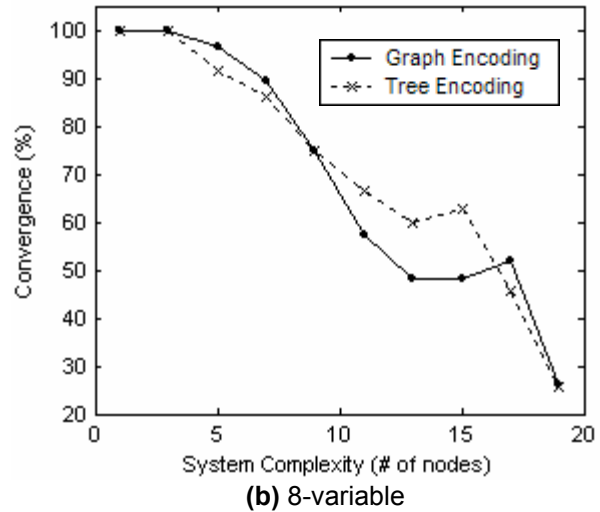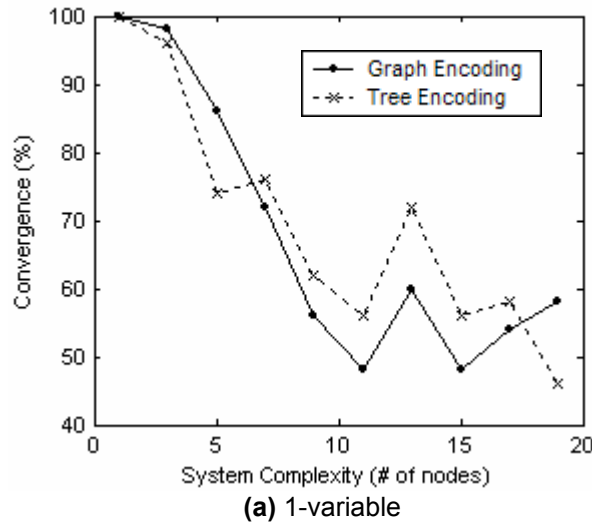

(a) 1-variable **(b)** 8-variable

Figure 3. Test set convergence versus target function complexity for 1-variable functions (a), and 8-variable functions (b). Each point is corresponds to 50 randomly generated target functions.
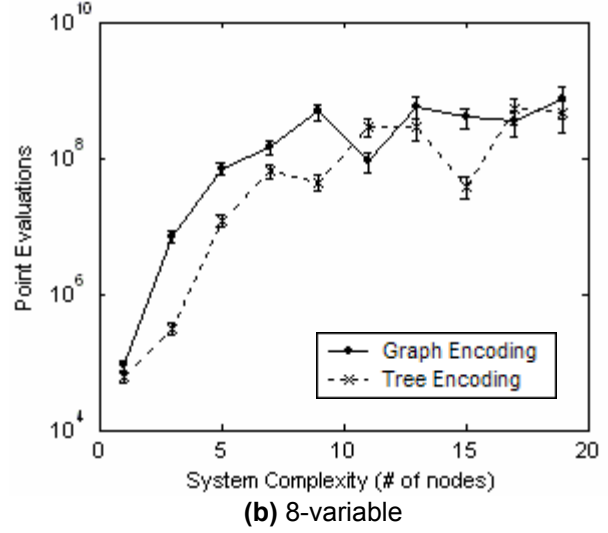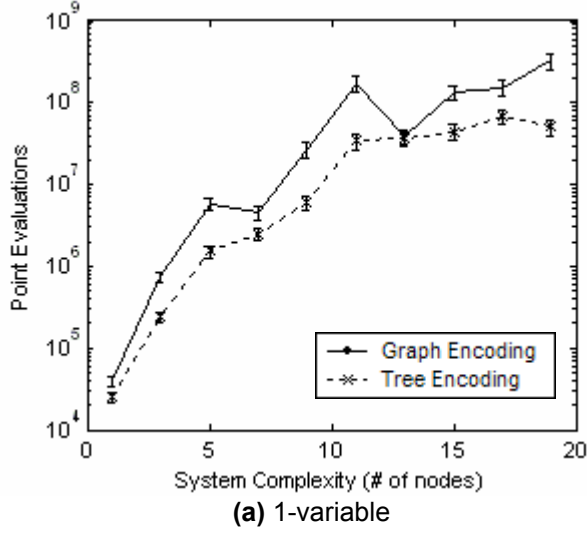
(a) 1-variable   **(b)** 8-variable

Figure 4. The number of point evaluations before convergence on the training set versus the target function complexity for 1-variable functions (a), and 8-variable functions (b). Each point is averaged over 50 randomly generated target functions. Error bars show the standard error.

before convergence on the training set. A point evaluation is a single execution of a candidate solution on a given input. Therefore, this is a metric of the total computational effort required for convergence.

Figure 4 shows the mean number of point evaluations to convergence for each encoding where the runs had converged on the training set. In the single variable case, the graph encoding always takes more evaluations on averaged to converge than the tree encoding. This suggests that the graph encoding is less evolvable, or perhaps more conservative considering it is less likely to bloat.

In the 8-variable case however, the difference in point evaluations decreases for higher complexity targets. At complexity ten and higher both encodings perform roughly the same. These figures show only runs where both encodings converged on the training set. In the 8-variable case the effort appears to require less

computation, but fewer runs were able to converge before a million generations.

9. BENEFICIAL CROSSOVERS

In this experiment we measure the number of beneficial crossover occurring during evolution. A beneficial crossover occurs when a child achieves higher fitness than its most similar parent.

Figure 5 shows the rate of beneficial crossovers for both encodings over the range of complexity target functions. In the single variable case, the tree encoding experiences more beneficial crossovers than the graph encoding, particularly at low complexities.

10. COMPUTATIONAL PERFORMANCE

In addition to evolvability, bloat, and convergence, the efficiency of encodings can have a large impact on the difficulty of problems

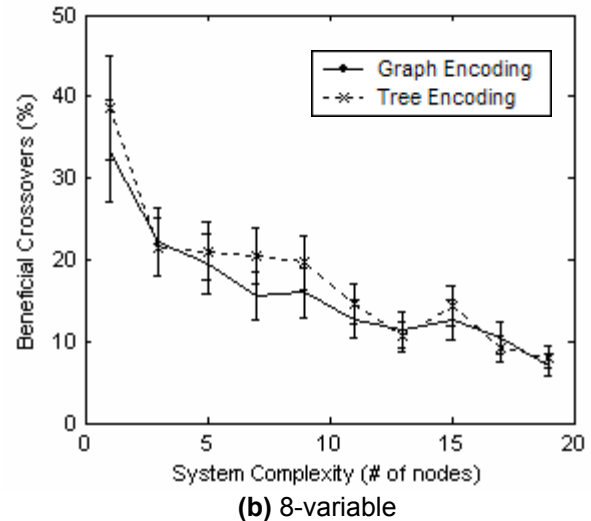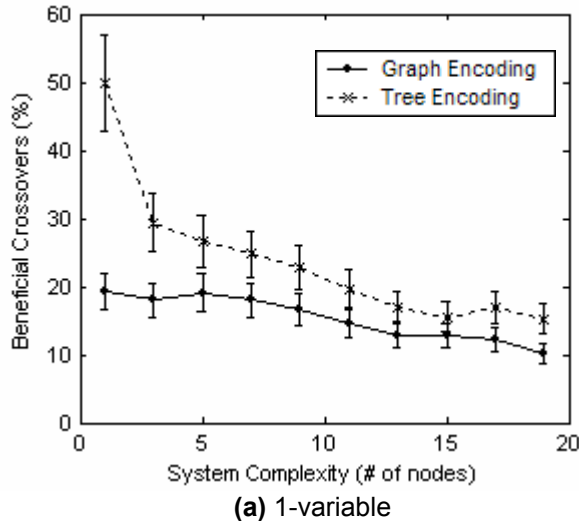**(a)** 1-variable   **(b)** 8-variable

Figure 5. The rate of beneficial crossovers versus target function complexity for 1-variable functions (a), and 8-variable functions (b). Each point is averaged over 50 randomly generated target functions. Error bars show the standard error.

that can be solved in practice. In this section we benchmark the tree and graph encodings.

Figure 6 shows the computational performance, measured in point evaluations per second over a range of complexities. The graph encoding remains roughly constant because it has a fixed encoding size. Variation still exists because it still executes operations in its list that do not affect the output.

The tree encoding is efficient on simple functions of less than five nodes. Performance drops significantly with complexity however as recursion deepens with complexity. The computational performance result indicates the tree encoding does not scale as well with complexity. At five nodes and higher, the graph encoding using an operator list more than triples the performance of the tree encoding.
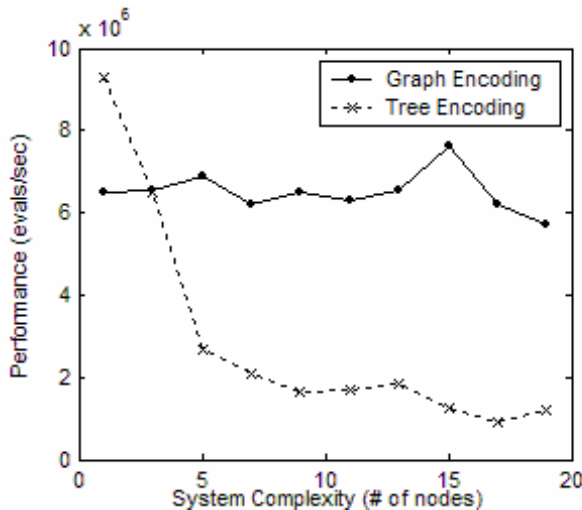

Figure 6. The point evaluations per second versus the function complexity.

11. CONCLUSIONS

We have compared two encoding schemes in increasingly complex problems using symbolic regression. While the tree and graph encodings are similar in application, they offer distinct advantages and disadvantages in genetic programming.

We have tested these two encodings on randomly generated nonlinear target functions, for both single variable and 8-variable input spaces.

Results show that the tree encoding solutions exhibit consistently higher bloat over all complexity targets. The tree encoding however offers slightly higher convergence rate (finding an exact fit) and time to converge, but there was no large trend difference over complexity. The tree encoding experiences more beneficial crossovers (offspring more fit than most similar parent) on single variable targets. Beneficial crossovers decrease with complexity. On 8-varible targets both encodings experienced similar trends in beneficial crossover trends. Finally, the computational comparison shows that the graph encoding to be significantly more efficient than the graph for high complexities.

From these results we conclude the graph encoding to be a attractive alternative to traditional tree based problems (eg. symbolic regression). Graph encodings provide similar performance in convergence over a range of complex target functions and different input sizes, and do so with less bloat. The graph encoding experiences fewer beneficial crossovers and converges slightly slower, however the computational performance outweighs this drawback.

12. ACKNOWLEDGEMENTS

13. REFERENCES

[1] Koza, J.R. Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA: The MIT Press, 1992.

[2] Augusto D. A. and Barbosa H. J. C. "Symbolic Regression via Genetic Programming," VI Brazilian Symposium on Neural Networks (SBRN'00), 01: 22-01, 2000.

[3] Schmidt, M., and Lipson, H. "Coevolution of Fitness Maximizers and Fitness Predictors", GECCO Late Breaking Paper, 2005.

[4] Rafal Kicinger, Tomasz Arciszewski and Kenneth De Jong, "Evolutionary computation and structural design: A survey of the state-of-the-art," Computers & Structures, Volume 83, Issues 23-24, Pages 1943-1978, 2005.

[5] Duffy, J., and Engle-Warnick J., "Using Symbolic Regression to Infer Strategies from Experimental Data," S-H. Chen eds., Evolutionary Computation in Economics and Finance, Physica-Verlag. New York, 2002.

[6] Hoai, N. McKay R., Essam D., and Chau R., "Solving the symbolic regression problem with tree-adjunct grammar guided genetic programming: comparative results," Evolutionary Computation, Vol 2. pp. 1326-1331, 2002.

[7] Rothlauf, F. "Representations for Genetic and Evolutionary Algorithms." Physica, Heidelberg 2002.

[8] Morales, C.O. "Symbolic Regression Problems by Genetic Programming with Multi-branches," MICAI 2004: Advances in Artificial Intelligence, pp.717-726, 2004.

[9] Caruana, Rich, Schaffer, J. David, "Representation and Hidden Bias: Gray vs. Binary Coding for Genetic Algorithms." Fifth International Conference on Machine Learning, 1988.

[10] Breiman, L. "Bias, variance, and arcing classifiers," Technical Report 460, Statistics Department, University of California Berkeley, 2996.

[11] Domingos, P. "A unified bias-variance decomposition and its applications," In Proceedings of the 17th International Conference on Machine Learning, 2002.

[12] Wolpert, D. "On bias plus variance," Neural Computation, 9, pp. 1211–1243, 1997.

[13] McKay, B., Willis, M., Barton, G. "Using a tree structured genetic algorithm to perform symbolic regression," First International Conference on Genetic Algorithms in Engineering Systems, vol. 414, pp. 487–492, 1995.

[14] Lipson H. "Principles of Modularity, Regularity, and Hierarchy for Scalable Systems," Genetic and Evolutionary Computation Conference (GECCO'04), 2004.