Analysis of Cartesian Genetic Programming's Evolutionary Mechanisms

Brian W. Goldman and William F. Punch

***Abstract*—Understanding how search operators interact with solution representation is a critical step to improving search. In Cartesian genetic programming (CGP), and genetic programming (GP) in general, the complex genotype to phenotype map makes achieving this understanding a challenge. By examining aspects such as tuned parameter values, the search quality of CGP variants at different problem difficulties, node behavior, and offspring replacement properties we seek to better understand the characteristics of CGP search. Our focus is two-fold: creating methods to prevent wasted CGP evaluations (skip, accumulate, and single) and creating methods to overcome CGPs search limitations imposed by genome ordering (reorder and DAG). Our results on Boolean problems show that CGP evolves genomes that are highly inactive, very redundant, and full of seemingly useless constants. On some tested problems we found that less than 1% of the genome was actually required to encode the evolved solution. Furthermore, traditional CGP ordering results in large portions of the genome that are never used by any ancestor of the evolved solution. Reorder and DAG allow evolution to utilize the entire genome. More generally, our results suggest that skip-reorder and single-reorder are most likely to solve hard problems using the least number of evaluations and the least amount of time while better avoiding degenerate behavior.**

***Index Terms*—Analysis, Cartesian genetic programming.**

I. Introduction

WHILE effective evolutionary methods are often simple to design and implement, discerning the reason that system leads to effective search can be challenging. This is exceptionally true for genetic programming (GP), which often employs both nontrivial genotype to phenotype maps and operators that act on genotypes without regard to phenotypic impact. Yet understanding the root causes of evolutionary success and failure is critical to improving existing techniques, designing new techniques, and understanding how and where to apply each optimization system.

There have been a number of previous studies into various aspects of GP evolution. For instance, [1] showed how a tree's shape effects its evolvability regardless of phenotype, resulting in the use of new grammar-based operations [2]. Analysis of the root cause of bloat has helped to inform methods for controlling bloat [3].

Manuscript received July 24, 2013; revised December 6, 2013 and April 4, 2014; accepted May 6, 2014. Date of publication May 14, 2014; date of current version May 27, 2015.

The authors are with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824-1226 USA (e-mail: brianwgoldman@acm.org; punch@msu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TEVC.2014.2324539

In this paper, we set out to develop a deeper understanding of what makes Cartesian genetic programming (CGP) an effective evolutionary optimizer. Previous studies of CGPs evolutionary mechanisms have focused on bloat [4], neutrality [5], and structural bias [6]. We shall focus on the interaction between mutation and genome ordering, both generally and how they specifically apply to the Boolean problem domain. While these features of CGP have trivial definitions, we will argue, using experimentation and analysis of alternatives, that they have a great deal of impact on search success.

II. Cartesian Genetic Programming

CGP was originally proposed as a method for general GP in [7]. While its name comes from its original application evolving circuits on a 2-D grid, modern CGP can represent any directed acyclic graph (DAG), and has been utilized in applications such as binary circuits [8], robot controllers [9], neural networks [10], image classifiers [11], and regression [12].

CGP represents DAGs using a linear genome of integer values. Each node in the DAG is encoded as a tuple of genes, with one gene specifying the function the node applies to its inputs, and the remaining genes expressing where the node takes input from. Nodes can take input from either a problem input or any node preceding them in the linear genome. Restricting connections in this way prevents the creation of cycles, while still allowing CGP to reuse values. This is in contrast to tree-based GP, which must duplicate functionality everywhere the same value is needed. To complete the representation, a set of extra genes are included at the end of the genome to specify which nodes or input locations to use as function outputs.

As both output locations and information flow in the DAG are evolvable, often only a tiny fraction of the genome participates in creating the output values. These nodes are referred to as "active," with the nodes not being used to create output values referred to as "inactive." Inactive nodes allow for genetic drift, as individuals can be mutated without effecting their fitness. These mutations can then be incorporated, as future mutations can change the DAG structure causing previously inactive nodes to become active. Previous work suggests CGP is most efficient when up to 95% of the genome is inactive [13], while more recent work suggests this may be a result of hidden parsimony pressure in CGP [14].

CGP uses very simple evolutionary mechanisms. The most common evolutionary strategy is $\mu + \lambda$ where $\mu \leftarrow 1$ and $\lambda \leftarrow 4$. This means during each generation a single parent produces four offspring using mutation. The best offspring then

competes with the parent,[1] with the offspring replacing the parent if it is no less fit. This replacement strategy encourages neutral drift. In CGP mutation is done in a myriad of nearly synonymous ways. Here we mutate each gene of each node at a set probability, where mutation involves changing a gene randomly to some different valid value. For example, if a function gene is chosen for mutation, its new value is randomly chosen from all possible functions, excluding the gene's current value. This version of mutation was chosen to allow precise prediction of changes in gene value necessarily in later sections. Combined, this form of CGP can be viewed as a stochastic hill climber with neutrality. For a more in depth description of CGP, see [15].

III. Duplicate Evaluation Avoidance

CGPs mutation operator is traditionally applied to both active and inactive genes uniformly. This has the potential to create offspring who's only mutated genes are in inactive nodes. These offspring are actively identical (contain identical active genes) to their parents, and therefore by definition have identical fitness to their parents. As such, reevaluating these individuals is computationally wasteful as their fitness is known. Detecting duplicated individuals can be done without significant overhead since finding the set of active nodes is already done prior to evaluation [16]. Our previous work [17] has taken an initial look at the effect this waste can have, and proposed skip, accumulate, and single as methods for avoiding or preventing these duplicate evaluations.

A. Skip

The skip method for avoiding duplicate evaluations involves the least amount of modification from CGPs normal behavior. After an offspring is produced and its set of active nodes is determined but before it is evaluated, it is compared with its parent. Each gene in each active node is compared for equivalence with the corresponding gene in the offspring's parent. If all active genes are found to be equal then the offspring is not evaluated and is instead given the same fitness as its parent. Note that since inactive genes are not compared for equivalence it is still possible for offspring to genetically differ from their parents.

Because skip does not modify any evolutionary mechanisms found in traditional CGP, it will produce identical individuals, with a potential reduction in evaluations. If the mutation probability is high enough relative to the number of active genes, skip will require the same number of evaluations as traditional CGP, since the probability of an offspring being actively identical to its parent is effectively zero. When the mutation probability is low enough relative to the number of active genes, some number of offspring will be actively identical to their parents, resulting in a reduction in evaluations but no change in the evolutionary trajectory.

This technique incurs a minor increase in runtime, as each active gene in the offspring must be compared against the parent's gene at the same locus. This operation is at most linear in genome size. Yet on problems where evaluation is even somewhat expensive, the evaluations prevented by performing this check will result in a net reduction in runtime.

We would expect both intuitively and from our initial experimentation [17] that skip will be less sensitive to the mutation probability than normal CGP. We would also expect skip to use a lower mutation probability than normal CGP when both are optimized to reduce evaluations. The former comes from the fact that in normal CGP any mutation probability that has a significant chance of creating offspring actively identical to their parents wastes evaluations. In skip there are only two penalties for reducing the mutation probability. From an evolutionary standpoint, if search becomes trapped in a local optima, low mutation probabilities may have increased difficulty escaping to find the global optimum. From a performance standpoint, exceptionally low mutation probabilities may spend a prohibitive amount of time attempting to produce an evaluable offspring. Note that these penalties also exist in normal CGP.

B. Accumulate

Similar to skip, accumulate works by adding a step between offspring creation and offspring evaluation. Instead of skipping evaluations when offspring are determined to be actively identical to their parents, accumulate enters into a cycle of repeated mutation until an individual worth evaluating is created. To understand this technique, consider an example where the parent P produces the offspring F_0 using mutation. If F_0 is actively identical to P, F_0 creates its own offspring F_1 using mutation. This process of F_i producing F_{i+1} continues until F_n is produced, where F_n is not actively identical to P. F_n is then evaluated. If F_n is no worse than P, it replaces F_0, and evolution continues as though P had produced F_n directly. If F_n is worse than P, F_{n-1} replaces F_0, as F_{n-1} has accumulated the most mutations to inactive genes without reducing fitness. Note that the mutation used each time is identical to normal CGPs mutation, such that F_n will have one or more active genes different from P. Furthermore, even though this technique produces $n+1$ individuals, only F_n is evaluated. Viewed in another way, accumulate performs a micro evolution on each offspring before it is evaluated, with an emphasis on encouraging neutral drift.

Previous experimentation has shown that accumulate acts very similarly to skip [17], with the exception that accumulate favors lower mutation probabilities when tuned to reduce evaluations to success. While at first these algorithms may appear quite different, further consideration shows how they are similar. In skip any actively identical offspring that is produced is likely to be selected as we expect mutations to active genes to more often reduce fitness than improve it. In the next generation this actively identical offspring is then mutated again, with its lineage likely continuing until it finally does mutate one or more active genes. At this point, if the mutant is better it replaces the parent, otherwise the parent, which has been accumulating mutations to inactive genes, is kept. In this way skip mirrors accumulate, except skip acts over multiple generations instead of compressed into a single generation.

[1] Ties are broken randomly.

Similar to skip, accumulate incurs a minor computation cost in order to prevent wasting evaluations. Generating each F_i requires at most $O(AN)$, where A is the arity of the nodes and N is the number of nodes. While accumulate may produce many F_i in the process of creating a single offspring, conceptually skip using the same mutation rate will create a similar number of total individuals when performing the same number of evaluations. As such, we suggest accumulate will take no more than a constant factor more runtime to produce the same number of evaluable individuals as skip. As before, this increase is likely eclipsed when solving problems with sufficiently time consuming evaluation functions.

C. Single

While skip and accumulate save evaluations by not evaluating offspring which are actively identical to their parents, single changes mutation to ensure that only evaluable offspring are created. Instead of mutating each gene at a set probability, single mutates genes at random until exactly one active gene is mutated.

This modification gives single three properties distinct from the other forms of duplicate avoidance. First, single forces offspring to have an active gene which differs from their parent. This limits drift as the mutated active gene must either be to an intron or represent another way to code a solution of equal quality. Second, as a benefit of forced changes, single avoids the overhead of repeatedly generating individuals without creating an evaluable offspring. Third, single does not require a mutation parameter, effectively setting the mutation probability to $1/a$ for active genes and $1/(a+1)$ for inactive genes, where a is the number of active genes.

In encodings without inactive genes, limiting mutation to changing exactly one gene could prevent an algorithm from escaping some types of local optima. Yet because CGP allows for inactive genes, single is still able to escape most local optima using sufficient drift of inactive genes. When a high percentage of the genome becomes active or when single is otherwise limited in its ability to drift due to lack of introns, it does have an increased potential to become stuck. As will be discussed in Section IV and as shown in [14] this problem of highly active genomes is very unlikely.

Conceptually this method could be extended to mutate some number of active genes before terminating. Doing so would reintroduce a parameter to govern how many active genes to mutate. Our choice of a single mutation allows for the minimal amount of change to ensure the individual should be evaluated, while mutations made to inactive genes still give CGP sufficient power in each mutation to escape local optimums.

IV. Genome Ordering

As was discussed in Section II, CGP uses node ordering in the genome to prevent cycles, by ensuring nodes only receive input from sources that precede them in the genome. This restriction does not limit CGPs ability to represent DAGs, as all DAGs can be serialized to fit this requirement. However, this representation is likely having an impact on CGPs ability to evolve specific DAGs [14].

Primarily, enforcing node ordering adds artificial limitations to CGPs ability to connect nodes. Nodes which can be connected without creating a cycle may still be prevented from forming that connection because of random genome ordering. Similarly, it is impossible for adjacent nodes to ever be connected through an intermediate node. As a result, useful structures in the genome may need to be evolved repeatedly in order to find the best location for that structure in the genome.

Ordering, when combined with the mutation's uniform resetting of connection genes, may be the cause of CGPs immensely inactive genomes. As was rigorously defined in this paper [14], the number of active genes in a genome is expected to scale logarithmically with genome size, independent of problem application. Furthermore, the less active a genome, the lower the likelihood of structure ordering preventing structure connection.

As an extension to the theory presented in [14], we have derived the formula which predicts the probability a node is active based on its index, given in

$$p(i) \leftarrow 1 - \left(\frac{N+I-1}{N+I}\right)^{O} \cdot \prod_{j=i+1}^{N} 1 - p(j) \cdot \left(1 - \left(\frac{j+I-1}{j+I}\right)^{A}\right). \quad (1)$$

In this equation, i is the index of the working node, N is the genome size, I is the number of input locations, O is the number of output locations, and A is the arity of each node. This equation is constructed as a negation, such that we determine the probability the node is not active due to each potential source that could make it active. The product term calculates the probability that the node at index j is connected to the node at index i, multiplied by the probability that the node at index j is active. This is negated to determine the probability node i is not active because of node j. All together, the product term calculates the probability that no active node is connected to the node at index i. Nodes can also become active if they are directly connected to by an output location. As such the product is scaled by the probability that all output locations connect to inputs or nodes not at index i. This formula assumes only operator bias, not selective pressure, is acting on the genome.

Substituting $N = 2000$, $A = 2$, $I = 6$, $O = 6$ into (1) (chosen to match experimental values used in Section VI) predicts only 160 active nodes, or 8% of the total genome. Positionally, 80% of the first 100 nodes are expected to be active while just 4% of the remaining 1900 nodes are expected to be active.

To rectify these issues and examine CGP without ordering bias, [14] created reorder and DAG as methods for genome ordering. For clarity, we shall refer to CGPs historical ordering method of enforcing only forward connections as normal.

It is important to note that (1) does not apply to all variants of CGP. For instance, the early versions of CGP [15] which utilize a levels back parameter L will have a flat active probability for all nodes at least L nodes from the start and L nodes from the end of the genome. Yet this does not remove the issue of artificially limiting connections, and in fact adds

```
procedure RANDOMSERIALIZATION
    addable ← input_locations
    index ← |input_locations|
    new_loc ← ∅
    while |addable| > 0 do
        working ← randpop(addable)
        if working ∉ input_locations then
            new_loc[working] ← index
            index ← index + 1
        end if
        for all link ∈ feeds_to(working) do
            Satisfy link's dependence on working
            if |unsatisfied(link)| = 0 then
                addable ← addable ∪ {link}
            end if
        end for
    end while
    return new_loc
end procedure
```

Fig. 1. Algorithm which converts any DAG into a random serial ordering such that all nodes take input from nodes that precede them in the genome. Used by reorder to determine new node locations and DAG to determine node evaluation ordering.

many new limitations. More recently, [18] proposed a method which reassembled the genome using semantic cues, which circumvents the forward only limitation in normal.

A. Reorder

The reorder method shuffles an existing genome's node ordering without changing node behavior. This is possible because for a given node, there can be a large number of other nodes which have no required ordering, as they neither take input (directly or indirectly) from the node nor provide their output (directly or indirectly) to the node. In general, this process works by assigning nodes new locations in the genome at random once all of the nodes they take input from have been assigned locations earlier in the genome.

The first step in performing reorder on a genome is to create data structures to store direct connection relationships. These structures are used to determine in constant time all connections incident on a node, and can be built in order $O(AN)$ time, where A is the arity of the nodes and N is the number of nodes.

With this preprocessing complete, the RANDOMSERIALIZATION algorithm given in Fig. 1 is used to assign each node a new location in the modified genome. It starts by constructing the addable set, which contains all nodes who's direct dependencies have already been added. Initially this contains only the input locations. Once a node is assigned a location, all of the nodes that depend on its output have that dependency marked as satisfied. Once all of a node's dependencies are satisfied, it can be put into addable. In this way, nodes are randomly removed from addable, assigned the next location in the genome, with new nodes added to addable as they becomes semantically viable. Iteration ends when all nodes have been assigned new locations. RANDOMSERIALIZATION requires $O(AN)$, as there can be only N iterations, with each iteration updating a distinct subset of the AN dependencies.

The final step in reorder is to use the list of new locations to convert the existing genome into the reordered genome. This can be done by converting connection genes and output location genes using the *new_loc* map returned from RANDOMSERIALIZATION. This also requires $O(AN)$ time, as all connection genes in all nodes must be converted. As this is the final step unique to reorder, and no previous steps are of higher complexity, it is also the complexity of the reorder algorithm as a whole. Furthermore, it is of the same complexity class as copying the entire genome, meaning it does not change CGPs overall complexity.

Reorder is used once each generation to shuffle the nodes of the parent's genome. As the shuffling does not semantically change the parent, it does not require reevaluation. Furthermore, as shuffling does not change the requirement that nodes only depend on those preceding them in the genome, shuffling does not require any changes to any other CGP methods, such as evaluation and mutation. It is important to note, however, that the potential mutations that can be applied to the parent have changed. Consider two nodes X and Y which are in the genome, and have no dependent relationship. As X does not take input from Y and Y does not take input from X, either can be placed before the other in the genome. Without shuffling, the initial ordering of nodes is fixed. No operation in normal is capable of making X take input from Y. Reorder's shuffling can allow subsequent mutations to make the connection.

Reorder has the potential to reduce node reinvention. In normal, if X preceded Y, mutation would have to recreate either X or Y in new locations in order to make the required connection. Reorder provides the opportunity to examine different orderings of the genome without such costs.

The price reorder pays for this potential improvement is a larger mutational search space. Though a shuffled individual is phenotypically unchanged, its mutational adjacency can be significantly different. As such the variety of offspring an individual can create is increased when using reorder. Therefore creating a specific offspring that improves upon the parent is less likely, in that the number of alternatives is increased. Yet many of these additional alternatives may also be improvements upon the parent. This may increase the likelihood of producing an offspring that improves upon the parent, allowing reorder to overcome this drawback.

B. DAG

Normal and reorder use node location in a linear genome to create an efficient protection against cycles: nodes must connect forward. DAG removes this restriction and allows forward and backward connections, relying on other, more complex, methodologies to prevent cycles. To accomplish this goal, modifications must be made to how CGP performs mutation and evaluation.

As in normal and reorder, when a connection gene is chosen for mutation, its value is chosen randomly. However, the

```
procedure ISDEPENDENT(working, known)
    if working ∈ known then
        return known[working]
    end if
    for all link ∈ reads_from(working) do
        if ISDEPENDENT(link, known) then
            known[working] ← True
            return True
        end if
    end for
    known[working] ← False
    return False
end procedure
```

Fig. 2. Algorithm used by DAG to determine valid connection gene values.

options to choose from are only limited to those connections which do not create cycles. To achieve this, we incrementally determine which nodes transitively depend on the mutating node and which do not, with information stored in a known map data structure. This data structure is initialized with the information that the node is dependent on itself and that all input locations are not dependent on the mutating node. Nodes are then tested in a random order for dependence using the ISDEPENDENT algorithm given in Fig. 2. The first node returned at the top level that is not dependent on the mutating node is then used as the new connection gene value. As a result, mutating a connection gene cannot introduce a cycle.

ISDEPENDENT is a nonrepeated recursive depth first search of the individual's DAG. Search terminates as soon as a dependent node is found, and information from previous searches is memorized to prevent repeated search. Each recursion level results in a node being added to known. As such initial calls will likely result in lots of recursion, but subsequent calls will find answers more quickly. In the worst case, this algorithm may need to examine all nodes in the genome to determine if working is dependent on the mutating node, giving ISDEPENDENT a complexity of $O(AN)$. In the worst case, this algorithm may be called on all possible nodes in the genome to check their dependence. Yet cumulatively those calls can only take $O(AN)$ time due to the nonrepeating nature of the search. As a result, the expected worst case run time for DAG mutation is $O(mA^2N^2)$, where m is the mutation probability. We expect mAN connection genes to be mutated, each requiring a worst case of N dependency checks each taking A time. This is in contrast to normal and reorder which require $O(mAN)$ to perform mutation, as individual mutations can be performed in constant time.

The method for evaluating DAG individuals is very similar to the method for efficiently evaluating normal and reorder individuals. As before, preprocessing is done to determine which nodes are active, and only those nodes are executed during evaluation. Also as before, the process begins from the output locations, recursively following connection genes and marking nodes as active until the input locations are reached. At this point, the process changes. In DAG, we must determine not only the set of nodes that are active, but also the order in which those nodes should be executed.

RANDOMSERIALIZATION can be reused to determine the order in which nodes should be executed during evaluation. Instead of using *new_loc* to reorder the genome, we can use this map to specify the order in which nodes should be executed. For instance, if a node was given the new location of X, we know that once all nodes at locations preceding X have been evaluated, the node at X can be evaluated. Stripping the inactive nodes from *new_loc* and inverting the map results in an efficient, valid ordering in which to execute the nodes.

The process of determining which nodes are active requires $O(AN)$ time, regardless of node ordering. As was discussed in Section IV-A, getting the *new_loc* map requires $O(AN)$ time. Finally, converting the map into an ordering of the active nodes takes $O(N)$ time. Each of these steps are sequential, meaning DAG does not increase the runtime complexity of evaluation preprocessing. Furthermore, once the preprocessing step is done, DAG takes identical time to evaluate as the other methods, regardless of the number of input combinations.

With these modifications, there is no longer any meaning to the location of a node in the genome. In normal, the number of valid values for a connection gene is related to the number of nodes preceding that gene, and independent of the number of nodes following that gene. In DAG a connection gene's valid values only depends on the current solution represented by the individual, independent of the genes location in the genome. As such, there are fewer artificial limitations on how DAG can modify solutions.

While reorder and DAG both reduce mutational limitations, DAGs changes are more sweeping. Though both can create any connection that does not form a cycle, reorder maintains a bias toward certain kinds of connections. Consider again two nodes X and Y such that neither is dependent on the other. If X is transitively dependent on very few nodes, and Y is transitively dependent on many nodes, reorder is more likely to shuffle X to precede Y. This is true because at any point during the construction of the new ordering, the probability that all of Xs dependencies have been added is higher than for Ys dependencies. As a result we expect X to be put into addable sooner, and therefore we expect it to be added sooner. The greater the discrepancy in the dependency set sizes the less likely the order between the nodes will be reversed. Conversely, DAG has no such bias. It is always possible to mutate X to connect to Y.

The probability of connecting X to Y is also influenced by how many nodes transitively depend on X and Y. For example, assume X has more nodes transitively dependent on it than Y. In reorder, X is again more likely to be placed before Y, this time because the number of possible nodes preceding it is reduced. DAG is biased in the reverse, such that X will have a higher chance of connecting to Y than Y to X. This is because X can connect to fewer nodes, so each mutation has a higher chance of choosing Y.

V. QUALITATIVE COMPARISONS

All told we have proposed three new methods for avoiding duplicate evaluations (skip, accumulate, and single) and two new methods for genome ordering (reorder and DAG). See

TABLE I
SHORT DESCRIPTIONS OF EACH OF THE ALGORITHM VARIANTS

Duplication	
Skip	Whenever an actively identical offspring is detected, assign it its parent's fitness.
Accumulate	Repeatedly perform full genome mutation until **one or more** active genes change.
Single	Select and mutate individual genes until **exactly one** active gene changes.
Ordering	
Normal	Nodes must take input from other nodes or inputs that precede them in the genome.
Reorder	Each generation the parent's node ordering is shuffled without effecting node behavior.
DAG	Connection genes can mutate to make any connection as long as no cycle is formed.

Table I for a short description each variant. As these two sets are nonoverlapping, and there is no intuitive groupings, we chose to test all possible combinations in our empirical analysis. As a control we included normal genome ordering, as it provides insight to how each of the duplicate prevention techniques work if applied alone. Similar measures are not needed in testing the genome ordering techniques, as skip has no impact on evolutionary search, and is strictly an improvement to using no duplication detection at all.

In total this creates nine algorithm combinations. Of these, our previous work has shown the effectiveness of the duplication methods with normal ordering [17] as well as analysis of all three ordering methods with single [14]. This means there remain four novel combinations to be tested. This is also the first time the ordering methods have been rigorously tested for quality, as much of the work in [14] was analysis oriented.

A. Problem Set

In order to provide test landscapes for evolution, we chose problems common to the CGP literature and previous work with the duplication prevention and ordering techniques. We have chosen four binary circuit problems, as the binary representation allows for some of the precise analysis performed in Section VI. For all problems, we used the function set {AND, OR, NAND, NOR}. To cover a range of different binary problems, each of the four chosen have different numbers of inputs, different number of outputs, and different levels of difficulty.

The first problem, 3-bit parity, represents the most common application for CGP [8], [13], [19]. We include this problem purely for backward comparison, as we agree that in general it is too simple of a problem to merit its own conclusions [20]. Yet as part of the group it may help yield understanding about how each variant performs on simple problems.

As representatives of binary problems with varying input and output sizes, we reuse the 16-4-bit encode and 4-16-bit decode proposed in [17]. In these problems, the circuit evolved must either convert a 4-bit encoded integer to a 1 on the corresponding output line (decode), or take a 1 on one of the 16 input lines and convert it back into a 4-bit encoded integer (encode). These problems share properties with the commonly used multiplexer problem, but include multiple output lines, and can be evaluated quicker as each only requires 16 possible test points.

Finally we chose the 3-bit multiply to represent hard binary problems, as was suggested in [20] and used by [5], [8], and [13]. This problem is very difficult by comparison to the other problems, and has the largest number of test points.

B. Parameter Setting

To perform a fair comparison, we ensure proper parameter configuration to avoid potential bias that might benefit a specific method. While tuning can lead to the alternative issue of methods only being effective after extensive problem specific tuning, we set out to use rough-grained values and provide each technique with equal tuning time to alleviate that bias.

We focus on two parameter values in our configuration: mutation probability and genome size. While there are other potential configurable parameters (population size, offspring size, etc.), we feel the CGP literature has converged on settings for those parameters (1, 4). Furthermore, as duplication detection explicitly deals with the relationship between mutation probability and number of active genes, and the number of active genes depends on the genome size and ordering method, mutation probability and genome size seem to be the most likely to impact results.

To set parameter values, we started by defining a grid of parameter values: 50, 100, 200, 500, 1000, 2000, 5000, 10 000 for genome size and 0.05, 0.02, 0.01, 0.005, 0.002, 0.001, 0.0005, 0.0002 for mutation probability. Note that here a genome size of 50 means there are 50 nodes, each composed of multiple genes, plus the required genes to specify output locations. Also note that a mutation probability of 0.05 is used here to mean each gene has a 5% probability of being mutated in a single application of mutation. These ranges were chosen to completely cover the range of previously used parameter settings. The grid is formed by trying all possible mutation probabilities with all possible genome sizes, resulting in 64 potential configurations for each of the algorithm combinations. Finally note that as single does not use a mutation probability, only the eight configurations with different genome sizes were used when single was employed.

To choose which parameter configuration to use with each algorithm combination for each problem, we used an iterative process of performing runs, comparing medians, removing configurations, and repeating. For each algorithm combination and problem, we first ran all potential parameter configurations five times (320 runs if not using single, 40 otherwise). The parameter configurations are then sorted based on their median. The best half are then each run four more times, with the best half of that group chosen. In this way, 32 configurations are run five times, 16 are run nine times, 8 are run 13 times, 4 are run 17 times, 2 are run 21 times, and 2 are run 25 times. For algorithm combinations using single, all eight parameter configurations were run 17 times before any were

removed. This iterative reduction allows us to focus tuning on those parameter configurations most likely to be effective, with more information gathered before choosing between high quality configurations. We chose to run each initially five times to reduce the chance of eliminating a configuration due to outliers. We incrementally added four to provide enough information to significantly update estimates, while keeping the number of runs odd, allowing the median to be a representative run and not an interpolation. As a final consideration the number of experiments were set to make the total number of runs feasible. In total this means 15 696 complete runs are required to set the parameter configuration for all algorithm configurations for all problems.

To ensure termination, all runs were limited to 10 000 000 evaluations, well beyond the expected time to completion. As the median was used to compare configurations, selection should be insensitive to this limit unless a large portion of runs failed to optimize.

To prevent undue biasing in the final results, these tuning runs were not used in the subsequent data analysis. Effectively, the runs used to set the parameter configuration were considered the training runs for that configuration. As a result, our final results were gathered from a completely independent set of runs to test the configuration's general characteristics. Each configuration was run 50 times to ensure statistical power. All told over 11.5 billion evaluations were performed to gather our results. The source code used to produce our results is available from our website.[2]

C. Results

The results from running each tuned parameter configuration for each algorithm combination on each problem are summarized by Table II. Rows for each problem are ordered by that algorithm combination's median evaluations to success (MES), with the best algorithm combination appearing at the top of each section. The highlighted rows mark skip-normal, the control algorithm combination representing traditional CGP performance. All configurations for each problem were then tested using Kruskal–Wallis[3] to determine if any configuration was statistically different from any other. All problems were determined to be highly significant. As such, each algorithm combination was then compared with the control to determine statistical significance using pairwise Mann–Whitney U[4] tests. The resulting p-value for each test is reported in the table.

As discussed in Section V-B, the genome size and mutation probability for each algorithm combination was extensively optimized. In line with previous research [13], this led to massive genome sizes for almost all algorithm combinations on all problems. With the exception of some combinations using single, all algorithm combinations used over 1000 nodes. Half of all combinations without single used 10 000 nodes. As this was the maximum allowed by our tuning, these combinations may work better with even larger genomes.

Of the 12 configurations including single, 11 used a genome size smaller than all configurations not using single. The exception was single-normal on multiply. No other algorithm had as much impact on genome size tuning as the inclusion or exclusion of single. In our previous work [14] it appeared that reorder and DAG allowed CGP to use smaller genomes. In light of our current results, this conclusion should be amended. It appears that single is most effective when the genome size is small and that reorder and DAG are best able to cope with smaller genome sizes. Independent of duplication method, the ordering methods do not seem to create any pressure on which genome size is chosen.

Unlike genome size, all configurations choose approximately the same mutation probability for each problem. With the exception of parity, there appears to be a general consensus that each gene should be mutated with probability 0.002, independent of genome size. Parity used a higher mutation probability, but this is likely because 3-bit Parity is by far the easiest problem tested and therefore likely benefits from higher exploration.

On many of the problems, many of the configurations have little to no statistical difference from the control in their evaluations required to reach the global optimum. On parity and encode it is probably safe to conclude that single is worse than the control. As these are the easiest problems tested, that difference may have little real world meaning. DAG is almost certainly worse than the control on decode. Its results are so much worse that we can also likely conclude that DAG is worse than all non DAG configuration on that problem, even though no statistical tests are presented here. The control on multiply did by far the worst of all configurations, with only accumulate normal likely failing statistical significance.

By ordering the algorithm combinations by MES and examining that ordering across each problem, we can look for general trends. From the table, it appears that on easier problems the method of duplication used is the strongest predictor of rank, while the ordering methods are more predictive on harder problems. This comes from the observation that in parity and encode (upper half of Table II) the first column is effectively sorted, while for decode and multiply (lower half of Table II) the second column is sorted. Each table includes coincidentally one exception to these rules. On parity and encode, pairings with DAG are ranked lower than other ordering methods, so much so as to break the ordering once on both problems. On decode accumulate-reorder ranks worse than expected. Finally on multiply single-DAG does far better than expected.

On almost every problem, DAG required more evaluations to success than reorder, and it only outperformed normal on multiply. Combined with its's algorithmic overhead described in Section IV-B, DAG is therefore not likely to be useful for anything but comparison purposes.

On all four problems, reorder finds the solution in either comparable or significantly less evaluations than normal ordering, independent of the duplicate method in use. Combined

[2]https://github.com/brianwgoldman/Analysis-of-CGPs-Mechanisms

[3]Kruskal-Wallis, a nonparametric equivalent of ANOVA, determines if samples were drawn from the same distribution. If this test is significant, at least one of the samples differs from the others.

[4]Mann–Whitney U, a nonparametric equivalent of student's t-test, determines if two populations were drawn from the same distribution. If this test is significant, the populations are not equal.

TABLE II
RESULTS FOR ALGORITHM CONFIGURATION ON EACH PROBLEM. HIGHLIGHTED ROW IS THE CONTROL CONFIGURATION SKIP-NORMAL. MES = MEDIAN EVALUATIONS TO SUCCESS; CONFIDENCE INTERVAL = 95% BOOTSTRAPPED CONFIDENCE INTERVAL; ACTIVE = MEDIAN NUMBER OF ACTIVE NODES IN EVOLVED SOLUTION; USED = MEDIAN NUMBER OF NODES ACTIVE IN EVOLVED SOLUTION AFTER APPLYING SIMPLIFY (SEE SECTION VI-B); p-VALUE = RESULT FROM MANN–WHITNEY U COMPARISON WITH CONTROL

3-bit Parity, Kruskal-Wallis=$5.99 \cdot 10^{-14}$								
Duplication	**Ordering**	**Genome Size**	**Mutation Probability**	**MES**	**Confidence Interval**	**Active**	**Used**	**p-value**
Accumulate	*Normal*	10,000	0.02	682	445 .. 869	122	20	0.5602
Accumulate	*Reorder*	5,000	0.01	796	576 .. 1,019	331	23	0.7174
Skip	*Normal*	10,000	0.02	823	662 .. 1,034	119	20	NA
Skip	*Reorder*	2,000	0.01	870	708 .. 1,070	198	20	0.3125
Accumulate	*Dag*	5,000	0.01	964	759 .. 1,250	1,174	22	0.3310
Skip	*Dag*	10,000	0.02	1,042	734 .. 1,339	2,083	24	0.0248
Single	*Normal*	500	NA	1,214	885 .. 1,487	39	17	0.0046
Single	*Reorder*	200	NA	1,506	552 .. 2,122	57	18	0
Single	*Dag*	100	NA	2,569	2,109 .. 3,358	43	15	0
16-4-bit Encode, Kruskal-Wallis=$1.19 \cdot 10^{-6}$								
Duplication	**Ordering**	**Genome Size**	**Mutation Probability**	**MES**	**Confidence Interval**	**Active**	**Used**	**p-value**
Skip	*Normal*	10,000	0.005	16,226	9,315 .. 22,229	283	49	NA
Skip	*Reorder*	10,000	0.005	17,883	10,690 .. 23,999	2,070	62	0.6918
Accumulate	*Reorder*	10,000	0.002	18,132	13,646 .. 22,983	2,027	64	0.8822
Accumulate	*Normal*	10,000	0.005	21,428	17,536 .. 25,948	279	49	0.2525
Accumulate	*Dag*	10,000	0.002	24,855	18,222 .. 33,069	4,059	69	0.0736
Skip	*Dag*	10,000	0.002	25,678	16,170 .. 33,422	4,178	62	0.0211
Single	*Reorder*	100	NA	26,558	20,678 .. 31,613	52	32	0.0076
Single	*Normal*	2,000	NA	31,057	24,145 .. 42,278	141	43	0.0085
Single	*Dag*	100	NA	34,208	29,840 .. 38,035	59	34	0
4-16-bit Decode, Kruskal-Wallis=$1.02 \cdot 10^{-36}$								
Duplication	**Ordering**	**Genome Size**	**Mutation Probability**	**MES**	**Confidence Interval**	**Active**	**Used**	**p-value**
Skip	*Reorder*	2,000	0.002	62,853	50,033 .. 72,267	757	114	0.5247
Single	*Reorder*	500	NA	63,731	56,070 .. 70,706	245	97	0.3610
Accumulate	*Normal*	1,000	0.002	67,390	60,738 .. 74,624	211	87	0.8767
Skip	*Normal*	1,000	0.002	68,231	63,266 .. 74,463	202	85	NA
Single	*Normal*	1,000	NA	68,819	62,940 .. 74,939	203	85	0.7695
Accumulate	*Reorder*	2,000	0.002	72,991	67,163 .. 78,131	760	115	0.1868
Skip	*Dag*	10,000	0.001	132,821	117,034 .. 155,471	4,983	125	0
Accumulate	*Dag*	5,000	0.002	133,188	121,369 .. 147,073	2,542	125	0
Single	*Dag*	500	NA	160,696	142,001 .. 160,696	299	97	0
3-bit Multiply, Kruskal-Wallis=$5.14 \cdot 10^{-18}$								
Duplication	**Ordering**	**Genome Size**	**Mutation Probability**	**MES**	**Confidence Interval**	**Active**	**Used**	**p-value**
Skip	*Reorder*	10,000	0.0005	196,116	160,073 .. 223,500	3,175	484	0
Accumulate	*Reorder*	10,000	0.001	199,193	123,496 .. 275,013	3,117	489	0
Single	*Dag*	200	NA	239,844	204,347 .. 286,418	114	74	0
Single	*Reorder*	500	NA	243,664	178,566 .. 301,246	266	131	0
Skip	*Dag*	2,000	0.002	369,931	234,885 .. 483,753	952	149	0.0100
Accumulate	*Dag*	2,000	0.002	379,202	255,165 .. 460,783	973	158	0.0053
Single	*Normal*	5,000	NA	399,911	280,070 .. 516,034	273	112	0.0047
Accumulate	*Normal*	2,000	0.002	483,938	308,049 .. 611,034	168	87	0.45
Skip	*Normal*	2,000	0.002	657,121	469,287 .. 864,073	175	84	NA

with the complexity analysis in Section IV-A which shows it is no more asymptotically complex, we would suggest reorder as an alternative to normal on hard problems.

The difference between using skip and accumulate remains negligible [17], even in combination with different ordering methods. This supports the idea that the two are actually achieving synonymous behavior through different means. As accumulate is far more complex in terms of algorithm function than skip, we would suggest skip be used over accumulate for any future applications.

As a final check that our results do not somehow unfairly handicap traditional CGP, we compared our MES with previous publications. The results reported for traditional CGP in [8] for 3-bit parity required six times as many evaluations as skip-normal. On the 3-bit multiplier problem using a slightly different operator set, the same source had traditional CGP requiring 4 million evaluations to success, as opposed to our skip-normal requiring 600 thousand, also a factor of six improvement. To determine the effect the difference in function set had on results, we ran skip-normal and skip-reorder using their functions but our parameters. The former improved to 453 684 while the latter increased to 303 657, maintaining a statistically significant difference. These results make sense as the operator set in [8] is able to more compactly represent the solution than our set. This puts less strain on normal to deal with complex solutions, while reorder's extremely large genome size is likely made unnecessary. Note that as we did not tune using their operator set, our results should be considered a lower bound on quality for using skip-normal and skip-reorder to solve 3-bit multiplier with their operators.

Our parameter tuning focused on minimizing the number of evaluations each algorithm combination required to solve problems, as that is the primary measure of runtime for CGP and GP in general. As such, comparing the true runtime of the tuned parameters is likely to give potentially misleading results. In using profilers and examining general trends, one of the largest indicators of runtime per evaluation was genome size. This makes sense as each time an offspring is generated all of the parent's genes must first be copied. Furthermore, all genetic operations in CGP scale with genome size. Also, larger genome sizes correlate with more active nodes, each of which must be executed on each input tested during individual evaluation.

With these caveats, the observed runtime for each algorithm combination follows what we would expect given the number of evaluations performed and the genome size. When reorder and normal use the same genome sizes, they require nearly the same amount of time per evaluation, with reorder's added steps and increased number of active nodes constituting a relatively small increase. In general this difference in time per evaluation is trumped by differences in actual number of evaluations, which remains the primary cause of differential total runtime. As predicted, DAG does take longer than the other algorithms to perform each evaluation, reinforcing the conclusion that DAG should not be used for applications.

As our implementation of skip and accumulate required each gene to generate a random number to determine if it should mutate, these methods took significantly longer per evaluation than single. Utilizing a binomial distribution, identical behavior could have been achieved in effectively constant instead of linear time. Further skewing the comparison is that single's tuning resulted in far smaller genomes, giving it a significant edge over the other techniques in runtime per evaluation.

If we had tuned using runtime instead of evaluations, we would expect whichever technique is most effective at small genome sizes to obtain the best results. From theoretical work in [14], we would expect reorder to work better with this limitation than normal. Assuming the tuned values we obtained are any gauge, single-reorder will likely require the least amount of runtime when tuned to reduce runtime.

VI. Analysis of Evolutionary Mechanisms

Comparing how many evaluations each algorithm requires to solve benchmark problems provides very little understanding of why one is more efficient than another. Thus we must turn to a more detailed view to support theorized capabilities of each variant and to discover hidden aspects of CGP. To accomplish this task we have devised novel metrics to expose what makes a CGP run successful.

A. Never Active Nodes

In line with the suggestions made in [13], the evolved number of active nodes across all problems and all algorithm configurations in Table II was significantly lower than the genome size. With the exception of a few combinations with DAG and single, the evolved genomes were over 50% inactive. Skip-normal, our control representing normal CGP, had only 1%, 3%, 20%, and 9% of nodes in the genome active in the median runs of parity, encode, decode, and multiply, respectively. These final results fall in line with the predictions made in [14], in that if the problems are ordered based on their number of output locations, their evolved percentage of active nodes is also ordered. Furthermore, if (1) is used to calculate the expected number of active nodes for configurations using normal ordering, we find that in all cases the evolved number is only marginally higher than the expectation.

With such a small percentage of the final genome active, what is the rest of the genome for? Just because a node is not active in the final solution does not mean it was never useful. For a node to be truly of no use to search, we must consider nodes that were never active. Never active nodes are nodes such that for all ancestors of the final solution (the line of descent), no parent had that node active. Note that for reorder this demarcation of node activity follows the node through each shuffle, such that if a node was marked as never active before the genome was shuffled it is marked as never active after the shuffle, and vice versa. This is in contrast to marking locations as never active, which was not done in this paper.

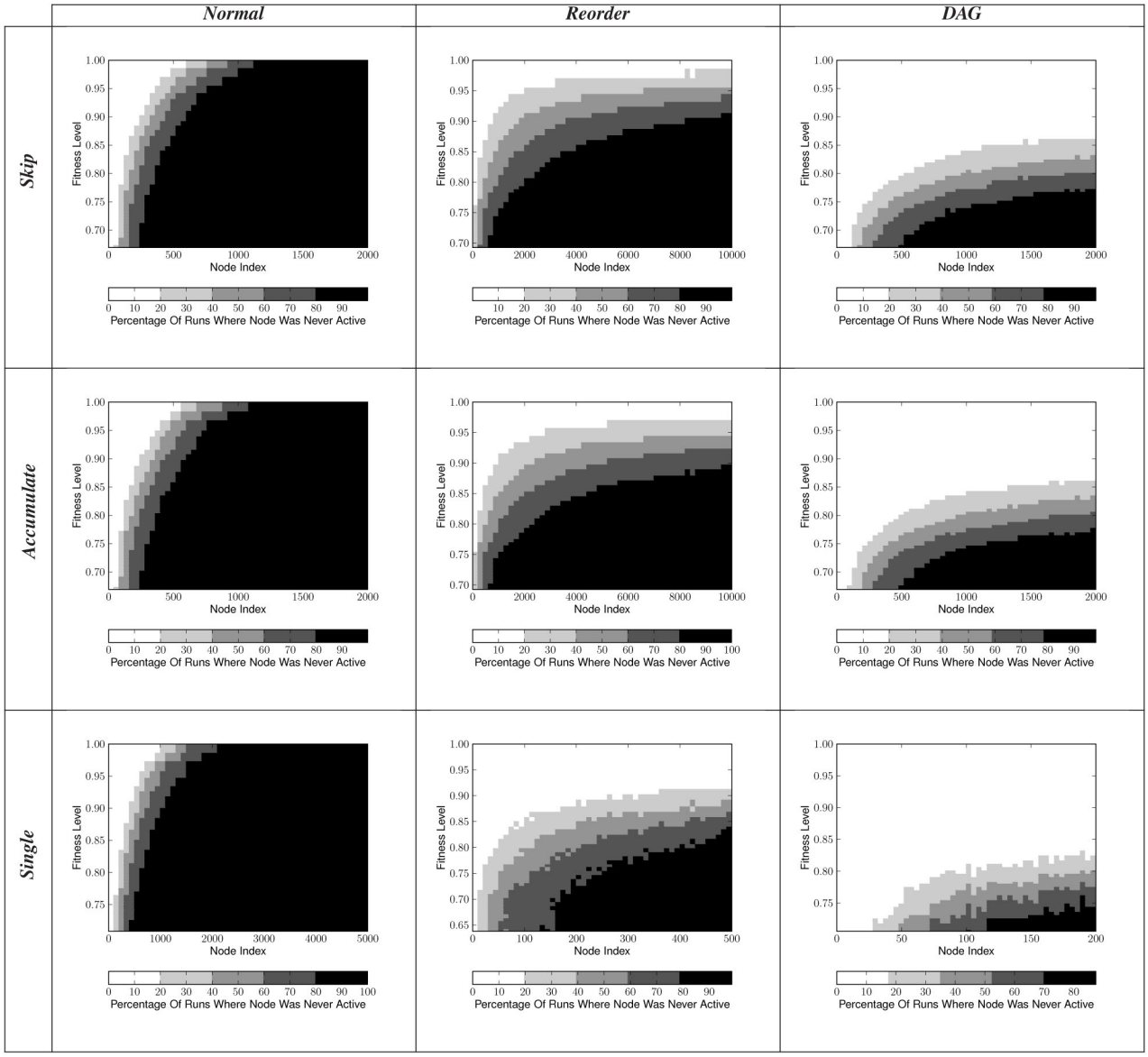
Table III provides information about never active nodes for each algorithm combination on the 3-bit multiply problem. The horizontal axis in each of the nine plots shows different genome locations, while the vertical axis denotes the level of fitness the recorded individual obtained. The shading of each coordinate gives the percentage of 50 runs in which the first individual of the given fitness level had no ancestor in which the node at a given location was active. While only multiply is shown here, all of the other problems tested had similar behavior, with allowances for differences in number of evaluations performed.

Examining the columns of Table III makes it clear that each of the ordering methods had significantly different never active behavior. Looking at normal we see that only at very high fitness levels does evolution begin to activate nodes in the latter half of the genome. Even by the final solution large percentages of the genome were never activated, with skip, accumulate, and single having median percentages of never active nodes of 43.35%, 42.05%, and 56.17%, respectively. This is in stark contrast to reorder and DAG, which never had a median higher than 0.05%.

Compared to the difference in columns, the difference in the rows of Table III caused by the different duplication methods were almost nonexistent. Only single has a distinguishable behavior, but this is almost certainly the result of the genome size and not the duplication method. Single-normal used a genome size 2.5 times larger than normal did with the other techniques, so the resulting increase in never active nodes makes sense. When single was paired with reorder and DAG the genome sizes were far smaller, resulting in noisier graphs with less never active nodes.

In Section IV we used (1) to suggest normal has a strong bias on the location of active nodes regardless of fitness landscape, and that reorder and DAG would likely have less or no bias. The example inputs used in that section match those of

TABLE III
LOCATION OF NEVER ACTIVE NODES ACROSS ALL 50 RUNS FOR ALL NINE ALGORITHM COMBINATIONS ON THE MULTIPLY PROBLEM



normal's tuned configuration on 3-bit multiply. The predicted number of active nodes and their location in the genome match almost exactly the behavior displayed in all three normal plots in Table III.

Examining the progression in fitness across the different ordering methods shows the effect of initialization on each. All three order methods share the same initialization method, and as such all have very similar never active node behavior early in evolution. This initialization similarity explains why DAG has any positional bias at all, even though its evolutionary mechanisms have none. Similarly reorder diverges from normal given sufficient time, eventually losing its positional bias.

This early similarity may be part of the reason normal and reorder have such similar behavior on relatively easy problems. After relatively few evaluations both have explored similar sections of their genome. Yet on hard problems that require further exploration, reorder is able to do so, while normal remains positionally limited.

B. Node Behaviors

In order to understand how each node is being used by CGP, we first determine the semantics of each node [21]. By semantics we mean that for each input to an individual, each node will produce a single output. These output values can be recorded and ordered. As the domain in use is binary, the outputs can be viewed as a bit string, where each bit indicates the value a node outputs when an individual is presented with a specific input. Similar recording can be done for other domains, but binary allows for by far the simplest encoding.

This semantics representation of a node is a complete description of its functionality. Any two nodes with an identical semantic representation have identical behavior, even if

```
procedure SIMPLIFY
    reconnect ← ∅
    for all node ∈ reversed(active) do
        reconnect[semantics(node)] ← node
    end for
    for all i ∈ input_locations do
        reconnect[semantics(i)] ← i
    end for
    for all g ∈ genes do
        if g is a connection gene then
            if semantics(g) ∈ reconnect then
                g ← reconnect[semantics(g)]
            end if
        end if
    end for
end procedure
```

Fig. 3. Algorithm to ensure each active node calculates a unique semantic.

their method for encoding that behavior is drastically different. From this perspective, we can view CGP evolution as attempting to evolve a node who's semantics match the desired output semantics. Doing so allows us to examine how the evolutionary operators change the behavior of nodes, not just their gene values.

Utilizing semantic analysis, we can extract the useful portions of an individual, a subset of the active nodes. Consider two active nodes X and Y with the same semantics such that X proceeds Y in the genome. As CGP is a graph representation, only one of these nodes is actually necessary. Any node that reads the output of Y can instead read the output of X, removing the need to include Y in the solution.

The SIMPLIFY algorithm given in Fig. 3 is designed to efficiently remove redundancy encoded in a genome. This algorithm moves all connections to the first node that produces the desired semantic. After application, and recalculation of which nodes are still active, each active node produces a unique semantic string. Any node that is transitively dependent on its own semantic will always be removed from the active set. In general, if a set of nodes all produce the same semantic, whichever is predicted to have the least transitive dependencies will be kept. Note that this does not guarantee the minimum possible genome, as that would require exponential runtime. The SIMPLIFY algorithm requires $O(AN)$ time, since the final for all loop is the most expensive step. As before, this means SIMPLIFY is no more complex than copying a genome.

In CGP it is common to report what fraction of the genome is active, as this represents the easiest estimate of solution size. By utilizing semantic comparison and the SIMPLIFY algorithm, we can get a much better estimate of what purpose each node in the genome serves. Table II provides two columns relevant to node behavior: active and used. The former gives the traditional measure of the median amount of active nodes in the solution found. The latter is the median number of nodes still active after applying the SIMPLIFY algorithm.

On all problems and for all algorithm combinations, the number of active nodes is far smaller than the genome size, and the number of used nodes is far smaller than the number of active nodes. In some extreme cases less than 1% of active nodes in the evolved solution are actually necessary. Only the most extreme configurations have even 50% of active nodes in use by the simplified solution.

Tables IV and V help illustrate the behavioral breakdown of nodes for each algorithm combination on the encode and multiply problems, respectively. These two problems were chosen as they represent different ends of the quality spectrum for skip-normal and because together they are sufficiently representative of behaviors seen on the other problems.

Each diagram describes node behavior as falling into eight distinct categories split between active and inactive nodes. Nodes are put into the first category they satisfy.

1) *Constant:* Independent of problem input, constant nodes produce a fixed output.
2) *Used:* Nodes still active after applying SIMPLIFY.
3) *Useful:* Nodes with identical semantics to a "used" node.
4) *Intron:* Any node with semantics identical to a node that was active before applying SIMPLIFY, but was inactive afterward.
5) *Explore:* All remaining nodes.

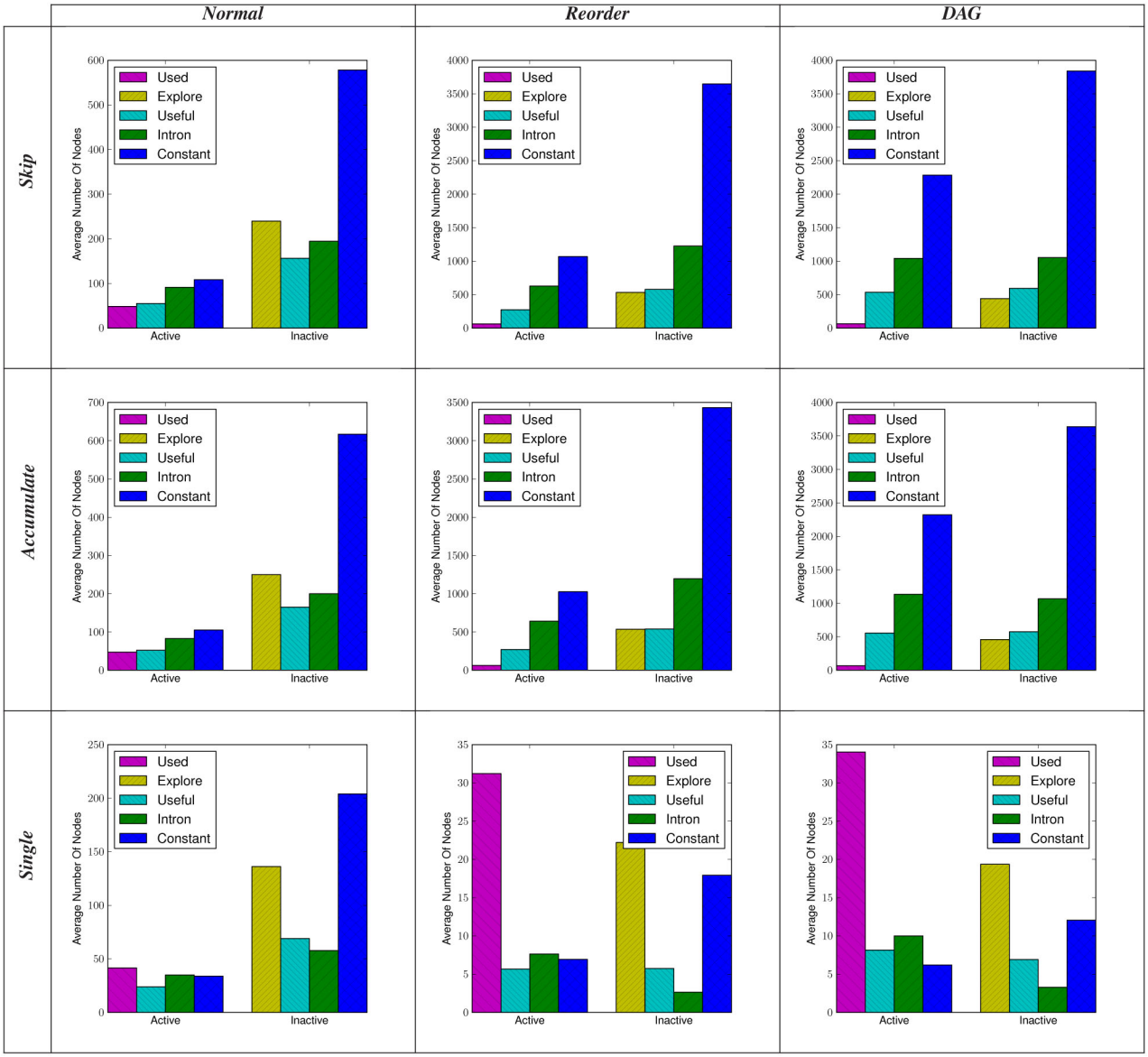
Used nodes are by definition only in the active portion of the genome. Similarly Explore nodes can only be in the inactive portion of the genome. Note that all never active nodes are ignored by these plots, as their behavior is meaningless.

The first striking feature of these tables is how many nodes produce constant outputs. As the instruction set used in our genomes never require constant values to produce any non constant behavior (TRUE NAND X $\Leftrightarrow$ X NAND X) and most operations with constants result in pass through nodes (FALSE OR X $\Leftrightarrow$ X), we would not expect evolution to proliferate these constant nodes. If these constants are in fact of no use, future work could likely improve CGPs performance by reducing their prevalence. Yet it is possible they serve an evolutionary function by allowing mutation more flexibility. For instance, pass through nodes may be allowing mutation to create adjustments not easily done if this behavior was forbidden.

As with previous analysis, skip and accumulate have nearly identical plots, regardless of ordering method and problem. Single again has a significantly different plot when paired with reorder or DAG, with some hits of a change on Encode with normal. These combinations were also the most likely to have significantly smaller genome sizes. From Table II we see that the number of used nodes on each problem is relatively constant across the combinations even though the number of active nodes and the genome sizes changed significantly. As such, the proportion of nodes in the used category will vary as these other features change.

From Tables IV and V, having the plurality of active nodes in the "used" category appears to correlate with increase evaluations to success in Table II. Notice that in all single plots the tallest bar for active nodes is "used," and the only non single plots where this is true are normal on multiply. On encode, single runs had the highest MES, with normal runs doing similarly on multiply. This appears to imply that having duplication of node behavior in the active nodes, and not just inactive nodes, is beneficial to search. If this is true, it may

TABLE IV
AVERAGE NODE BEHAVIOR FOR ALL NINE ALGORITHM COMBINATIONS ON THE ENCODE PROBLEM. EXCLUDES NEVER ACTIVE NODES



help explain why CGP uses such large genomes, as normal is only able to achieve enough active nodes to allow duplication when the genome size significantly exceeds the actual solution size. Also the success of reorder and DAG on multiply may also be partially be attributable to this ability to increase the number of active nodes.

One significant difference in behavior between the two problems is proportion of explore nodes. A potential explanation is that on 16-4-bit encode there are only $2^{16} = 65\,536$ unique semantics while on 3-bit multiply there are $2^{64} = 1.845 \cdot 10^{19}$ unique semantics. Combined with the fact that not all semantics are equally likely to be produced, the difference may just be that on Encode nodes are that much more likely to recreate existing behavior. Another possibility is that CGP increases the semantic diversity of inactive nodes over evolutionary time. As such, the fact that multiply required approximately an order of magnitude more evaluations to solve would lead to the discrepancy.

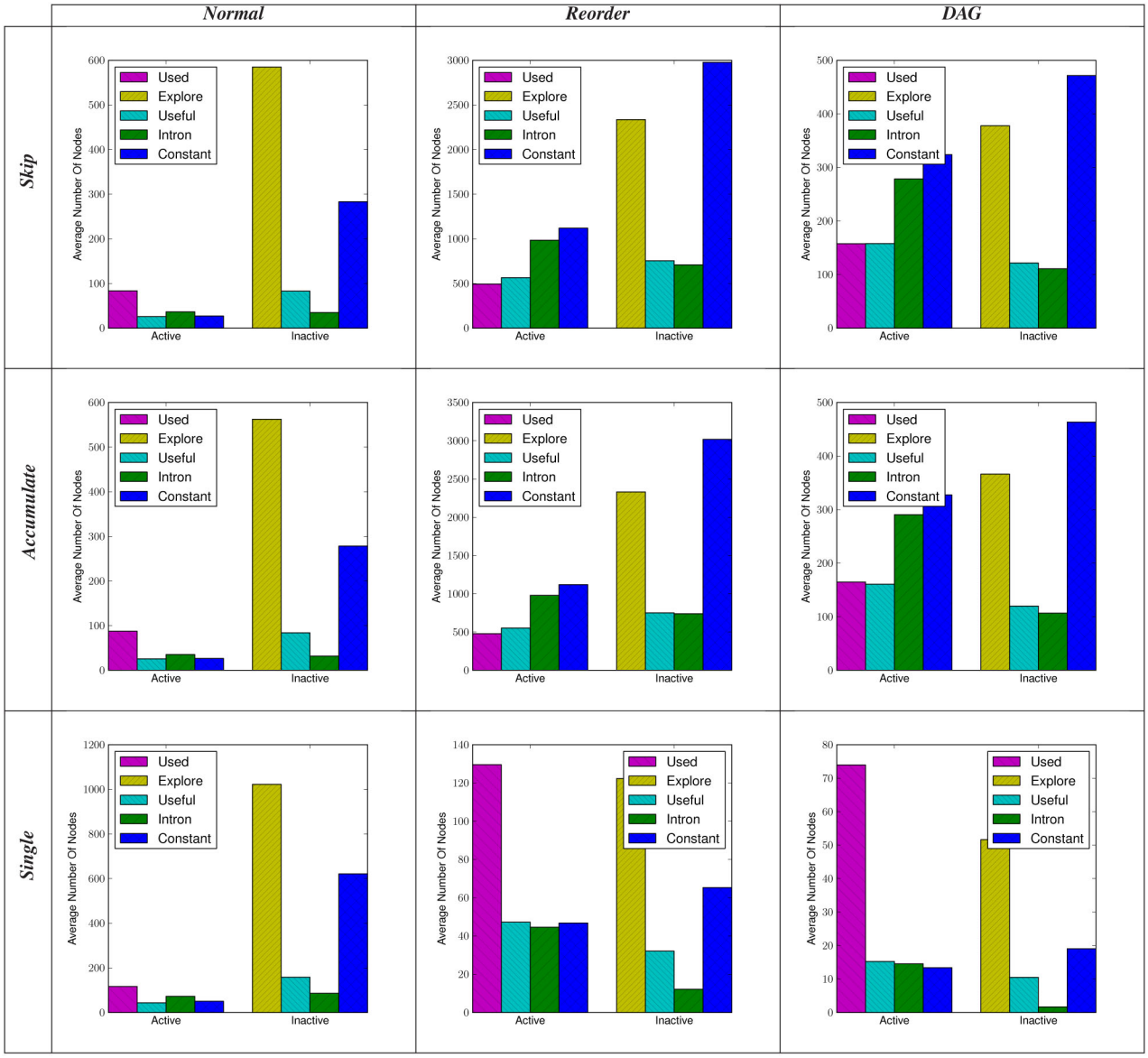
C. Parent Replacement

An important aspect of CGP is its replacement strategy that allows for neutral drift. In each generation the best offspring replaces its parent if it is no less fit. Previous research has shown the negative impact removing this feature can have on performance [19]. However, how it interacts with the different variants of CGP has not been investigated previously.

Table VI provides a novel look at CGPs replacement behavior. Each time an offspring replaced a parent in 50 runs, we recorded how the behavior of each node differed between the parent and the offspring.

The active unchanged column reports how often all active nodes in the offspring have identical semantics to the corresponding nodes in the offspring's parent. This goes beyond the concept of actively identical offspring and checks if the behavior of any active node was modified by the mutation. The rows in each of the four tables are sorted using this column. Smaller

TABLE V
AVERAGE NODE BEHAVIOR FOR ALL NINE ALGORITHM COMBINATIONS ON THE MULTIPLY PROBLEM. EXCLUDES NEVER ACTIVE NODES



values indicate an algorithm combination that produced more changes in active node behavior.

The no reactivation column uses the word reactivation to refer to nodes that were active in some ancestor of the offspring, were inactive in the offspring's parent, but are now again active in the offspring. Reactivation is a measure of how often CGP reuses previously useful active nodes. Reported in the table is how frequently an offspring replaces its parent such that is has no reactivated nodes. Smaller values indicate an algorithm combination that frequently reuses nodes, as opposed to including randomly generated nodes (never active nodes) or ignoring inactive nodes entirely.

The reactivated changed column shows the percentage of reactivated nodes that have a different semantic from the last time they were active. Conversely, reactivated nodes without a change mean that CGP has taken a previously active behavior, stored it in the inactive space, and reactivated it in later mutations. Smaller values indicate an algorithm combination that more often preserve node behaviors while inactive.

Combined these three measures provide some surprising insights. Most notably is that single, a technique designed to force offspring to be actively different from their parents, produces offspring with no changes in active node behavior 20%–40% of the time. This further exposes CGPs ability to have genetically different yet phenotypically identical individuals. Even more surprising is that single appears to increase the chance of producing offspring without changing active behavior. Somewhat unexpectedly normal appears to also reduce changes to active node behavior, causing the largest effect after single.

The likely cause of both is the number of active nodes receiving mutations each generation. Even though single forces a gene in an active node to be mutated, it limits mutation to only one change. If this one change does not

TABLE VI
BEHAVIOR OF OFFSPRING REPLACEMENT FOR EACH ALGORITHM COMBINATION ON EACH PROBLEM. ACT-UN = ACTIVE UNCHANGED; NO REACT = NO REACTIVATION; REACT CHANGED = REACTIVATION CHANGED

3-bit Parity				
Duplicate	**Ordering**	**Act-Un**	**No React**	**React Changed**
Skip	*DAG*	0.00%	3.48%	55.63%
Accumulate	*DAG*	0.00%	8.48%	52.82%
Accumulate	*Reorder*	2.09%	33.26%	58.46%
Skip	*Normal*	2.25%	39.35%	67.17%
Accumulate	*Normal*	2.69%	39.99%	67.27%
Skip	*Reorder*	11.46%	53.99%	59.47%
Single	*Reorder*	24.94%	80.49%	60.63%
Single	*Normal*	26.44%	83.55%	59.96%
Single	*DAG*	27.80%	83.73%	59.92%
16-4-bit Encode				
Duplicate	**Ordering**	**Act-Un**	**No React**	**React Changed**
Accumulate	*DAG*	0.05%	4.06%	48.70%
Skip	*DAG*	0.06%	3.85%	48.33%
Accumulate	*Reorder*	1.77%	19.24%	55.47%
Skip	*Reorder*	2.02%	19.80%	56.15%
Skip	*Normal*	14.16%	54.78%	64.14%
Accumulate	*Normal*	14.52%	54.79%	64.17%
Single	*Normal*	33.77%	79.34%	59.86%
Single	*DAG*	35.98%	80.91%	58.39%
Single	*Reorder*	39.95%	78.91%	59.72%
4-16-bit Decode				
Duplicate	**Ordering**	**Act-Un**	**No React**	**React Changed**
Accumulate	*DAG*	0.03%	10.10%	54.99%
Skip	*DAG*	0.05%	10.21%	51.33%
Accumulate	*Reorder*	9.98%	49.65%	70.57%
Skip	*Reorder*	12.53%	51.63%	70.24%
Single	*Reorder*	28.38%	79.73%	66.85%
Single	*Normal*	29.43%	80.86%	65.85%
Single	*DAG*	32.60%	81.73%	57.70%
Accumulate	*Normal*	51.11%	84.54%	72.57%
Skip	*Normal*	65.47%	89.17%	72.21%
3-bit Multiply				
Duplicate	**Ordering**	**Act-Un**	**No React**	**React Changed**
Accumulate	*Reorder*	1.14%	22.09%	71.75%
Accumulate	*DAG*	4.52%	45.38%	63.42%
Skip	*DAG*	5.05%	45.58%	63.03%
Skip	*Reorder*	10.40%	47.67%	71.79%
Single	*Normal*	18.85%	78.19%	70.11%
Single	*DAG*	20.72%	80.20%	65.10%
Single	*Reorder*	22.14%	80.21%	69.76%
Accumulate	*Normal*	57.69%	87.27%	78.95%
Skip	*Normal*	73.46%	92.04%	79.07%

change that node's behavior, the offspring will be marked as active unchanged. Conversely, skip and accumulate can mutate any number of active genes each generation, improving their chance of a change in behavior. Similarly, even though normal makes no explicit change in mutation, it is characterized by a significantly smaller number of active nodes than reorder or DAG. More active nodes means more chances for mutation to change the behavior of at least one.

Somewhat unintuitively, node reactivation and active node semantic changes appear to be highly correlated. This is apparent in the fact that the no reactivation column in each table is almost perfectly sorted, even though rows were sorted by the active unchanged column. Again the number of active nodes is the likely cause. Having fewer active nodes reduces the number of introns, making it more difficult for inactive nodes to become active. Fewer active nodes also decreases how many inactive nodes could have ever been active.

Potentially the most interesting results come from the control combination of skip-normal. As problem difficulty increases, skip-normal produces an increasingly large number of offspring with no change in active node behavior, with little node reactivation, and a high chance of changing nodes before they are reactivated. This suggests two important features: skip is likely saving a significant number of evaluations, and normal CGP is likely using inactive nodes primarily to inject random behavior. The former comes from the argument that most of the active unchanged individuals are also likely actively identical to their parents. The latter is supported by the fact that skip-normal infrequently reactivates a node with tested behavior with offspring frequently connecting in untested inactive nodes.

VII. CONCLUSION

Generally, CGP appears to require the least number of evaluations when the genome size far exceeds what is necessary for the problem at hand. When properly tuned, almost all variants had only a tiny fraction of the genome active in their final solution. Furthermore, a significant portion of these active nodes were found to be redundant; removable using a very simple reduction algorithm. The only algorithm combinations that preferred more reasonable genome sizes were those using single, especially when paired with reorder or DAG. Yet we theorize this may have more to do with single scaling poorly with the number of active nodes than with it overcoming whatever causes traditional CGP to prefer enormous genomes.

Compared on MES, normal and reorder are likely to be equally good for simple problems, with some suggestion that reorder performs better as problems become more difficult. In general DAG was found to be unnecessarily time consuming, as it required higher algorithmic complexity without any performance gain. As such we would suggest reorder for general future CGP use.

In line with the results from [17], skip and accumulate continued to show no significant difference in effectiveness or behavior. As skip is simpler to understand and implement, we suggest accumulate be ignored in all future CGP applications. Single's behavior continues to be enough different from skip to warrant further study, especially with problems requiring higher numbers of active genes. In general though, if evaluation time is significantly impacted by the number of active nodes or the total size of the genome, we suggest single. While mutation probability appears relatively independent of problem and algorithm combination (all of our tuning resulted in 0.002 being optimal or close to it), single still has the advantage of not requiring this parameter at all.

As to actual behavior, CGP genomes include a surprising amount of redundant and unused nodes. As theoretically predicted in [14], normal ordering leaves large sections of the genome completely unused while reorder and DAG are more likely to utilize the whole genome. Further analysis showed that CGP, at least with our function set, produced a large number of constant nodes, such that their output is independent of problem inputs. As the binary domain only has two constants (always true and always false), each genome at most should

require two constant nodes. Even so these behaviors were found duplicated in both the active and inactive portions of final solutions for all algorithm variants. Also common was duplication of node behavior, even though CGPs encoding does not require any duplication to produce any value. Yet attempting to prevent or remove this behavioral duplication may hinder CGPs evolutionary success, as it will limit neutral drift.

CGPs use of neutrality has become one of its cornerstone arguments for its success. Yet the reason for neutrality's importance appears to be different from previous arguments. In our testing of traditional CGP we found large sections of the genome were never used by any ancestor of the final solution. Furthermore, offspring almost never include active nodes that were inactive in their direct parent, but active in a previous ancestor. Even when reactivation does happen, the node has almost always changed behavior significantly from when it was active. As such we suggest the advantages of neutrality in regards to inactive nodes is not that it allows the storage of previously useful structures, but that it allows inactive nodes to be constantly changed facilitating mutational inclusion of what are effectively random nodes. Under this theory, CGP would likely benefit from periodically resetting all inactive nodes. Note that with reorder and DAG these conclusions do not hold as a reasonable amount of node reactivation occurs.

While this analysis was limited to boolean problems by necessity of some of the analysis methods, most of the conclusions have no obvious reason to be limited to the boolean domain. Still, applying semantic analysis to detect changes in behavior and behavior duplication, along with looking at never active nodes and tuned parameter settings, on other domains will help solidify our conclusions.

REFERENCES

[1] J. M. Daida, H. Li, R. Tang, and A. M. Hilss, "What makes a problem GP-hard? Validating a hypothesis of structural causes," in *Genetic and Evolutionary Computation—GECCO-2003* (Lecture Notes in Computer Science), vol. 2724. Chicago, IL, USA: Springer-Verlag, pp. 1665–1677.

[2] N. X. Hoai, R. I. B. McKay, and D. Essam, "Representation and structural difficulty in genetic programming," *IEEE Trans. Evol. Comput.*, vol. 10, no. 2, pp. 157–166, Apr. 2006.

[3] S. Luke and L. Panait, "A comparison of bloat control methods for genetic programming," *Evol. Comput.*, vol. 14, no. 3, pp. 309–344, 2006.

[4] J. Miller, "What bloat? Cartesian genetic programming on Boolean problems," in *Proc. 2001 Genet. Evol. Comput. Conf. Late Breaking Papers*, San Francisco, CA, USA, pp. 295–302.

[5] V. K. Vassilev and J. F. Miller, "The advantages of landscape neutrality in digital circuit evolution," in *Proc. 3rd Int. Conf. Evolvable Syst.*, Edinburgh, U.K., 2000, pp. 252–263.

[6] A. J. Payne and S. Stepney, "Representation and structural biases in CGP," in *Proc. 2009 IEEE Congr. Evol. Comput.*, Trondheim, Norway, pp. 1064–1071.

[7] J. F. Miller and P. Thomson, "Cartesian genetic programming," in *Proc. Genet. Program. (EuroGP)*, vol. 1802. Edinburgh, U.K., Apr. 2000, pp. 121–132.

[8] J. A. Walker and J. F. Miller, "The automatic acquisition, evolution and reuse of modules in Cartesian genetic programming," *IEEE Trans. Evol. Comput.*, vol. 12, no. 4, pp. 397–417, Aug. 2008.

[9] S. Harding and J. F. Miller, "Evolution of robot controller using Cartesian genetic programming," in *Proc. 8th Eur. Conf. Genet. Program.*, vol. 3447. Lausanne, Switzerland, Mar./Apr. 2005, pp. 62–73.

[10] M. M. Khan and G. M. Khan, "A novel neuroevolutionary algorithm: Cartesian genetic programming evolved artificial neural network (CGPANN)," in *Proc. 8th Int. Conf. Frontiers Inf. Technol.*, Islamabad, Pakistan, 2010, pp. 48.1–48.4.

[11] S. Harding, V. Graziano, J. Leitner, and J. Schmidhuber, "MT-CGP: Mixed type Cartesian genetic programming," in *Proc. 14th Int. Conf. Genet. Evol. Comput. Conf. (GECCO)*, Philadelphia, PA, USA, Jul. 2012, pp. 751–758.

[12] S. Harding, J. Miller, and W. Banzhaf, "Self modifying Cartesian genetic programming: Fibonacci, squares, regression and summing," in *Proc. 12th Eur. Conf. Genet. Program. (EuroGP)*, vol. 5481. Tuebingen, Germany, Apr. 2009, pp. 133–144.

[13] J. F. Miller and S. L. Smith, "Redundancy and computational efficiency in Cartesian genetic programming," *IEEE Trans. Evol. Comput.*, vol. 10, no. 2, pp. 167–174, Apr. 2006.

[14] B. W. Goldman and W. F. Punch, "Length bias and search limitations in Cartesian genetic programming," in *Proc. 15th Int. Conf. Genet. Evol. Comput. Conf. (GECCO)*, Amsterdam, The Netherlands, Jul. 2013, pp. 933–940.

[15] J. F. Miller, "Cartesian genetic programming," in *Cartesian Genetic Programming* (Natural Computing Series). Berlin, Germany: Springer, 2011, ch. 2, pp. 17–34.

[16] Z. Vasicek and K. Slany, "Efficient phenotype evaluation in Cartesian genetic programming," in *Proc. 15th Eur. Conf. Genet. Program. (EuroGP)*, vol. 7244. Malaga, Spain, Apr. 2012, pp. 266–278.

[17] B. W. Goldman and W. F. Punch, "Reducing wasted evaluations in Cartesian genetic programming," in *Proc. 16th Eur. Conf. Genet. Program. (EuroGP)*, vol. 7831. Vienna, Austria, Apr. 2013, pp. 61–72.

[18] X. Cai, S. L. Smith, and A. M. Tyrrell, "Benefits of employing an implicit context representation on hardware geometry of CGP," in *Proc. 6th Int. Conf. Evolvable Syst. (ICES)*, vol. 3637. Sitges, Spain, Sep. 2005, pp. 143–154.

[19] T. Yu and J. Miller, "Neutrality and the evolvability of Boolean function landscape," in *Proc. Genet. Program. (EuroGP)*, vol. 2038. Lake Como, Italy, Apr. 2001, pp. 204–217.

[20] D. R. White *et al.*, "Better GP benchmarks: Community survey results and proposals," *Genet. Program. Evolvable Mach.*, vol. 14, no. 1, pp. 3–29, Mar. 2013.

[21] N. F. McPhee, B. Ohs, and T. Hutchison, "Semantic building blocks in genetic programming," in *Proc. 11th Eur. Conf. Genet. Program. (EuroGP)*, Naples, Italy, Mar. 2008, pp. 134–145.

Brian W. Goldman received the B.S. and M.S. degrees in computer science from Missouri University of Science and Technology, Rolla, MO, USA, in 2010 and 2012, respectively. He is currently working toward the Ph.D. degree from the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, USA.

His research interests include evolutionary computation and genetic programming.

William F. Punch received the Ph.D. degree from The Ohio State University, Columbus, OH, USA.

He is an Associate Professor with the Computer Science and Engineering Department, Michigan State University (MSU), East Lansing, MI, USA. He is also the Director of the MSU High Performance Computing Center. His research interests include evolutionary computation, high performance computing, and computing pedagogy. He co-directs the Genetic Algorithms Research and Application Group and is on the Executive Committee of the NSF Science and Technology Center, BEACON, Center for the Study of Evolution in Action, MSU. His book with Rich Enbody entitled *The Practice of Computing Using Python* (Addison-Wesley, Boston, MA, USA) is now in its second edition.