SAND2015-2326R

A Comparison of Genetic Programming Variants for Hyper-Heuristics

Sean Harris
Computer Science Department

Dr. Daniel Tauritz
Research Advisor

3/25/2015

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Contents

Abstract

Modern society is faced with ever more complex problems, many of which can be formulated as generate-and-test optimization problems. General-purpose optimization algorithms are not well suited for real-world scenarios where many instances of the same problem class need to be repeatedly and efficiently solved, such as routing vehicles over highways with constantly changing traffic flows, because they are not targeted to a particular scenario. Hyper-heuristics automate the design of algorithms to create a custom algorithm for a particular scenario.

Hyper-heuristics typically employ Genetic Programming (GP) and this project has investigated the relationship between the choice of GP and performance in Hyper-heuristics. Results are presented demonstrating the existence of problems for which there is a statistically significant performance differential between the use of different types of GP.

1 Introduction

Hyper-heuristics is a field of study which aims to be able to automatically create novel algorithms designed to perform better on certain classes of problem than would a general algorithm modified by hand for that purpose. Often this is done through the use of genetic programming techniques, which apply evolutionary

algorithms to functions or algorithms in order to find algorithmic solutions to problems. Genetic programming originated with representing the program structures as trees, and this has continued to be the most common implementation of the technique to this day. However, the field has since grown and diversified to the point where there are now many representations of program structure other than trees which have shown success. As always, it is important to understand which implementation is best to pick when designing the solution to a problem; this paper thus aims to quantify the differences in performance between five genetic programming variants on the boolean satisfiability (SAT) problem: Tree, Linear, Cartesian, Grammatical, and Stack genetic programming.

1.1 Genetic Programming

1.1.1 Tree GP

The original formulation of genetic programming was done with a tree representation. Here, each node on the tree is a function which takes its children as inputs and sends its output to its parent. This is a very intuitive way of thinking about the created programs, because each subtree is a complete sub-program. However, there is no capacity for parts of the tree to be reused in a calculation, which can make some things more difficult to do [1].

1.1.2 Linear GP

Linear genetic programming is inspired by a more direct handling of genetic programs as computer programs, with each node being an instruction in a list that inputs from and outputs to registers that store program state. This capacity to store and reuse data results in linear genetic programming having the potential to create significantly more powerful programs, at the cost of increasing the search space and making it more difficult for the algorithm to construct meaningful programs. The fact that some pieces of the program will not have a meaningful effect on the output can be both helpful as spare genetic information or harmful in its overcomplication of the code [1].

1.1.3 Cartesian GP

Related in the desire of reusing computations is Cartesian genetic programming, which can be thought of as several rows of nodes which take their inputs from previous rows, ultimately forming a graph rather than a tree. This allows for similar behavior to linear GP, including the capacity to store some unused nodes. Cartesian GP has an explicitly defined size, which prevents bloat but also makes its performance very sensitive to its parameters [1].

1.1.4 Grammatical GP

Grammatical genetic programming uses a genotype that, rather than directly representing a set of nodes, represents a numbered list of grammatical expansions which eventually produce its nodes. This potentially allows for a lot more control by the user over the search space, and also facilitates the use of multiple data types in the algorithm. The genotype is represented linearly but can be decoded into a tree for execution [1]. In terms of what can be represented, unless restricted in its grammar this GP variant has the same search space as with tree GP.

1.1.5 Stack GP

Stack genetic programming uses a linear list of instructions, but unlike with linear GP these instructions simply pop inputs from and push their outputs to a data stack [2]. This effectively means that despite representing its genotype similarly to the linear GP, the stack GP actually has the same search space as the tree GP does.

1.2 Boolean Satisfiability Problem

The boolean satisfiability problem, or SAT, is a common problem in the field of computer science. It is popular as a test problem due to its simple implementation and high complexity, but is also notable due to its property of NP-completeness, meaning that many other difficult problems can be quickly reduced to it. The SAT problem is also particularly simple to represent for an evolutionary algorithm. For these reasons, it has been chosen as the problem on which to compare these GP types.

The SAT problem is defined as such: Given a set of boolean variables, and a boolean function of those variables, determine if there is a set of values that the variables can take on that will cause the function to evaluate as true. Generally, determining so involves finding such a set of values for the variables. Thus in many contexts the SAT problem is treated as simply finding those values if they exist.

A major sub-problem of SAT is called 3-SAT, which restricts the form of the boolean function to conjunctive normal form with clauses of three variables: this results in the function taking the form $(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6) \wedge \ldots$, where x_i's can be reused or inverted. A SAT problem can be converted to a 3-SAT problem in polynomial time, so solving the 3-SAT problem is effectively the same as solving the SAT problem. For the purposes of this paper, the algorithms generated by the hyper-heuristics will be solving the 3-SAT problem.

1.3 Related Work

The SAT problem is a well-studied problem in computer science, so many algorithms for searching for solutions already exist. [3] gives a comparison of a number of local search algorithms designed for the SAT problem, which continually improve proposed solutions using a fixed set of rules informed by understanding of the problem. Research has also been performed on the use of evolutionary algorithms, which are instead a type of black-box search algorithm that does not require detailed information about the problem [4]. The use of hyper-heuristics for this problem is intended to be able to generate both types of algorithm, as well as hybrid approaches between the two. Hyper-heuristic approaches to the SAT problem have been studied before [5], but this paper is only using the SAT problem as an environment to study different forms of GP hyper-heuristic, a subject for which there is very little existing research, rather than attempting to improve on prior results.

2 Methodology

In order to determine any difference in performance between the selected genetic programming variants, each of the five were implemented in a common framework which shares as much code as possible between them, in order to minimize differences in performance due to implementation, rather than the actual quality of the variant. Each variant produces a representation of a population individual which codes for an algorithm that can be run against a SAT problem and return a fitness, as well as rules for how to randomly generate, mutate, or recombine individuals of that representation type.

2.1 SAT Problem Generation

The problems against which the variants are evaluated, as well as a separate verification set, are generated randomly by first creating a random set of boolean variable values, and then constructing clauses in the generated problem such that the boolean values selected are a solution to the problem. This guarantees that each problem is possible to solve.

2.2 Evaluating Individuals

Instead of requiring that an individual in the population represent an entire BBSA, it is assumed that the algorithm will take the form of an iterative process which at each step inputs and outputs a set of proposed solutions to a SAT problem. This reduces the amount of code that needs to be evolved to just the actions to be performed during each step.

In order to evaluate an individual, its code is run against each of a set of SAT problems several times, in order to determine that individual's general performance rather than its performance against a specific problem. On each run, an initial population of SAT solutions is generated at random. The individual's code is then run with the initial population as an input, which outputs a new population. This process is then repeated with each step taking the previous step's output as an input. After the termination condition is met, the fitness of the individual for that run is calculated as the number of clauses of the SAT problem which are satisfied by the best solution in the most recent step of the run. The evaluated fitness of the individual is the average fitness of all of its runs, minus a small factor of the amount of nodes used to function as a parsimony pressure, in order to mitigate bloat.

2.3 Meta-Evolution

Individuals are evolved through a generic genetic programming algorithm which after each generation evaluates the fitnesses of the whole population by running them against several problems. The fitnesses of the population are stored, as well as the fitness of the best individual of the generation tested against a larger verification set of problems. The next generation's population is then generated primarily by recombination (with parents selected through tournament selection from the previous generation), with small minorities generated by mutation (also selected by tournament selection) and by truncation selection from the previous generation to ensure that the best solutions survive to future generations. The methods of generating the initial population and performing mutation and recombination are dependent on the variant of genetic programming being tested.

2.4 Genetic Programming Nodes

Each variant of algorithm uses the same set of algorithm nodes which each take inputs and outputs of sets of SAT solutions (except for terminal nodes which take no input). The nodes often take parameters as well, but these are not treated as inputs for genetic programming in order to simplify the generated algorithms and are instead randomly generated when the nodes are with values within a certain range. Each node is a self-contained algorithm which has been taken from existing BBSAs. There are no nodes for evaluation of solutions, instead they are evaluated when their fitnesses are first required. Solution fitness is determined by the amount of clauses in the associated SAT problem which are satisfied by that solution. The selection of nodes was based on work by [6].

2.4.1 Terminal Nodes

The population from the previous step in evaluation can be input as a terminal node. Alternatively, a new randomly generated population of a given size can also be input as a terminal node.

2.4.2 Selection Nodes

Several options are given to select a subset of solutions from a population: tournament selection, fitness-proportional selection, truncation selection, and random subset. These all have as a parameter the amount of individuals to select. Tournament selection also has the size of tournament as a parameter. Truncation selection does not necessarily sort the population beforehand; instead a separate sort node is given.

2.4.3 Mutation Nodes

A number of algorithms of varying complexity are given in order to allow an algorithm to modify solutions. These are applied to each solution in the input population. A bitwise mutation operation is provided to allow random changes to solutions at a given rate. There is also a pair of greedy mutation algorithms: one checks all variables which could be flipped and flips the one which causes the highest positive increase to solution fitness, if any, and the other simply flips a random variable which will increase fitness.

Also provided is a function which uses stepwise adaptation of weights to select which variable to change: The solution stores a value for each clause in the problem representing how long that clause has been unsatisfied. When the SAW mutation node is called, it selects the clause which has been unsatisfied for the

longest time, and flips a random variable in it. Then the counters are incremented by one for each false clause and reset to zero for each true clause [7]. An implementation of the related Novelty function [8], [3], a result from studies of local search algorithms, is included as well.

2.4.4 Other Nodes

In addition to the aforementioned sorting node, which sorts the members of a population by number of satisfied clauses, a set union node is also provided which will combine two population sets into one. This allows for branching in the algorithms generated. After each step in evaluation of the algorithms, the population size is automatically truncated if it exceeds a maximum size, in order to prevent slowdown resulting from misuse of the union node.

2.5 Evolutionary Operators

Due to differences in the ways that the different genetic programming variants represent their algorithms, different evolutionary operators had to be used between some of the different algorithms:

2.5.1 Tree GP

The tree genetic programming implementation uses the well-established ramped half-and-half algorithm to generate individuals, which provides a combination of both full trees and smaller, more diverse trees. Subtree mutation is used as a mutation operator by replacing a random subtree with a new subtree with depth equal to a gaussian random value centered on the original subtree's depth. The recombination operation used simply replaces a random subtree on the parent with a random subtree on a donor tree.

2.5.2 Linear GP

The linear genetic programming implementation generates individuals as a random number of random nodes, up to a maximum size. The mutation and recombination operators are designed in order to be similar to the ones used in tree GP: the mutation operator replaces a random subsection of the program with another one of a similar size to the original using a gaussian random offset, and the recombination operator selects a random subsection of a donor and places it randomly into the parent, overwriting anything already using that section and increasing the program length if necessary.

2.5.3 Cartesian GP

Cartesian genetic programming's unique representation does not allow for a lot of variation in the design of operators, so individuals are created by randomly selecting nodes to fill out the grid; mutation randomly replaces nodes at a set rate, and recombination uses a uniform crossover.

2.5.4 Grammatical and Stack GP

Grammatical and stack genetic programming both store their representations in a list (a list of expansions and a list of nodes respectively), so this implementation uses the same operations which are used for linear GP, for consistency.

2.6 Experimental Parameters

Experimental parameters were chosen by hand after a small number of tests were performed for each of them. The high computation time of meta-evolution unfortunately made a more exhaustive tuning of parameters prohibitive.

The evaluation data set contained 3 problems, and the verification set contained 8, all of which were 3-SAT problems containing 2000 clauses of 500 variables. This problem size was chosen so that only the highest-performing algorithms generated would get perfect fitnesses, in order to better distinguish the GP types. In order to ensure more accurate results, the generated algorithms were tested on each problem 3 times.

The meta-evolution for all GP types used a population size of 20 individuals, with future generations selected by tournament selection with tournament size of 5. 70% of children were generated through recombination, and 20% were generated through mutation. The remaining 10% was taken by truncation selection from the previous generation to ensure a small amount of elitism. Runs were terminated after 40 generations, which was generally enough to ensure convergence.

For the evaluation of individual algorithms generated by the hyper-heuristic, a maximum population size was set at 100 solutions (overly large populations were truncated), though many algorithms used smaller sizes. These were given 30 seconds of wall time to run on each evaluation. The reason for the use of wall time rather than number of evaluations was because the number of evaluations per node did not correlate well with the actual computational cost of executing those nodes. Thus, some nodes which were computationally expensive but did not make heavy use of evaluations might be unfairly selected for if only evaluations were limited.

Tree GP used a parsimony pressure of 0.1 per node, and a soft maximum size of 20 nodes. Individuals which exceeded that maximum size were heavily penalized but otherwise treated normally. Fitnesses recorded in the results section of this paper do not include these penalties. The initial population was generated to a depth of 5.

Linear GP used 3 registers, one of which was designated as an output register. It used a parsimony pressure of 0.1 per node and had a soft maximum size of 20 nodes. The initial population was generated with 10 nodes.

Cartesian GP individuals had 20 layers of width 3, with the option to take input from nodes at most 5 layers higher. Due to the fixed maximum size of individuals, no penalties were used.

Grammatical GP used a grammar equivalent to what was allowed for tree GP. It used a parsimony pressure of 0.1 per expansion and had a soft maximum size of 50 expansions. The initial population was generated with 30 expansions. This approximately corresponds to an equivalent amount of nodes as was given for the other variants, because the expansions also encoded parameters for the nodes.

Stack GP used a parsimony pressure of 0.1 per node and had a soft maximum size of 20 nodes. The initial population was generated with 10 nodes.

3 Results

After running each genetic programming variant thirty times, the best individual from each run was evaluated three times against the verification set. The resulting average fitnesses are shown in Table 1, with a score of 2000 indicating that the best algorithm found was able to repeatedly find satisfying solutions to all of the verification problems after 30 seconds. These results were compared to their reported performances on the evaluation data sets used in the course of genetic programming. While their performance on the evaluation sets was somewhat inflated, as this was the value which they were selected for, the closeness of the algorithms' performance on a set they were not bred to solve (usually within 10-20 fitness points) indicates that the fitness results are indicative of general capability.

Statistical analysis of the data shows that tree and stack GP perform similarly to each other, as do linear and Cartesian GP. However, there is a statistically significant difference between these two pairs, and with grammatical GP. Thus, the choice of GP type can make a real difference in the performance of a hyper-heuristic for the SAT problem. Tree and stack GP give the best results, followed by linear and Cartesian, followed by grammatical.

4 Conclusions

Tree and stack GP performed very similarly to each other, which was not unexpected. This is because stack GP generally functions as linearly stored postorder tree, and thus with the exception of stack GP's capacity to contain introns, have an identical search space. Many of the generated algorithms between the two were very similar in shape. While the mapping is less exact, the graph-based representations of linear and Cartesian GP also have similar search spaces and performed similarly as expected.

The differences between the tree and graph-based GPs' search spaces appear to be largely responsible for the performance disparity. None of the solutions generated by the hyper-heuristics make heavy use of

Tree	Linear	Cart.	Gram.	Stack
1983	1891	1954	1824	1991
1990	1853	1994	1989	1980
1969	1990	1971	1949	1998
1997	1916	1964	1971	1977
1993	1907	1954	1824	1987
1992	1986	1989	1988	1988
1999	1958	1947	1936	1991
1991	1991	1991	1966	1930
1991	1992	1990	1823	1995
1975	1991	1990	1902	1999
1924	1979	1965	1999	1988
1997	1980	1988	1885	1999
1993	1990	1943	1833	2000
1999	1986	1995	1902	1993
1990	1985	1976	1812	1999
1990	1845	1924	1943	1986
1999	1958	1966	1986	1956
1994	1989	1993	1990	1996
1975	1989	1966	1990	1969
1993	1988	1981	1998	1979
1977	1932	1988	1993	1996
1991	1993	1990	1991	1963
1995	1985	1990	1853	1996
1999	1884	1976	1873	1991
1999	1988	1967	1987	1971
1997	1982	1963	1867	1982
1983	1990	1899	1962	1999
1992	1971	1988	1852	2000
1969	1993	1919	1963	1978
2000	1991	1961	1991	1989

Table 1: Average number of clauses satisfied by best individual generated per run, over the verification set (out of 2000)

	Tree	Linear	Cart.	Gram.	Stack	Rand.
Eval.	1995.8(6.81)	1980.9(29.01)	1991.0(9.50)	1980.0(41.20)	1996.7(3.91)	n/a
Veri.	1987.9(14.78)	1962.4(43.43)	1969.4(23.84)	1928.1(65.03)	1985.5(5.33)	1750.2(4.80)

Table 2: Average performances of algorithms on evaluation and verification sets (out of 2000), with standard deviations in parentheses; Rand. indicates the average number of clauses satisfied by random SAT solutions

branching, with most solutions having zero or one union nodes. The solutions in the two graph GPs were found to almost never re-use results, which negates one of their primary advantages over the tree-based approaches. This leaves them with a more complex set of solutions to search through with nothing more to show for it. The grammatical GP likely failed for a similar reason: the use of a grammar is intended to allow the use of prior knowledge of the shape of solutions to constrain the search space. The use of a generic grammar that does not constrain the search space then results in little benefit at the cost of a larger genotype whose genes are less meaningful out of context.

These results apply specifically to the use of these GP types as hyper-heuristics for the SAT problem, but the detection of significant differences between their performance has further-reaching implications. The existence of problems for which the choice of GP matters indicates that users of hyper-heuristics need to carefully select their GP type for the problem in order to get the best results.

5 Future Work

The purpose of this study was to determine the relative performances of different GP types. However, the SAT problem is a very well-studied problem and there are many conventional algorithms known to solve it effectively [5]. It would be useful to use some of the existing algorithms to provide a benchmark performance level in order to give context to the capacities of the different algorithms generated by the hyper-heuristics.

The SAT problem is a simple problem which is effective for testing hyper-heuristics on, but features of it such as the low amount of branching that occurs in most solutions and the lack of any obvious and useful grammatical constructions mean that not all of these variants are able to show their strengths. Testing these against each other on a larger sampling of problems would give a better understanding of how these variants perform in general, and might provide insight in how to match problems to the GP types best suited for them.

Additionally, each of these GP types were tested in a fairly basic formulation. This was done deliberately in order to ensure a more even comparison focused on the fundamental properties of each GP type. However, there exist modifications and improvements to these core GP types which would likely be used in many real-world applications and it would be useful to see how these affect GP performance to get results more applicable to realistic scenarios.

6 Acknowledgments

This work was done under the supervision of Dr. Daniel Tauritz, and with assistance of students from his Natural Computation Laboratory at Missouri University of Science and Technology including Travis Bueter, Matthew Martin, Caleb Roberts, and Alex Bertels.

This research was funded in part by Sandia National Laboratories and in part by Missouri S&T's OURE program.

References

[1] Riccardo Poli, William B Langdon, and Nicholas F Mcphee. *A Field Guide to Genetic Programming*. Number March. Lulu Press, Inc., 2008.

[2] T. Perkis. Stack-based genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 148–153, 1994.

[3] Holger H Hoos and Thomas Stützle. Local Search Algorithms for SAT: An Empirical Evaluation. *Automated Reasoning*, 24(4):421–481, 2000.

[4] Jens Gottlieb, Elena Marchiori, and Claudio Rossi. Evolutionary algorithms for the satisfiability problem. *Evolutionary computation*, 10(1):35–50, 2002.

[5] Mohamed Bader-El-Den and Riccardo Poli. Generating SAT local-search heuristics using a GP hyper-heuristic framework. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4926 LNCS, pages 37–49, 2008.

[6] Matthew A Martin and Daniel R Tauritz. A Problem Configuration Study of the Robustness of a Black-Box Search Algorithm Hyper-Heuristic Categories and Subject Descriptors. In *Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion*, pages 1389–1396, 2014.

[7] A.E. Eiben and J.K. van der Hauw. Solving 3-SAT by GAs adapting constraint weights. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, pages 81 – 86, Indianapolis, IN, 1997.

[8] David McAllester, Bart Selman, and Henry a Kautz. Evidence for Invariants in Local Search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence (AAAI'97/IAAI'97)*, pages 321–326, 1997.