A Comparison of Genetic Programming Variants for Hyper-Heuristics

Sean Harris
Natural Computation Laboratory
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri, U.S.A.
snhcn6@mst.edu

Travis Bueter
Natural Computation Laboratory
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri, U.S.A.
tjbxv7@mst.edu

Daniel R. Tauritz
Natural Computation Laboratory
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri, U.S.A.
dtauritz@acm.org

ABSTRACT

General-purpose optimization algorithms are often not well suited for real-world scenarios where many instances of the same problem class need to be repeatedly and efficiently solved. Hyper-heuristics automate the design of algorithms for a particular scenario, making them a good match for real-world problem solving. For instance, hardware model checking induced Boolean Satisfiability Problem (SAT) instances have a very specific distribution which general SAT solvers are not necessarily well targeted to. Hyper-heuristics can automate the design of a SAT solver customized to a specific distribution of SAT instances.

The first step in employing a hyper-heuristic is creating a set of algorithmic primitives appropriate for tackling a specific problem class. The second step is searching the associated algorithmic primitive space. Hyper-heuristics have typically employed Genetic Programming (GP) to execute the second step, but even in GP there are many alternatives. This paper reports on an investigation of the relationship between the choice of GP type and the performance obtained by a hyper-heuristic employing it. Results are presented on SAT, demonstrating the existence of problems for which there is a statistically significant performance differential between the use of different GP types.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search; I.2.2 [**Artificial Intelligence**]: Automatic Programming–*program modification, program synthesis*

Keywords

Hyper-Heuristics, Genetic Programming, SAT

GECCO '15, July 11 - 15, 2015, Madrid, Spain
© 2015 ACM. ISBN 978-1-4503-3488-4/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2739482.2768456

1. INTRODUCTION

Hyper-heuristics is a field of study which aims to automatically create novel algorithms designed to perform significantly better on a specific class of problems than would a general-purpose algorithm, trading a very high upfront computational cost for increased savings over time as the automatically generated custom algorithm is repeatedly applied to instances of the specified problem class. This makes hyper-heuristics a good match for real-world problems. For instance, hardware model checking induced Boolean Satisfiability Problem (SAT) instances have a very specific distribution which general SAT solvers are not necessarily well targeted to. Hyper-heuristics can be used to automate the design of a SAT solver customized to a specific distribution of SAT instances.

The first step in employing a hyper-heuristic is creating a set of algorithmic primitives appropriate for tackling a specific problem class. The second step is searching the associated algorithmic primitive space. Hyper-heuristics have typically employed Genetic Programming (GP) to execute the second step, but even in GP there are many alternatives. This paper reports on an investigation of the relationship between the choice of GP type and the performance obtained by a hyper-heuristic employing it, with the hypothesis being that there are problems for which there is a statistically significant performance differential between the use of different types of GP. Subsection 1.1 provides a summary overview of five main types of GP, namely tree-based GP (TGP), linear GP (LGP), Cartesian GP (CGP), Grammatical Evolution (GE), and stack-based GP (SGP). The problem class chosen to experiment on is SAT, being a well-studied NP-complete problem for which it is straightforward to generate custom problem instances. Subsection 1.2 provides a quick overview of SAT.

The rest of the paper is organized as follows: Section 2 provides an overview of related work, Section 3 describes our methodology for comparing hyper-heuristic performance between the five different GP types, Section 4 presents our results, Section 5 provides our conclusions, and Section 6 proposes the next steps in this line of inquiry.

1.1 Genetic Programming

This section discusses each of the GP types explored in this paper. Each GP is described by its origin, structure, and characteristics relative to hyper-heuristics.

1.1.1 Tree-based GP

The use of tree-based structures in genetic algorithms was first introduced in [1] and further explored in [2]. Koza later introduced and popularized the use of TGP for automating the generation of computer programs [3]. It naturally represents solutions in the form of parse trees composed of the primitives and terminals of a program. The fact that any search algorithm can be represented as a parse tree implies that any search algorithm can be evolved from a tree data-structure. This makes TGP seem like a natural choice for use in hyper-heuristics [4].

In the tree representation, internal nodes are the program's primitives, while the terminal nodes are the program's inputs. Parent nodes take the output of child nodes as their inputs, having as many children as their primitive's arity. A depth-first recursive process is used to get the values of children first, and then execute the parent nodes. The output of the root node becomes the program's output.

Strong-typing and the reuse of data are common elements of computer programs, but are not expressed well in tree representation. Normally, nodes are restricted to a single type for inputs and outputs in order to simplify genetic operations and ensure that all programs generated are valid for execution. However, genetic operations can be implemented to respect typing at the cost of greater implementation and run-time complexity. The tree structure also restricts node output to only being used once in a program. A program that uses the same data multiple times can still be represented in TGP, but requires the same sub-tree that generates the data to be present at each reference. Koza addressed this with *automatically defined functions* (ADFs), which are program branches that can be called multiple times by the main program and other ADFs [5].

A characteristic of tree representations is that each sub-tree correlates to a complete sub-program; recombination takes advantage of this to create children by swapping sub-trees of the parents. Mutation uses it to generate children by copying the parent and generating a new sub-tree at a random node. These generally result in high locality between parents and children. However, this can also result in nodes closer to the root nodes of individuals to become common, and potentially cause premature convergence [6].

With strong-typing methods and components like ADFs, TGP can be used to create solutions for a great variety of scenarios. This makes it a good fit for hyper-heuristics, but does not mean that it is the best tool for every scenario. In scenarios where the search space is rougher, with many sub-optima, having high-locality during evolution and low-diversity in the population can make finding the global optima harder [7].

1.1.2 Linear GP

Linear GP (LGP) refers to any form of GP that uses a linear list of instructions for representing programs, to which there are many variants [8, 9, 10]. The variant described in this paper refers to a generalized form of LGP as presented in [11]. Instead of using instructions from a specific programming language, this generalized LGP uses user-defined operations for creating programs. This allows programs to be evolved using higher-level primitives, while still maintaining the linear structure.

In LGP, a program is represented as a fixed- or variable-length list of primitives. Instead of terminal operations, registers are used for storing output and reading input, which are pre-loaded with values at the beginning of execution. The input and output registers for a node are defined as part of its parameters. One or more registers are designated as the sources of the program's output, while the others exist as supplementary registers.

The use of multiple registers gives LGP the ability to create complex programs by storing data in supplementary registers where it can be reused multiple times. However, the size of the search space and quality of programs are very sensitive to the number of registers used. If too few registers are used, it may be impossible for LGP programs to produce good solutions that require greater complexity. As more registers are included, the search space size increases as does the potential for intron operations, or operations that do not contribute to the final result. Although evidence has shown that introns can be beneficial in GP [12, 13], too many can cause over-complication of programs and lower locality from genetic operations [7].

Since linear representations adhere to a less rigid program structure, the effect that crossover has on diversity and locality is fairly different than that seen in tree representations. Crossover can hinder the capability for high locality and be fairly destructive to sub-sequences. This makes it harder for good sub-sequences to be passed on to offspring, but it can also result in a greater population diversity.

LGP is capable of creating programs with the same functionality as those created by TGP, but the differences in their structures and components can lead to differences in how hard it is to evolve a particular program. Given its lower locality during evolution, LGP may be best suited for hyper-heuristics solving problems where the search space is rougher so the hyper-heuristic may benefit from the lower locality in order to escape local optima. Regardless, great consideration should be given to the number of registers being used for any problem LGP is applied to.

1.1.3 Cartesian GP

CGP was proposed by Miller et al. in 2000 [14], which they derived from their earlier method for evolving digital circuits [15]. It has two characteristics that separate it from other GP types. First, CGP has a distinct separation between genotype and phenotype representations. Second, programs are represented as directed acyclic graphs, containing input, computational, and output nodes. The graph can be thought of as a 2-D grid with input nodes on one side, columns of computational nodes in the middle, and output nodes on the other side. Output nodes can be connected to any input node or computational node. Computational nodes can be connected to any input node and other computational nodes in prior columns of the grid. In the graph, input and computational nodes can be connected from other nodes any number of times, or not at all. The program genotype is a list of genes − integers representing a primitive or input location. Groups consisting of a function gene and multiple input genes define the computational nodes of the graph. At the end of the genotype are the output genes. The program phenotype is an abstract representation of the graph derived from the genotype via a mapping process. Instead of generating the graph structure, the function and input information of necessary nodes are stored in an array to be used by a decoding algorithm for calculating outputs. The mapping process starts with the output genes and continues through all the referenced input locations until ending with program inputs. Any nodes not referenced as inputs are not included and do not contribute to outputs. The nodes used are contained in the array in numeric order, ensuring output is generated before it is requested as input. The decoding algorithm uses a second array for managing inputs and outputs, initialized with the program inputs. Each primitive in the program array is executed, in order, until the program outputs have been generated.

CGP's representation makes it easier for the output of good sub-graphs to be reused in multiple parts of a program, giving it the potential to find good solutions faster. It has also been shown to benefit from a large presence of introns [16]. If crossover is used at a very low rate to minimize disruption, diversity can still be managed through neutral drift. Neutral drift occurs when identical programs are mutated, but with mutation only affecting their introns. These programs will continue to have identical functionality, until an active node is finally mutated, and will result in greater diversity in the affected sub-graph. The genotype-phenotype mapping also benefits CGP by allowing simpler genetic operations to be used for evolving the graph. Although it is a notable feature of CGP, it is not currently known if being able to output multiple populations has any benefit in hyper-heuristics. As with LGP, crossover operations in CGP can be very destructive to sub-graphs. Most forms of crossover between two gene lists are not likely to result in a child with similar functionality. If crossover is used often, the population may have too great of a diversity with beneficial sub-graphs being frequently disrupted [16].

Since directed acyclic graphs can be translated into derivation trees, CGP and TGP programs should exist within similar search spaces for the same primitive set and number of outputs. Their ability to derive similar programs differs greatly, however. While TGP can be less restricted in shaping derivation trees and form new sub-trees more easily, CGP can better represent derivation trees that contain reoccurring sub-trees. CGP can also be less disruptive to sub-trees during mutation. This implies that there could be problems where a hyper-heuristic using CGP develops a better search algorithm faster than one using TGP. However, the effect that genetic operators have on traversing the search space of algorithms and wasted evaluations could still cause CGP to be less ideal for many problems [17].

1.1.4 Grammatical Evolution

Grammatical Evolution (GE) was proposed for creating programs in languages besides LISP, while still being able to retain syntactic context during evolution [10]. It was built on the work of previous grammar-based methods and attempted to address some of their shortcomings. Like in CGP, GE programs also have a distinct genotype and phenotype, using a mapping process between them. GE programs are defined by context-free grammars using Backus-Naur form [18] with production rules defined by the user. The grammar's terminal and non-terminal symbols represent the program inputs and primitives, respectively. The genotype is a variable-length sequence of integers which are used to expand the grammar and decode the phenotype derivation tree. This is done by using the integer value to determine which production rule to select for the current symbol.

The process continues to repeat through the sequence until it either creates a complete program or hits a maximum number of repeats. If the latter occurs, the process ends with the worst fitness value being assigned. If a complete mapping is successful, the result is the phenotype − an executable representation of the derivation tree. The use of grammars allows users to ensure strong-typing and meaningful structures in programs. This is especially important if the user knows how the search space can be restricted to improve the density of good solutions. Careful consideration should be taken when defining the grammar as undesired bias can be easily introduced.

Low locality has been a source of debate surrounding GE's genotype-phenotype mapping. If an integer is changed in the genotype, all following integers are likely to be mapped to different production rules, possibly resulting in a vastly different program. In other words, two neighboring genotypes may have phenotypes with little structural similarity. This means even small mutations could result in a more random walk of the search space, making defining the grammar's restriction of the search space more critical for finding good solutions faster. Low locality may be a primary cause of poor performance by GE and users are encouraged to develop more localized mapping to increase performance [7]. If GE's grammar was used to restrict the search space to that of another GP type, it probably would not find a good solution faster due to low locality. Also, while the wrapping mechanic helps reduce it, GE has a chance to produce genotypes that do not result in complete phenotypes. This can introduce wasted population space, which is not found in TGP, LGP, and CGP. Nevertheless, GE still may be useful to employ in a hyper-heuristic that solves a problem class where certain structures are known to be beneficial or detrimental, and the search space can be accordingly restricted.

1.1.5 Stack-based GP

SGP is a form of linear GP implemented to be simpler and more efficient than TGP, while matching, if not surpassing it, in performance [9]. SGP's unique characteristic, and name, comes from its use of data-stacks instead of registers for managing input and output of operations. Programs are represented as linear sequences of primitives and terminals. When a terminal operation is performed, the terminal's value is simply pushed onto the stack. When a primitive operation is performed, the stack is first checked if it has the appropriate number of inputs to match the primitive's arity. If it does, the inputs are popped off the stack and passed to the primitive for execution, and the result pushed onto the stack. If it does not, however, the stack is left unchanged and the operation is skipped, effectively becoming an intron. Generally, the top element on the stack after execution becomes the program's output.

Although a derivation of linear GP, SGP actually shares the same search space as TGP. This is due to the first-in-last-out nature of stacks which results in the program being executed like a post-order tree representation. However, SGP has features that can give it an advantage over TGP. Introns are allowed by SGP, which had previously been mentioned to be useful. The reuse of data can be easily accomplished in SGP by implementing a special duplication operator in the primitive set. Strong-typing can also be enforced by implementing multiple stacks, each for a specific type [19]. Like LGP, crossover operations will generally result in a lowered locality. Again, although this makes passing good subsequences to children more difficult, it also has the potential to maintain population diversity and reduce the time spent searching local optima. Since their programs exist in the same search space, SGP may be the best direct alternative to TGP in hyper-heuristics. SGP has the potential to find a good solution faster for a problem where the program search space is found to have many sub-optimas. It may also produce better results for a wider variety of problem classes since it can benefit from the characteristics of other GP types, such as introns, strong-typing, and the reuse of data. However, these benefits may be hampered for certain problems due to its lower locality and the possible destruction of good sub-trees caused by crossover.

1.2 Boolean Satisfiability Problem

The Boolean Satisfiability Problem (SAT) is a classic problem class 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 reduced to it. SAT solution instances are also particularly simple to represent. For all these reasons, it has been chosen as the test problem on which to compare the hyper-heuristic performance differential between the previously discussed GP types.

The SAT problem is defined as follows: given a Boolean formula, determine if there is a set of values that the Boolean variables can take on that will cause the function to evaluate as true. Generally, determining this 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. Any SAT problem can be converted to 3-SAT in polynomial time, so solving 3-SAT is effectively the same as solving SAT. For the purpose of this paper, the algorithms generated by the hyper-heuristics will be solving 3-SAT.

2. RELATED WORK

[20] provides a comparison of a number of local search algorithms designed for SAT, 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 for solving SAT [21]. Hyper-heuristic approaches to the SAT problem have been studied before [22, 23], but this paper is only using the SAT problem as an environment to study different GP types in hyper-heuristics, a subject for which there is little to no existing research, rather than attempting to improve on prior SAT solver results.

Hyper-heuristics have typically employed GP, such as TGP for competitively evolving Black Box Search Algorithms (BBSAs) [24], grammar-based LGP to find competitive tours for the Traveling Salesman Problem [25], and GE for automatically designing competitive evolutionary algorithms [26].

Comparisons of multiple GP types on different problem classes has not been widely explored. Perkis compares his SGP to TGP using benchmarks by Koza in [3] for symbolic regression, boolean majority on, and boolean even 3-parity problems [9].

Oltean and Grosan compared different forms of linear-style GP, including LGP and GE, on several numerical experiments [27].

3. METHODOLOGY

In order to determine any difference in performance between the selected GP variants, each of the five were implemented in a common framework which facilitated sharing as much code as possible, in order to minimize differences in performance due to implementation, rather than the intrinsic ability of the variants. While this does have the potential to create bias by flattening out certain implementation details specific to individual GP types which improve their performance, the alternative of attempting to maximize the performance of each variant would similarly bias results by allowing for uneven qualities of implementation. Each variant produces a representation of an 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.

3.1 SAT Problem Generation

The problems against which the variants are evaluated, as well as a separate test 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.

3.2 Evaluating Individuals

Instead of requiring that an individual in the population represent an entire SAT solver, it is assumed that the algorithm will take the form of an iterative process which at each step inputs and outputs a population of multiple sets of variable assignments as 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 when applicable, in order to mitigate bloat.

3.3 Meta-Evolution

Individuals are evolved through a generic GP 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 test 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 GP being tested.

3.4 Genetic Programming Nodes

Each GP variant employs 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 GP in order to simplify the generated algorithms and are instead randomly generated with values within a certain range (1 to 100 for population sizes such as selection operators, 0 to 1 for percentages like mutation rate) when the nodes are first created. These values do not change after creation; mutation only affects parameters through fully replacing the node. Each node is a self-contained algorithm which has been extracted from existing SAT solvers. There are no nodes for evaluation of solutions, instead they are evaluated when their fitnesses are first required. Solution fitness is determined by the number of clauses in the associated MAXSAT problem which are satisfied by that solution. The selection of nodes was based on previous published hyper-heuristic work [24].

3.4.1 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.

3.4.2 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 number 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.

3.4.3 Mutation Nodes

A number of mutation operations of varying complexity are provided 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 operations: 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 [28]. An implementation of the related Novelty function [29, 20], a result from studies of local search algorithms, is included as well.

3.4.4 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.

3.5 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:

3.5.1 Tree-based GP

The TGP 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.

3.5.2 Linear GP

The LGP 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 TGP: 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.

3.5.3 Cartesian GP

CGP'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.

3.5.4 Grammatical Evolution and Stack-based GP

GE and SGP 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 LGP, for consistency.

3.6 Experimental Parameters

Experimental parameters were manually tuned; the high computational cost of meta-evolution unfortunately made a more exhaustive tuning of parameters prohibitive. The training data set contained 3 problems, and the test 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. To increase result accuracy, 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% 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. While the small population size limits how much of the search space can be visited, it is necessary due to the long execution time of meta-evolution. However, with these parameters evolution generally converges within the 40 generation limit.

For the evaluation of individual algorithms generated by the hyper-heuristic, the 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.

TGP used a parsimony pressure of 0.1 per node and a soft maximum size of 20 nodes; individuals which exceeded that maximum were heavily penalized, but otherwise treated normally. Fitnesses recorded in the results section do not include these penalties. The initial population was generated to a depth of 5. During mutation, a normal function with standard deviation 2 was used to determine the change in depth of replaced segments. LGP 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. During mutation, a normal function with standard deviation 2 was used to determine the change in size of replaced segments. CGP 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. The mutation rate used was 0.1. GE used a grammar equivalent to what was allowed for TGP. 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. The mutation function used a standard deviation of 5 for similar reasons. SGP 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. The mutation function used a standard deviation of 2.

4. RESULTS

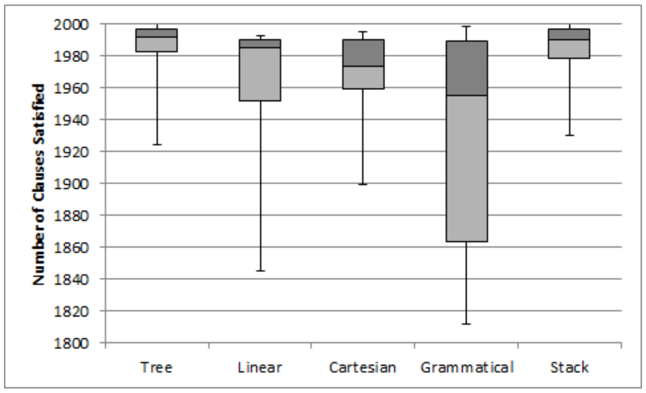

Figure 1: Box plot describing the average number of clauses satisfied by the best individuals of each run, on the test set.

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

Table 1: Average performance of algorithms on training and test sets (out of 2000), with standard deviations below their respective values in parentheses; Rand. indicates the average number of clauses satisfied by random SAT solutions

After running each GP variant thirty times, the best individual from each run was evaluated three times against the test set. The resulting average fitnesses are shown in Figure 1, with a score of 2000 indicating that the best algorithm found was able to repeatedly find satisfying solutions to all of the test problems after 30 seconds. These results were compared to their reported performance on the training data sets used in the course of GP. While their performance on the training 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.

While the fitness scores listed are out of 2000, the fact that all solutions produced satisfied above 1800 clauses does not indicate that the algorithms all performed well. Random assignments of boolean variables were shown to satisfy an average of 1750 clauses. This is due to the nature of the problem: each clause can be satisfied in one of three ways, and while this makes it trivial to satisfy most of the clauses, satisfying all or close to all of them requires satisfying each clause ways which do not conflict with each other. Thus the amount of clauses an individual satisfies, their fitness score, is not linearly correlated with the actual performance of individuals on the SAT problem.

Statistical analysis of the data through the use of two-sample t-tests for sample means ($\alpha = 0.05$) shows that TGP and SGP perform similarly to each other, as do LGP and CGP. However, there is a statistically significant difference

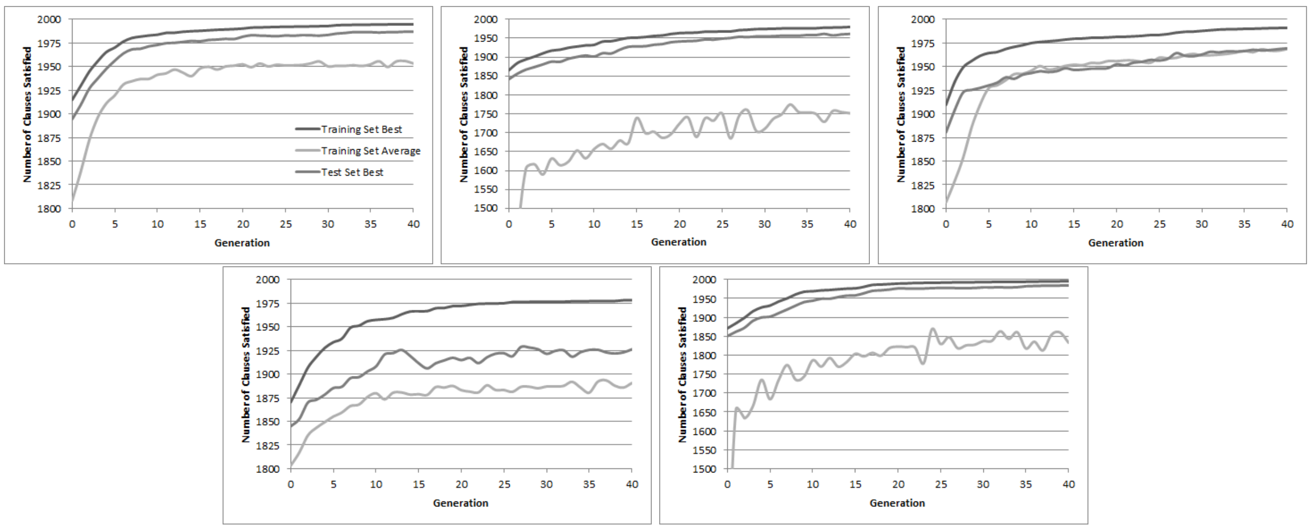

Figure 2: Fitness performance of algorithms over time, averaged across all runs. Top, from left: Tree, Linear, Cartesian. Bottom, from left: Grammatical, Stack. "Training Set Best" describes the fitness of the highest-scoring individual from that generation on the training set. "Training Set Average" describes the average fitness of all individuals from that generation on the training set. "Test Set Best" describes the average number of clauses satisfied on the test set by the individual from that generation that had the highest fitness on the training set.

between these two pairs as well as between each of them and GE. Thus, the choice of GP type can make a real difference in the performance of a hyper-heuristic for the SAT problem. TGP and SGP give the best results, followed by LGP and CGP, followed by GE.

5. CONCLUSIONS

TGP and SGP performed very similarly to each other as expected. This is because SGP generally functions as a linearly stored postorder tree, and thus with the exception of SGP'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 LGP and CGP also have similar search spaces and performed similarly as expected. The differences between TGP and both LGP and CGP's graph search spaces appear to be largely responsible for the performance disparity. None of the solutions generated by the hyper-heuristics make heavy use of 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. GE 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 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 their problem in order to get the best results.

6. FUTURE WORK

While the purpose of this study was to determine whether there exist problems which exhibit a significant hyper-heuristic performance differential between different types of GP – which was shown to be the case – a perhaps more interesting question is whether this differential still holds when all GP types are highly tuned, which is the next logical step in the line of inquiry started in this paper.

Also, while this study was not concerned with the relative performance of hyper-heuristic generated SAT solvers compared to existing SAT solvers, this is an obvious next question to ask. Therefore, another logical next step is to compare the performance of hyper-heuristics employing highly tuned versions of all five GP types reported here, with other hyper-heuristic generated SAT solvers [22, 23] as well as the state-of-the-art in SAT solvers.

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 may be expected to 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.

7. ACKNOWLEDGMENTS

We gratefully acknowledge the advice and support of our Natural Computation Lab colleagues 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.

8. REFERENCES

[1] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the First International Conference on Genetic Algorithms*, pages 183–187, 1985.

[2] John R Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *IJCAI*, pages 768–774. Citeseer, 1989.

[3] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

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

[5] John R Koza and James P Rice. *Genetic programming II: automatic discovery of reusable programs*, volume 40. MIT press Cambridge, 1994.

[6] Nicholas Freitag McPhee and Nicholas J Hopper. Analysis of genetic diversity through population history. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1112–1120. Citeseer, 1999.

[7] Franz Rothlauf and Marie Oetzel. *On the locality of grammatical evolution.* Springer, 2006.

[8] Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. *Advances in genetic programming*, 1:311–331, 1994.

[9] 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.

[10] Conor Ryan, JJ Collins, and Michael O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Genetic Programming*, pages 83–96. Springer, 1998.

[11] Markus Brameier and Wolfgang Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *Evolutionary Computation, IEEE Transactions on*, 5(1):17–26, 2001.

[12] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. *Advances in genetic programming*, 2:111–134, 1995.

[13] James R Levenick. Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In *Proceedings of the Fourth International Conference on Genetic Algorithms.* Citeseer, 1991.

[14] Julian F Miller and Peter Thomson. Cartesian genetic programming. In *Genetic Programming*, pages 121–132. Springer, 2000.

[15] Julian F Miller, Peter Thomson, and Terence Fogarty. Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study, 1997.

[16] Julian F Miller. *Cartesian genetic programming.* Springer, 2011.

[17] Brian W Goldman and William F Punch. *Reducing wasted evaluations in cartesian genetic programming.* Springer, 2013.

[18] Adam Nohejl. *Grammar-based genetic programming.* PhD thesis, MSc Thesis, Charles University of Prague, 2011.

[19] Lee Spector, Jon Klein, and Maarten Keijzer. The push3 execution stack and the evolution of control. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1689–1696. ACM, 2005.

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

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

[22] 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.

[23] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *Proceedings of the 17th European Conference on Genetic Programming.*

[24] 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.

[25] Robert E Keller and Riccardo Poli. Linear genetic programming of parsimonious metaheuristics. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 4508–4515. IEEE, 2007.

[26] Nuno Lourenço, Francisco Pereira, and Ernesto Costa. Evolving evolutionary algorithms. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 51–58. ACM, 2012.

[27] Mihai Oltean and Crina Grosan. A Comparison of Several Linear Genetic Programming Techniques. *Complex Systems*, 14(4):285–313, 2004.

[28] 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.

[29] 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.