Dynamic Primitive Granularity Control: An Exploration of Unique Design Considerations

Braden N. Tisdale
BONSAI Lab
Auburn University
Auburn, Alabama
braden@auburn.edu

Aaron Scott Pope
Advanced Research in Cyber Systems
Los Alamos National Laboratory
Los Alamos, New Mexico
aaronspope@acm.org

Daniel R. Tauritz
BONSAI Lab
Auburn University
Auburn, Alabama
dtauritz@acm.org

ABSTRACT

Dynamic primitive granularity control (DPGC) is a promising avenue for improving the performance of genetic programming (GP). However, it remains almost entirely unexplored. Further, it may pose many unique challenges in its design and implementation that traditional GP implementations do not. This paper presents an implementation of DPGC in order to determine what aspects of conventional GP design and implementation require special consideration. There are some common techniques used in GP that have been found here to negatively impact DPGC's ability to improve performance. Parsimony pressure appears to disproportionately penalize low-level primitives, and a mixed-granularity population suffers from heavy biases towards particular granularity levels, seemingly to the detriment of evolution. This paper provides hypotheses as to why these conventional techniques harm DPGC implementations, as well as several potential alternatives for use in the future that may remedy these detrimental effects.

CCS CONCEPTS

• **Mathematics of computing** → **Graph algorithms**; • **Software and its engineering** → **Genetic programming**; • **Computing methodologies** → **Genetic programming**; • **Theory of computation** → **Algorithm design techniques**.

KEYWORDS

Evolutionary computation, genetic programming, generative hyper-heuristic, primitive granularity control

ACM Reference format:

Braden N. Tisdale, Aaron Scott Pope, and Daniel R. Tauritz. 2020. Dynamic Primitive Granularity Control: An Exploration of Unique Design Considerations. In *Genetic and Evolutionary Computation Conference (GECCO '20)*, July 8–12, 2020, Cancún, Mexico. ACM, New York, NY, USA, 9 pages. DOI: 10.1145/3377929.3398131

1 INTRODUCTION

Genetic Programming (GP) is one of the most prevalent forms of generative hyper-heuristic, a type of machine learning which seeks to find an optimal sequence of low-level operations that, combined, form an algorithm suited to solve some particular class of problems. These low-level operations are referred to as primitives, and can range in complexity from full algorithms that could potentially be a solution on their own down to the lowest level of instructions available to a given processor. This complexity is referred to as the granularity of the primitive. High-level or coarse-grained primitives are those which are most complicated and algorithmic, while low-level or fine-grained primitives are the simpler instructions or operations. For the remainder of this paper the terms 'high-level' and 'low-level' will be used exclusively.

High-level primitives have traditionally been preferred for many problems for several reasons [1]. Perhaps most importantly, they require far less human effort to implement than lower-level primitives. Because representing a given algorithm requires fewer primitives if represented at a higher level, fewer primitives need be implemented. In addition to lowering development time, this means the number of possible combinations of primitives is dramatically lower, leading to a quicker traversal of the search space.

However, while the search space may be traversed more rapidly, this is because it is much smaller – a high-level primitive set will not be able to represent many algorithms, which may include globally optimal solutions. Any primitive set which is not Turing complete will exclude some algorithms from the search space. This is generally not considered a problem, as using such a primitive set would require searching through the space containing every possible algorithm, and because the primitives an intelligent user provides are typically hand-picked based on their presumed ability to represent high-fitness algorithms for the problem being targeted.

With these considerations in mind, it is important to consider how granular the primitive set for any given application of GP should be. A higher-level set will likely require less effort implementing the primitives, and will perform a quicker, coarser search. Yet lower-level primitives may allow for a higher fitness to be found overall. The naïve approach – simply providing a larger number of primitives, representing a wide range of the granularity spectrum – makes the search space prohibitively large as every

© 2020 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the United States Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

GECCO '20 Companion, July 8–12, 2020, Cancún, Mexico

© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7127-8/20/07...\$15.00
<https://doi.org/10.1145/3377929.3398131>

primitive added exponentially increases the possible number of algorithms that can be represented [2]. Thus, a better method is not just to provide that large number of primitives, but also to provide some scheme that limits the number of primitives available at any given time. This requires categorization or ordering of the primitives according to some criteria. Most naturally this may be done by categorizing the primitives by their granularity level.

Once this is done, the search may be restricted during various points to some granularity level or levels, mitigating the slow-down caused by a massive primitive set while providing a mechanism for the search to leverage the advantages of both high-level and low-level primitives when appropriate. Such a method for restricting the primitive set throughout evolution has been coined dynamic primitive granularity control (DPGC) [5]. However, DPGC has not been studied extensively other than as a proof-of-concept.

The goal of this paper is to begin a proper exploration of DPGC's impacts on existing GP applications. More precisely, this paper aims to show how some conventional techniques can be detrimental to DPGC, and to propose alternatives. This work also tests a more general implementation of DPGC than prior works.

2 BACKGROUND

DPGC is a fairly new concept and even the general investigation of primitive granularity's impacts, without considering control at all, is not much older. One of the studies that precipitated the study of primitive granularity was conducted by Martin & Tauritz in 2015, who empirically demonstrated the intuitive notion that making more primitives available increases the program's ability to represent an optimal solution, while also increasing the size of the search space [2]. This means that, while increasing the number of primitives available may end up allowing for more fit solutions, it may also cause evolution to take a prohibitively large amount of time or even cause it to converge at a mediocre local optimum. This demonstrates the importance of choosing the 'correct' primitives to add to the search, which may change during different stages of evolution.

Pope et al showed in 2016 that more high-level primitives can be used as opposed to more low-level primitives to achieve similar results as found by Martin & Tauritz [1]. Instead of simply adding more primitives to the search, existing primitives can be broken down to their constituent pieces to raise the potential fitness of the solutions at the cost of increasing the size of the search space. This empirically shows the value of lower-level primitives, while also demonstrating the drawbacks of using them exclusively. This forms a very compelling argument that DPGC can be used to leverage the advantages of both lower-level and higher-level primitives, and helped spur further interest in the subject.

Prior to these investigations of primitive granularity, in 2011, Goldman & Tauritz showed that varying parameters throughout the execution of an evolutionary algorithm leads to increased fitness over static parameters given an equal number of evaluations [3]. Though this was not tested on the primitive set (it was not tested on GP at all), it demonstrated that the evolutionary process can benefit significantly from dynamic parameters. This was an important step in the ideation of DPGC, as it demonstrated that even rudimentary

schemes for dynamically altering conditions during evolution have the potential to lead to significant performance gains.

In 2019, Harter et al combined these concepts to demonstrate that varying primitive granularity levels during evolution can significantly improve GP performance over static primitive granularity schemes [4]. This study used a sequence of values to prescribe a primitive granularity schedule. Each value in the sequence was used as the granularity level for a set amount of time before the next value in the sequence was used. In this way, the evolutionary process was divided into a series of equal-length epochs during which a specific granularity level was prescribed. A meta-search was used to find the best such sequences, which were then tested against control runs that used only one level of primitive granularity throughout the run. Again, the dynamic schedules were found to be superior to the static ones.

3 RELATED WORK

Likely the first individual to consider different levels of primitive granularity was Koza during his exploration of what he termed 'automatically defined functions' [9]. Koza devised a method of allowing GP to define a combination of primitives as an automatically defined function, which effectively became its own primitive. This is nearly identical to the concept of macro-primitives, another proposed method for improving GP performance that is closely related to DPGC. However, despite being implemented successfully very early on in GP's history, this technique is rarely if ever used today judging by a distinct lack of publications on the topic. This may be due to the difficulty of employing such a technique for complex problems rather than a lack of awareness in the GP community, as it was listed as one of the most important open issues in GP in a 2010 article and the concept has been studied to some extent under various different terms [10, 11, 12, 13]. Luckily, if DPGC proves successful, it should allow GP to leverage macro-primitives to an even greater extent than if they were used without it.

Most recently, Pope has investigated a handful of control schemes for primitive granularity [5]. One important finding was that moving from high-level primitives to low-level primitives appears to be more beneficial than vice versa. Pope also found two schemes that seem to perform particularly well – a self-adaptive scheme, where the solutions encode whether they should be high-level or low-level, and an alternating scheme, where the granularity level switches when convergence is detected. An interesting implication of this research is that more complicated and dynamic schemes appear to perform better than more rudimentary schemes. This suggests that more precise schemes can be devised that leverage the different granularity levels more intelligently for even greater performance. Worth noting, however, is that ultimately very few schemes were tested and that the number of tests performed were quite low due to prohibitively long runtimes. A follow-up to the experiments in [5], which laid out the basic schemes, was a further exploration of the self-adaptive scheme in particular [15]. These experiments were primarily focused on serving as proof-of-concept that DPGC could be beneficial to GP under at least some conditions.

4 METHODOLOGY

Solutions in this experiment are represented using strongly-typed Koza-style trees. Trees were chosen as the representation for several reasons. Firstly, they allow for relatively simple decomposition mappings, the mapping from a higher-level primitive to a functionally equivalent combination of lower-level primitives. Second, Koza has already shown that macro-primitives can be created successfully using trees [9], which this author hopes to explore as an extension of this research in the future. Third, they are the most widely used type of GP, and this paper is concerned with using the most typical GP implementation possible [14]. And, last, because Pope used trees in both his prior experiments with DPGC [5, 15]. The trees are strongly-typed for this last reason.

4.1 Strongly-Typed Tree GP

The tree implementation follows conventional GP methodology, with a typical $(\mu + \lambda)$ generational cycle and typical subtree algorithms for reproduction: each generation, λ parents are selected from the μ individuals by k -tournaments with replacement. Each parent will choose to generate a child by mutation based on the configurable mutation chance, or by crossover otherwise. If the child is to be created by crossover, a different parent is randomly selected to donate a subtree as described in [6]. This process produces only one child and does not affect the donor parent's reproductive process. After reproduction and evaluation, survival is determined by truncation.

The experimental results reported in this paper were obtained on a benchmark with 3 objectives; NSGA-II is used to determine levels of nondomination in the population which are used as the basis for comparing fitnesses [7]. Every generation, the nondominated individuals are added to a list of individuals referred to as the Pareto front. Then, the individuals in the Pareto front are compared against each other for dominance, and any dominated individuals are discarded from the Pareto front. In this way, the Pareto front stores every individual generated throughout the run that has never been dominated.

Due to the highly variable execution time of individuals, each individual is allotted a configurable amount of time to execute. If execution surpasses this threshold, it is immediately aborted and the individual is assigned an arbitrarily poor fitness. This threshold is updated by passing in the execution time of every execution when it either finished or was aborted to Algorithm 1. The threshold is checked every time a primitive executes by passing in the execution time to Algorithm 2.

4.2 Target Problem

It is highly preferable that the chosen problem be previously tested at multiple granularity levels. Further, by advice from Pope that DPGC likely requires sufficiently difficult problems to show significant improvements to GP, the chosen problem should be easily transformed into a more difficult problem. These criteria led to the selection of evolving random graph generators.

In the context of GP, this means evolving an algorithm that can generate random graphs which emulate certain characteristics of a

set of target graphs. Details on the specific sets of graphs being targeted are found in Section 5. Graphs here are simple (one edge per pair) and undirected (both ends of an edge are treated identically) with no loops (a vertex with an edge to itself).

Algorithm 1: Algorithm for keeping a running average of execution times

```

1 RunningAverage(time, window, hard, mult, memory, over)
Input: a length of time, a queue of times, a hard cutoff, a
cutoff multiplier, the number of elements to keep in the
window, and an overestimation factor
2 if |window| = memory then
3   above  $\leftarrow$  (time > average(window) * mult)
   OR (time > hard);
4   if above then
5     time  $\leftarrow$  time * over;
6   end
7   dequeue(window);
8   enqueue(window, time);
9 else
10  enqueue(window, time);
11 end
```

Algorithm 2: Algorithm for determining if execution should be aborted

```

1 Threshold(window, hard, mult, memory)
Input: a length of time, a queue of times, a hard cutoff, a
cutoff multiplier, and the number of elements to keep in the
window
Output: true if time is above either cutoff, false otherwise
2 if |window| = memory then
3   above  $\leftarrow$  (time > average(window) * mult)
   OR (time > hard);
4   return above;
5 else
6   return (time > hard);
7 end
```

4.3 Primitives

During execution, every primitive has access to the graph that will be returned by the *root* primitive. At the start of execution, this is an empty graph of the same order (number of vertices) as the target graph. The graph type does not actually pass any data; it simply denotes that a primitive may add edges to the graph. The signature of each primitive is shown in Table 1.

In order to accurately emulate higher-level primitives using lower-level primitives, some additional steps must be taken to allow data to be passed down the tree in addition to the more traditional upwards direction. For example, the *make-clique* primitive, which creates a clique (a set of vertices with every possible edge), takes an input list and operates on that list. In doing so, it must iterate through each entry in the list and perform operations based on the current iteration. Consider the subtree generated by decomposing this primitive. In the subtree, primitives must be able to access the data used in the current iteration in a synchronized fashion. This motivates the apparently unusual design decisions regarding how certain primitives behave.

The primitives *vertex* and *next-in-iteration* behave in a way that inverts the typical flow of information in a Koza-style GP tree by taking information from *for-vertex-in-list* and *for-two-lists* primitives above them. First, an explanation of the *for* primitives.

There exist three stacks which can be accessed by any primitive during execution; a *current-iterator* stack, an *end-iterator* stack, and a *begin-iterator* stack. When a *for-vertex-in-list* primitive is entered during execution, it collects the *vertex_list* of its first child then pushes onto these stacks; the *current-iterator* stack receives an iterator to the first vertex in the list, while the *end-iterator* and *begin-iterator* stacks receive iterators to the end and beginning of the list respectively, to allow for bounds-checking. *for-two-lists* behaves similarly, but takes two lists and pushes iterators for each to the stacks.

After this initialization process, the graph child of the *for* primitive is executed. Then, the *for* primitive increments its iterator(s) on the *current-iterator* stack and executes its *graph* child again. The *for-vertex-in-list* primitive repeats this process until it has iterated through its entire list. The *for-two-lists* primitive repeats this process until it has iterated through its shorter list. Then, the iterator stacks are popped and the *for* primitives have finished executing. The *vertex_list* children are only evaluated when the *for* primitives are first entered; the list will not change during execution of the *for* primitive or its *graph* child. However, if the *for* primitive is entered another time, as in the case of nested loops, then it will reevaluate its *vertex_list* child.

The *for-constant* primitive works in a simpler fashion; it executes its *graph* child the number of times indicated by its *loop_counter* child. It does not modify the stacks in any way.

The *vertex* and *next-in-iteration* primitives store a nonnegative integer as part of their instantiation. If this value is zero, then they will randomly select a vertex from the graph. Otherwise, this value denotes a location in the stacks: a value of one denotes the iterator at the bottom of the stack (pushed by the current primitive's highest *for-vertex-in-list* or *for-two-lists* ancestor), a value of two denotes the iterator above that in the stack, etc.

For a *vertex* primitive with a value that is not zero, it returns the vertex pointed to by the iterator at that location in the *current-iterator* stack. For a *next-in-iteration* primitive with a value that is not zero, it attempts to return the vertex after the iterator. If that would be out of bounds, it instead returns the first vertex in the iterator's list. This is facilitated by the *end-iterator* and *begin-iterator* stacks.

When a *vertex* or *next-in-iteration* primitive is instantiated by mutation or the initial population, its value is randomly selected from $[0, s]$ where s is the size of the stack in this part of the tree, which can be determined by traversing towards the root and noting the *for-vertex-in-list* and *for-two-lists* primitives encountered. After crossover, any *vertex* or *next-in-iteration* primitives in the donated subtree are checked to ensure that their values are in the range $[0, s]$; if not, their value is reset to a random integer value in that range.

The remaining primitives are simpler to understand. Note that for all primitives, when any edge is to be added, if the input vertices are already adjacent or are the same vertex, no edge is added.

Table 1: Primitives

Name	Output Type	Input Types
Low-level		
<i>root</i>		graph
<i>vertex</i>	vertex	see explanation
<i>next-in-iteration</i>	vertex	see explanation
<i>degree-proportional</i>	vertex	
<i>index-to-vertex</i>	vertex	index
<i>vertex-index</i>	index	
<i>add-edge</i>	graph	vertex, vertex
<i>if[-else]</i>	graph	probability, graph [, graph]
<i>random-wire</i>	graph	
<i>constantprob</i>	probability	
<i>all-vertices</i>	vertex_list	
<i>degree-[max min]-heap</i>	vertex_list	
<i>vertex-range</i>	vertex_list	index, index
<i>slice-list</i>	vertex_list	probability, vertex_list
<i>filter-list</i>	vertex_list	probability, vertex_list
<i>degree-[max min]-sort</i>	vertex_list	vertex_list
<i>for-vertex-in-list</i>	graph	vertex_list, graph
<i>for-two-lists</i>	graph	vertex_list, vertex_list, graph
<i>loop-counter</i>	loop_counter	
High-level		
<i>add-pairwise</i>	graph	vertex_list, vertex_list
<i>make-clique</i>	graph	vertex_list
<i>make-ring</i>	graph	vertex_list
<i>add-edges-with-prob</i>	graph	probability, vertex_list
Any-level		
<i>for-constant</i>	graph	loop_counter, graph
<i>fork</i>	graph	graph, graph

Primitives that are considered terminals are italicized.

root: executes its child then returns the graph

degree-proportional: returns a vertex from the graph, selected probabilistically with relative probability equal to its degree; if the graph is empty, all vertices have equal probability

index-to-vertex: converts the input index to the graph's corresponding vertex

vertex-index: constant integer in $[0, n]$ where n is the graph's order

add-edge: adds an edge between the input vertices

if[-else]: if/if-else branching based on the input probability

constantprob: constant value in $[0, 1]$

all-vertices: returns a list of all vertices in the graph

degree-[max|min]-heap: returns a list of all vertices in the graph, ordered as a max/min heap based on their degree

vertex-range: returns a list containing all vertices in the graph between the first index and the second index, inclusive, wrapping around if the second index is less than the first index

slice-list: with its probability input p and its *vertex_list* input of length l , returns the first $\lfloor l * p \rfloor$ elements of the input list

filter-list: returns a subset of the input list where each element has the input probability of being in the subset

degree-[max|min]-sort: returns the input list sorted by degree

loop-counter: a constant positive integer that is less than 150

add-pairwise: for each vertex in the shorter list, adds an edge between it and the vertex at the same index in the longer list

make-clique: adds an edge to every pair of vertices in the input list

make-ring: adds an edge between each vertex in the input list and the subsequent vertex in the list, including between the first and last vertices

add-edges-with-prob: for every pair of vertices in the input list, adds an edge with chance equal to the input probability

random-wire: picks two random vertices and adds an edge¹

fork: does nothing, is used to allow for more graph operations to be executed

4.4 Primitive Selection & Granularity

The initial population is created by using ramped half-and-half as described in [8]. Individuals were constrained to a configurable maximum depth, but could go over this due to the way primitives are selected². The only modification to this method is that the full-depth method may insert terminals before reaching the specified depth, if a required type is only returned by terminal primitives.

The methods for selecting valid granularity levels and primitives are described in Algorithm 3 and Algorithm 4 respectively. As there are only two levels of granularity implemented here, the granularity levels available are 0 (low-level) and 1 (high-level). Any-level primitives have a place in the lists at every granularity level. In effect, they can be chosen by these algorithms regardless of the input granularity level.

An important note about Pope's prior works on DPGC [5, 15] is that the higher-level primitives were explicitly defined as a combination of lower-level primitives. To test DPGC in a more generalized implementation, this paper relaxes that constraint. Here, the algorithms used by higher-level primitives during execution need not necessarily be representable using the lower-level primitive set. Instead, only their visible functionality needs to be representable, potentially as a different algorithm. Primarily, this facilitates quicker executions as higher-level primitives may be more optimally represented by algorithms outside of the algorithmic space reachable by the lower-level primitive set. This could feasibly be relaxed further to allow for some higher-level primitives whose functionality cannot be emulated, or which can only be approximated.

High-level primitives may be decomposed. That is, they are replaced with a subtree of low-level primitives that provides the same functionality. Any-level primitives may not be decomposed. The decomposition for *make-clique* is provided in Fig. 1.

¹ Due to strong typing, a terminal primitive with output type *graph* is necessary to avoid arbitrarily-deep trees. Otherwise, whenever a *graph* terminal is requested, a random *graph* nonterminal is returned. This nonterminal may in turn request one or more *graph* terminals, recursively leading to a massive tree. *random-wire* exists to

Algorithm 3: Algorithm for selecting a valid granularity

1 *SearchGranularity*(p, g)

Input: *list of primitives to search, and desired granularity*

Output: *index of a valid granularity*

```

2 probe ←  $g$ ;
3 while probe ≥ 0 do
4   if  $p[\textit{probe}] \neq \emptyset$  then
5     return probe;
6   else
7     probe ← probe - 1;
8   end
9 end
10 end
11 probe ←  $g + 1$ ;
12 while probe <  $|p|$  do
13   if  $p[\textit{probe}] \neq \emptyset$  then
14     return probe;
15   else
16     probe ← probe + 1;
17   end
18 end

```

Algorithm 4: Algorithm for selecting a primitive

1 *RandomPrimitive*($p, \textit{granularity}$)

Input: *list of primitives to search, and desired granularity*

Output: *a primitive of the desired type*

```

2  $g \leftarrow \textit{SearchGranularity}(p, \textit{granularity})$ ;
3  $x \leftarrow^R p[g]$ ; //Uniform random element from  $p[g]$ 
4 return  $x$ ;

```

make-ring, *add-pairwise*, and *add-edges-with-prob* also have their own unique decompositions. Though the provided subtree emulates the functionality of *make-clique* exactly, the actual algorithm it represents is different: given a list of n vertices, *make-clique* only attempts to add $n(n-1)/2$ edges, the actual number of edges in an undirected clique on n vertices, while the decomposition attempts to add n^2 edges, iterating through each vertex n times.

While not an issue with this specific primitive set, it is likely that some implementations of DPGC may be better off using approximated decomposition mappings that do not perfectly emulate the functionality of the higher-level primitive. There is a trade-off to be considered if having perfect emulation in the decomposition for a given primitive requires adding an excessive number of new lower-level primitives. As discussed previously, large primitive sets are often detrimental to performance [2]. Therefore, in many cases a 'good enough' decomposition may be preferable. Approximate decomposition mappings are also something that can be generated using GP itself, conferring many benefits during the implementation of DPGC – this idea is discussed further in Section 9.

avoid that scenario without complicating and potentially biasing the primitive selection logic.

² When a nonterminal is requested, it is not guaranteed that a nonterminal is returned – there may be no nonterminals of that type. The requested type is always returned.

Figure 1: The subtree generated by decomposing *make-clique*. *child* and *parent* are copies of the original primitive’s parent and child subtrees. The parentheses after the *vertex* primitives indicate their value for stack lookup; *i* is the size of the stack reachable from the *vertex* primitives.

5 EXPERIMENTAL SETUP

Each individual has three objective values. Each of these objective values is to be minimized. As discussed in Section 4.1, when comparing individuals, NSGA-II is used to calculate levels of nondomination with respect to these objective values [7]. The first objective value is the absolute value of the difference between the size (edge count) of the target graph and the size of the generated graph. The second objective value measures the similarity of the degree distributions, calculated as follows, where t and g are the target graph and generated graph respectively:

$$\sum_i (|\{u \in V(t) \mid \deg(u) = i\}| - |\{v \in V(g) \mid \deg(v) = i\}|)^2$$

The third objective value is calculated as $\max(\text{execution time}, \text{lower bound})$. This is clamped to the lower bound so that trivial solutions that execute in short periods of time can be dominated by solutions that generate more accurate graphs while remaining below the bound. This objective value is unrelated to the average execution time calculations in algorithms 1 & 2. The values reported in Section 6 refer to the average used in those algorithms, not to this objective value. This objective is simply a form of parsimony pressure and is not reported, though, as will be discussed in Section 7, this objective is likely a key factor in the observed results.

Seven control schemes have been tested. The first is *All*, which has access to all primitives at all times and never decomposes any. The next four are *Low*, *High*, *Low-to-High*, and *High-to-Low*, implemented the same as in prior work [5]. *High* and *Low* use only the high- and low-level primitives respectively. *Low-to-High* and *High-to-Low* start at one level, and switch to the other after half of their generations. *High-to-Low* decomposes all primitives when it switches.

The sixth scheme, *Self-Adaptive*, is implemented differently than in [5] & [15]. Here, individuals are initialized to a random granularity level, and are *not* segregated by granularity. This lack of segregation was explicitly chosen as a technique that better represents a more generic implementation of GP. During crossover,

Table 2: Configurable parameters

Variable	Value	Note
μ	150	
λ	25	
mutation chance	0.15	
tournament k	15	
generations	300	
max depth	10	
start decay	0.05	Decay scheme
decay scale	1.005	Decay scheme
execution time lower bound	0.01 seconds	for third objective value, not Results
hard cutoff	0.5 seconds	Algorithm 1
cutoff mult	2.5	Algorithm 1
memory	525	Algorithm 1
overestimate	10.0	Algorithm 1
num vertices	100	
ER prob	0.5	Erdős-Rényi
BA edges	5	Barabási-Albert

individuals always retain the granularity level of the parent that was originally selected for crossover (per Section 4.1). During mutation, individuals always have their granularity level switched. This distinctly separates crossover and mutation as being for exploitation and exploration respectively, again in an attempt to stay within GP conventions.

The seventh control scheme, *Decay*, works similarly to *High* in that all new primitives that are generated by initialization or mutation are high-level primitives. The difference here is that, each generation, every high-level primitive in every existing individual has a configurable chance to be decomposed. This chance is multiplied by a constant scaling factor every generation. This directly modifies the existing individuals; this is not tied to reproduction.

The targets used in this experiment were graphs generated following the Erdős-Rényi (ER) model and Barabási-Albert (BA) model described in [1]. A given run only uses one of these: the target graphs for a run are all generated using the same model and the same parameters. Configurable values are shown in Table 2.

Every generation, a new target graph was generated by that run’s model. Due to this, and in order to avoid over-fitting to a specific graph, every individual was evaluated every generation, even if they had already been evaluated in a prior generation.

6 RESULTS

Ten runs of each scheme were run on each graph generator. Measures of central tendency for the minimal values at the end of each run for the second objective (degree distribution) are listed in Table 3. Also listed in Table 3 are the average execution time values across every generation as calculated in Algorithm 1. The first objective is not listed or analyzed due to every run attaining a minimum value of 0. Despite this, the objective is still valuable and

impactful to the experiments as a constraint on the optimization of the other objectives.

Underlined mean values in tables 3 & 4 are statistically worse than all other values according to two-tailed Student's t-tests assuming unequal variance ($\alpha = 0.05$). Bold values are statistically better than at least one non-underlined value.

Table 3: Measures of central tendency for execution times and the minimal degree distribution objective

Scheme	μ_{ER}	σ_{ER}	μ_{BA}	σ_{BA}	t_{ER}	t_{BA}
All	<u>60.2</u>	32.3	<u>85.0</u>	51.3	1.62	2.55
Low	37.4	3.9	64.4	11.1	5.14	1.96
High	39.8	9.2	<u>77.6</u>	34.9	0.47	0.20
Low-to-High	36.0	3.1	58.4	9.9	4.18	3.03
High-to-Low	40.4	7.8	51.4	15.6	5.47	4.17
Self-Adaptive	45.4	16.8	<u>85.0</u>	56.9	2.77	1.57
Decay	43.4	24.5	<u>83.2</u>	30.3	6.40	3.67

Subscripts denote the graph type used as a target. μ & σ denote mean and sample deviation, respectively. t denotes average execution time in milliseconds.

As will be discussed in sections 7 & 8, there is some reason to believe that parsimony pressure biases the search and punishes decomposition. Therefore, Table 4 shows results when the same experiment was repeated without using execution time as an objective value, and without using Algorithm 2 to abort executions. Instead, execution was aborted if and only if it surpassed the hard cutoff of 0.5 seconds. Average execution time for these runs is marked by outliers which took extreme amounts of time compared to the average, which brings the significance of these execution time results into question given the relatively few runs performed. All discussion of execution times relates to Table 3 unless otherwise noted, as they were far less varied.

Table 4: Measures of central tendency for execution times and the minimal degree distribution objective without parsimony pressure

Scheme	μ_{ER}	σ_{ER}	μ_{BA}	σ_{BA}	t_{ER}	t_{BA}
All	<u>47.4</u>	14.7	64.6	11.7	15.12	0.72
Low	35.2	3.0	66.4	11.1	16.59	0.66
High	33.6	6.7	62.0	13.9	0.85	5.06
Low-to-High	36.0	5.1	63.0	15.4	6.33	0.71
High-to-Low	39.2	12.9	62.4	25.6	14.42	17.10
Self-Adaptive	42.0	23.5	66.8	31.1	8.04	46.04
Decay	<u>47.2</u>	25.6	64.8	24.8	41.36	93.54

Subscripts denote the graph type used as a target. μ & σ denote mean and sample deviation, respectively. t denotes average execution time in milliseconds.

7 DISCUSSION

The results in Table 3 seem to contradict prior results [5, 15], due to a variety of possible factors. Worth noting is that high-level individuals tended to execute in a much lower amount of time than

low-level individuals, even if they represent functionally identical solutions. Some effort to reduce the overhead of primitive-to-primitive communication may be necessary in order to help low-level individuals execute more rapidly.

With parsimony pressure, the *Self-Adaptive* scheme stands out as being very prone to premature convergence. In particular, most runs would eventually converge to having their entire population at the higher level. This is likely due to the fact that mutating a high-level individual in that scheme causes decomposition, which dramatically increases execution time. Since execution time is used as an objective, this means that mutation is heavily penalized unless it also mutates to immediately improve the other objective values. Removing parsimony pressure resolved this convergence.

It appears that decomposition slows execution times down compared to not decomposing, even when the schemes used high-level primitives more often. This may be explained by low-level GP finding viable solutions that execute more rapidly than high-level solutions' decompositions. When done in the existing population, as in *High-to-Low* and *Decay*, this effect is magnified. This is likely due to GP's inability to account for the slow-down by preferring the higher-level representation. On the other hand, such preference for a higher-level representation would highly bias the search, as in *Self-Adaptive*, which decomposed in reproduction. This implies decomposition will inherently be detrimental in one of two ways: in reproduction it will highly penalize children, or in the existing population it will greatly slow execution times. Worse, if the in-population decomposition is not applied equally at all times, as in *Decay*, it penalizes existing individuals relative to each other.

One potential solution to the problem of decomposition penalizing children is to perform execution by making a copy of the individual's parse tree using any decompositions available to reduce it to the lowest possible level; then, execute on that copy. This would completely negate the implicit preference for higher-level primitives, at the cost of performing every execution at a slower low-level representation and requiring that all high-level primitives have exact decompositions. This was how Pope's original works implemented higher-level primitives, suffering no setbacks due to decomposition [5, 15]. However, it seems unavoidable that schemes which perform decomposition will be slower than schemes that do not.

Alternatively, efforts could be focused on methods which permit decomposed individuals to exist for some time without being compared to non-decomposed individuals – generally some method for segregating the population. This would allow decomposed individuals to live while still submitting them to parsimony pressure. Furthering this notion is the fact that *Self-Adaptive* and *Decay* performed so poorly – they had access to all primitives at all times, making them not much different from *All*. As prior work has shown that schemes like this should perform poorly in general, this is something that should be avoided [2]. Segregation is a promising method for improving this. Pope did this with his self-adaptive scheme, which performed well [5, 15]. In particular, a hierarchical island model could be used which allows for thorough exploration of each granularity level in parallel, which feed into mixed-granularity islands. Methods for determining

which granularity level deserved the most computational resources could even be used, to improve DPGC's scalability.

8 CONCLUSION

The findings here suggest that dynamic primitive granularity control, while likely to be a viable route for the improvement of GP as shown in previous experiments [5, 15], introduces several unique design hurdles. There are certain traditional GP conventions which negatively impact DPGC, including parsimony pressure and a single population. Further, tree-based GP requires significant modifications to facilitate information passing from parents to children in order to properly emulate high-level primitives with low-level primitives.

It is apparent that, in order to derive value from DPGC, the ways in which it differs from convention need to be considered in order to make deliberate design decisions, rather than simply adding DPGC on top of canonical GP. This may prevent widespread adoption of DPGC unless simple and effective alternatives are found, though it remains a promising, if labor-intensive, method of improving GP's capabilities, provided that the GP is implemented with DPGC's unique design considerations in mind.

9 FUTURE WORK

One possible way to solve the issues caused by decomposition is to send recently-decomposed individuals to a segregated population, then reintroduce them to the primary population after some time. One may hypothesize that island models would work well with DPGC – both categorize the population and attempt to leverage that categorization in a way that helps the evolutionary process. An island model where each island represents a granularity level could potentially allow for thorough exploration of the search space represented by the primitives at each level, while helping to avoid the penalization of freshly-decomposed individuals. Further, island models could improve DPGC's scalability by allowing for a selective allocation of computational resources, somewhat mitigating the performance impact of exploratory decomposition.

Another way to resolve the issue of parsimony pressure is to implement the pressure in less harsh forms. For example, an asynchronous implementation where rapid execution is incentivized by the ability to reproduce more often, but individuals are not explicitly penalized or rewarded for execution time.

An important finding by Pope in his experiments regarding DPGC has been that it tends to not improve fitness much on problems that are already easily solved by GP. As such, it is important to consider that the problems tackled here were replicating some of the simplest graph generators available. More complex graphs should be used for future testing. In addition, more difficult objectives (such as various measures of vertex centrality, or a more robust statistical test for the degree distribution) should be added to ensure the graphs more accurately match the target.

The time required to implement high-level primitives is dramatically increased due to the desire to add decomposition mappings. It should be possible to allow GP itself to create decomposition mappings automatically – provide the set of lower-level primitives and attempt to emulate the desired higher-level

primitive. This should be relatively simple to accomplish, as the problem is already represented in a fashion that GP can solve.

Further, such a decomposition mapping might potentially avoid some rather difficult-to-find bugs. For example, *add-edges-with-prob* had a flawed decomposition mapping due to an easily-overlooked subtlety of its implementation. The proper decomposition is shown in Fig. 2. Since the doubly-nested *for-vertex-in-list* primitives will iterate over each unique pair of different vertices twice, p must be changed to accommodate this. This sort of change to the iterating behavior is likely to be a feature of many primitives' decomposition mappings. In such cases, probabilities p need to be replaced with $1 - \sqrt[k]{1-p}$ where k is the number of times they will be evaluated. This is a rather complicated and easy-to-forget rule for a human developer to obey while creating decomposition mappings, but GP with access to the necessary arithmetic operators should be able to replicate this behavior very easily. Other decompositions may require even more complex and unintuitive alterations, making automatic mappings even more valuable. Even if only used to generate a starting point for a human to base a proper implementation on, the evolutionary process would make these subtle requirements more noticeable.

A much more difficult (if perhaps more powerful) way to solve this issue of developing higher-level primitives and their decomposition mappings is to implement a way for higher-level primitives to be dynamically created during evolution, as discussed in Section 3. Such macro-primitives would be created by identifying useful subtrees found in the population. On its own this would lessen the need for manually-implemented primitives, and such macro-primitives could even be used in subsequent runs to reduce this need further (though this could introduce severe bias). This could certainly be implemented on its own without DPGC, and would be a valid avenue for improving performance (though likely for an inordinate amount of effort), as shown in [9]. DPGC would logically make macro-primitives even more useful however, since they by definition have a distinct level of granularity which makes them markedly different from lower-level primitives.

Figure 2: A correct decomposition mapping for *add-edges-with-prob*.

10 ACKNOWLEDGEMENTS

This work was supported by Los Alamos National Laboratory via the Cyber Security Sciences Institute under subcontract 570204.

REFERENCES

- [1] Aaron S. Pope, Daniel R. Tauritz, and Alexander D. Kent. 2016. Evolving random graph generators: A case for increased algorithmic primitive granularity. *IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE.
- [2] Matthew A. Martin and Daniel R. Tauritz. 2015. Hyper-Heuristics: A Study On Increasing Primitive-Space. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO Companion '15)*. Association for Computing Machinery, New York, NY, USA, 1051–1058.
- [3] Brian W. Goldman and Daniel R. Tauritz. 2011. Meta-evolved empirical evidence of the effectiveness of dynamic parameters. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation (GECCO '11)*. Association for Computing Machinery, New York, NY, USA.
- [4] Adam Harter, Aaron S. Pope, Daniel R. Tauritz, and Chris Rawlings. 2019. Empirical evidence of the effectiveness of primitive granularity control for hyper-heuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 1478–1486.
- [5] Aaron S. Pope, "The Automated Design of Network Graph Algorithms with Applications in Cybersecurity" Chapter 7, pp. 110-138. PhD diss, Auburn University, 2019
- [6] David J. Montana. "Strongly Typed Genetic Programming," *Evolutionary Computation*, vol. 3 issue 2, pp 199-230, June 1995.
- [7] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," in *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182-197, April 2002.
- [8] Walker M (2001) Introduction to genetic programming. Tech Np: University of Montana, Missoula
- [9] John R. Koza. "Genetic Programming II: Automatic Discovery of Reusable Programs.," in *Artificial Life*, vol. 1, no. 4, pp. 439-441, July 1994.
- [10] Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. 2010. Open issues in genetic programming. *Genetic Programming and Evolvable Machines* 11, 3–4 (September 2010), 339–363. DOI:<https://doi.org/10.1007/s10710-010-9113-2>
- [11] L. Spector, Evolving control structures with automatically defined macros. In *Working Notes for the AAAI Symposium on Genetic Programming (MIT, Cambridge, MA, USA, 10–12 Nov 1995)*, ed. by E.V. Siegel, J.R. Koza (AAAI, 1995), pp. 99–105
- [12] J.P. Rosca, Towards automatic discovery of building blocks in genetic programming. In *Working Notes for the AAAI Symposium on Genetic Programming (AAAI, 1995)*, pp. 78–85
- [13] I. Jonyer, A. Himes, Improving modularity in genetic programming using graph-based data mining. In *Proceedings of the 19th International Florida Artificial Intelligence Research Society Conference (Melbourne Beach, FL, USA, May 11–13 2006)*, ed. by G.C.J. Sutcliffe, R.G. Goebel (American Association for Artificial Intelligence, 2006), pp. 556–561
- [14] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*. Published via <http://lulu.com> and freely available at <http://www.gp-field-guide.org.uk>, 2008. (With contributions by J. R. Koza).
- [15] Aaron S. Pope and Daniel R. Tauritz. 2020. Automated Design of Multi-Level Network Partitioning Heuristics Employing Self-Adaptive Primitive Granularity Control. In *Genetic and Evolutionary Computation Conference (GECCO '20)*, July 8–12, 2020, Cancún, Mexico. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3377930.3389819>