Improving Genetic Programming with Novel Exploration - Exploitation Control

Jonathan Kelly[1], Erik Hemberg[1], and Una-May O'Reilly[1]

MIT, Cambridge MA 02139, USA
jgkelly@mit.edu, {hembergerik,unamay}@csail.mit.edu

Abstract. Low population diversity is recognized as a factor in premature convergence of evolutionary algorithms. We investigate program synthesis performance via grammatical evolution. We focus on novelty search – substituting the conventional search objective – based on synthesis quality, with a novelty objective. This prompts us to introduce a new selection method named `knobelty`. It parametrically balances exploration and exploitation by creating a mixed population of parents. One subset is chosen based on performance quality and the other subset is chosen based on diversity. Three versions of this method, two that adaptively tune balance during evolution solve program synthesis problems more accurately, faster and with less duplication than grammatical evolution with lexicase selection.

Keywords: program synthesis · novelty · diversity

1 Introduction

Program synthesis is an open, relevant and challenging problem domain within genetic programming, (GP) [1]. Synthesis of introductory programming problems has been tackled using methods such as PUSHGP [2] and Grammar Guided Genetic Programming (G3P) [3]. These methods have, to date, not been able to solve the complete suite of program synthesis benchmark problems introduced in [1]. One possible explanation is convergence to local optima. Premature convergence is often correlated with low solution diversity. The population becomes concentrated within a small part of the solution space and crossover and mutation operators yield only sub-optimal solutions. In GP, for example, diversity has been narrowly examined in lexicase selection studies solving program synthesis [4]. In this paper, we present a broader study of diversity in GP with Grammatical Evolution (GE) and program synthesis. Our central question is: **Can diversity be controlled within GP to improve synthesis performance and prevent premature convergence?**

An approach called novelty search, introduced in [5] provides us with partial inspiration. In novelty search, rather counterintuitively, selection is altered so its objective is not to select a solution of superior "quality", e.g. performance on synthesis, but of higher novelty, using a score calculated by measuring how

different a solution is from others. Over the course of a search completely biased by novelty, untimely convergence is circumvented and solutions of better quality can be coincidentally identified as side effects, despite quality being ignored by selection, see [5–7]. Novelty search has a demonstrated track record in benchmark tunably deceptive problems and academic problems but there are differences between these domains and program synthesis. In GP, on robot controller, symbolic regression and genetic improvement, novelty search results are mixed [7, 8, 6].

Our study finds that some distance measures used in "pure" novelty search can struggle to find solutions that perform synthesis well. But, it reveals an intuitive similarity between lexicase selection and novelty search that considers output novelty. Time plots of novelty search point to a new hypothesis stating that a search relying upon a population which has members that are good at synthesis and others that are novel, will yield better solutions. Distinctly, this hypothesis does NOT propose a population composed of members that *individually* combine novelty and synthesis performance. Instead, it seeks a population of mixed composition.

To validate the hypothesis empirically, the study proposes a form of tunable selection that we call `knobelty`. The name `knobelty`, a porte-manteau of *novelty* and *knob*, conveys that the selection method has a parameterized threshold (vis. *knob*) that controls the likelihood of a novelty selection objective being used vs a performance-based one. In expectation, `knobelty` populates a population of parents, some fraction of which have been selected by a novelty measure, and the rest by synthesis performance. The study evaluates three versions of `knobelty` that differ by how the threshold between novelty and performance is controlled. The base case splits between novelty and performance selection using a static threshold, unchanged over a run. Two others vary how much novelty is selected for *during* evolution, i.e. dynamically. One decreases the threshold using a decreasing exponential function sensitive to time. The other is sensitive to duplication. Every generation, it re-adjusts the threshold according to the duplication that arises after selection, mutation and crossover. The study proceeds with a representative subset of program synthesis problems and reports the accuracy, speed, and efficiency of `knobelty`, concluding it is successful and worthy of more study.

The structure of the paper is as follows, Section 2 has background. Our methods are in Section 3. Experiments, results and discussion are in Section 4. Finally, Section 5 has conclusions and future work.

2 Background

The foundations of this study are grammar based GP – in the form of Grammatical Evolution, program synthesis and search convergence analysis through the lens of diversity and novelty search. We present background for each topic in this section.

2.1 Grammar Based Genetic Programming

Grammatical Evolution (GE) is a genetic programming algorithm, where a BNF-style, context free grammar is used in the genotype to phenotype mapping process [9]. A grammar provides flexibility because the solution language can be changed without changing the rest of the GP system. Different grammars, for the same language, can also be chosen to guide search bias. The genotype-phenotype mapping allows variation operators, crossover and mutation, to work on the genotype (an integer vector), or the derivation tree that is intermediate to the genotype and phenotype. Following natural evolution, selection is based on phenotype behavior, i.e. performance of program on required task. A drawback of GE is lack of locality [10].

A context free grammar (CFG) is a four-tuple $G = \langle N, \Sigma, R, S \rangle$, where: *1)* N is a finite non-empty set of non-terminal symbols. *2)* Σ is a finite non-empty set of terminal symbols and $N \cap \Sigma = \emptyset$, the empty set. *3)* R is a finite set of production rules of the form $R : N \mapsto V^* : A \mapsto \alpha$ or (A, α) where $A \in N$ and $\alpha \in V^*$. V^* is the set of all strings constructed from $N \cup \Sigma$ and $R \subseteq N \times V^*$, $R \neq \emptyset$. *4)* S is the start symbol, $S \in N$ [11].

GE uses a sequence of (many-to-one) mappings to transition from a genotype to its fitness:

1) **Genotype**: An integer sequence

2) **Derivation Tree**: Each integer in the genotype controls production rule selection in the grammar. This generates a rule production sequence which can be represented as a derivation tree

Program/Phenotype: The leaves (terminals) of the derivation tree, which together form the sentence. For example, in the program synthesis domain this is the generated program.

3) **Fitness**: For each test case, a value is assigned measuring the distance between the desired outputs and program outputs when executed with it. For a solution, fitness summarizes this value for all test cases. It can be a scalar statistic, e.g. sum, or a bit vector for each test case's success test.

2.2 Program Synthesis

In GP, program synthesis is formulated as an optimization problem: find a program q from a domain Q that minimizes combined error on a set of input-output cases $[X, Y]^N, x \in X, y \in Y$. Typically an indicator function measures error on a single case: $\mathbb{1}$: $q(x) \neq y$.

Initial forays into program synthesis considered specific programming techniques, such as recursion, lambda abstractions and reflection, or languages such as C or C++, or problems such as caching, exact integer and distributed algorithms, [12–18]. Other approaches consider implicit fitness sharing, co-solvability, trace convergence analysis, pattern-guided program synthesis, and behavioral archives of subprograms [19].

Subsequently, a general program synthesis benchmark suite of 29 problems systematically selected from sources of introductory computer science programming problems became available to the GP community [1]. Multiple studies,

many with PUSHGP, have drawn on this suite, e.g. a study of different selection operators, in particular one named lexicase selection [4, 20]. Diversity was measured in this work as a means of explaining why lexicase selection works. The goal of lexicase selection can be summarized as promoting into the next generation, parents that collectively solve different test cases. In this work, our baseline algorithm, `GE_Perf` and the performance selection modules of `knobelty` algorithms use lexicase selection. Referencing the same benchmarks G3P attempts to present a general grammar suitable for arbitrary program synthesis problems [21]. Another G3P effort analyses test set generalization [3].

2.3 Search Convergence

Solution discovery can be hindered by convergence to local optima. Diversity and behavioral novelty are two methods to address this.

Diversity Early diversity related work in genetic algorithms includes "crowding" [22, 23] where a solution is compared bitwise to a randomly drawn subpopulation and replaces the most similar member among the subpopulation. Later studies enforce some sort of solution niching. Separation of the population by age is done with ALPS [24]. Spatial separation of solutions is used in coevolutionary learning [25]. Behavioral information distance sustains diversity on the Tartarus problem [26]. In GP, diversity measures and diversity's correlation with fitness were studied as early as [27], and diversity has continually been demonstrated to play an important role in premature convergence [28]. Surprisingly, [29] showed that even variable-length GP trees can still converge genotypically.

Interestingly, bloat confounds tree distance. When trees get bigger, but do not change behaviorally because of bloat, distance becomes nonsensical, see [30]. This indicates bloat must be under control if we use diversity to guide the search. Overall, past efforts show that it is rare to use diversity to guide search. In addition, there is also limited work on diversity and program synthesis.

Novelty Search Novelty search [5] is one approach to overcome convergence and lack of solution diversity. In pure novelty search, there is absolutely no selection pressure based on performance. The method uses a distance measure (defined over 2 solutions), a novelty measure defined for a solution (summarizing many distance measurements to others), a management policy for the memory holding solutions that the search has to date generated (from which distance/novelty will be calculated) and a selection objective maximizing novelty. Novelty search stresses selecting for novelty in a *behavioral* domain. The most intuitive representation of behavior in GP has been explored – program outputs. Since programs and derivation trees express behavior through statically analyzable semantic information, alternate representations remain to be explored. This study, for example, uses distance measures over GE programs and derivation trees. It also implements an updated memory management policy that uses an unbounded cache and sampling for novelty approximation.

GP explorations with novelty search [8] are comparable on different axes: problem domains and new methods. Problem domains include robot controller navigation, symbolic regression, classification among others [7, 8, 6, 31, 32]. New methods include crowding selection methods and a weighted combined fitness and novelty score [33]. Similarly, [7] combine diversity and performance into one objective convergence in a robot controller problem.

The background implies that there are research opportunities at the intersection of GP using grammars, program synthesis and novelty search. The next section introduces the methods we develop for this investigation.

3 Method

We proceed in three phases. In the first phase we evaluate novelty during GE program synthesis evolution. Methodologically this requires a distance measure (see 3.1), a novelty measure and a statistical summarization of novelty for a population (see 3.2).

3.1 Our Distance Measures

A distance measure is defined between two solutions $i, j, d(i, j) \in \mathbb{R}$. It describes how far apart solutions are, in some basis. We measure distance using each of the representations in GE that map from genotype to phenotype, plus outputs:

1) Genotype: We measure with Hamming distance.

2) Derivation Tree (DT): Distance is a common and different nodes count. Ignoring structure, we collapse the tree structure into node counts. We measure distance as the Euclidean norm of the nodes common to both trees plus how many nodes are different.

3) Phenotype: We measure with (Levenshtein[1]) string edit distance and divide all measurements by the size of the largest phenotype.

4) Output distance: We record the success of each test case in a binary vector cell. Output distance is the Hamming distance between the two vectors.

3.2 How We Measure Novelty

A novelty measure of a GE solution j is based on its pairwise distances to a set of programs C. We average these pairwise distances to obtain a scalar novelty value.

Conceptually C must function as a memory structure. One option is to make it an archive that is finite and selectively updated depending on some entry and retirement policy. A policy typically has multiple threshold parameters which makes it complex to manage. Alternatively, C can be an "infinite" cache but the cost of computing distance from j to every solution in a cache is computationally expensive.

[1] `https://en.wikipedia.org/wiki/String_metric`

Algorithm 1 $A(i, d, C, N)$ **Novelty approximation**
Parameters: I Individual, d distance measure, $\mathbf{C}$ cache, N sample size

1: $n \leftarrow 0$ ▷ Initial novelty
2: **for** $i \in [1, \ldots, N]$ **do** ▷ Draw individuals to compare against
3: $O \leftarrow\sim \mathcal{U}(\mathbf{C})$ ▷ Uniform sampling with replacement
4: $n \leftarrow n + d(I, O)$ ▷ Get distance
5: **return** n/N ▷ Return average distance

We propose to simplify C and streamline its computation. We record every unique encountered individual in C. To approximate the novelty of j, we draw N samples (with replacement) from C and average $d(k, j)$ over them. To choose N, we scan a range of sample sizes and choose a value that is stable under many draws. Whereas the extreme case of a bottomless cache scales $O(P^2T^2)$, (T is the number of generations and P is the population size), sampling reduces the complexity to $O(NPT)$. Algorithm 1 shows our approximate novelty calculation.

3.3 Knobelty Selection

We observe that convergence must be influenced by dual forces: the diversity of solutions in the population and the performance or quality of solutions in the population. The former fosters explorative search and the latter exploitive search; at issue is how to juggle these. They are not always conflicting so a multi-objective framing is inappropriate. A weighted score balancing each of them could be used as fitness but this would not explicitly yield either good performers or highly diverse solutions but solutions in between. We propose, therefore, to control this balance between exploration and exploration by creating a mixed parent pool. One subset will be selected based on novelty and the other based on performance. A parameterized threshold (knob) $\kappa \in [0, 1]$ choosing between novelty and performance selection will determine the subsets' sizes in expectation. We call this `Knobelty` Selection. Our hypothesis is that the decrease in fitness selection pressure and increase in novelty selection pressure will prevent convergence to local optima without severely degrading the efficiency for finding a global optima. Algorithm 4 supports three control methods for κ:

`Static` Keep κ constant. (line 3)

`Gen_Adapt` Change κ every generation, $\kappa(t) = 2^{-\lambda t}$, t =generation, using an exponential schedule to initially boost novelty and then afterwards allow the population to slowly converge.

`Dup_Adapt` Change κ according to duplication in the population. We initialize κ and, realizing crossover and mutation can disrupt the balance between novelty and performance selection, we check the duplication ratio of the population and adjust κ to $\kappa(t) = 1 - |\{x|x, y \in P, x \neq y\}|/|P|$.

See Algorithms 2, 3, and 4 for more details. With these distance measures, novelty definitions and `knobelty` selection we proceed to experimental evaluation.

Algorithm 2 $S_\nu(P, C)$ **Novelty Based Selection**
Parameters: P Population, C cache
Local: ω tournament size, d distance measure

```
τ ← S_t(ω, P)          ▷ Randomly choose competitors for a tournament
for i ∈ τ do           ▷ Calculate novelty of each competitor
    i_ν ← A(i, d, C, N)   ▷ Approximate novelty, see Alg. 1
return max(τ_ν)        ▷ Pick most novel competitor
```

Algorithm 3 $S(P, C, \kappa)$ **Knobelty Selection**
Parameters: P population, C cache, κ novelty probability
Global: S_π Performance selection function, S_ν Novelty selection function

```
P' ← ∅                 ▷ New population
for i ∈ [1, ..., |P|] do   ▷ Select an Individual
    k ←~ U([0, 1])     ▷ Uniform random value
    if k < κ then      ▷ Get selection measurement
        P' ← P' ∪ S_π(P, C)   ▷ Performance based lexicase selection
    else
        P' ← P' ∪ S_ν(P, C)   ▷ Novelty based selection Alg. 2
return P'              ▷ Return new population
```

4 Experiments

This section presents experimental setup, and results and discussion.

4.1 Setup

In order to focus on the potential of explicit diversity control, we selected a small subset of problems from the general programming synthesis benchmark suite [1]. As our intent was not to match the previous standards set by related work done by PUSHGP[20] or G3P[3], but rather to explore the effect of explicit diversity control, we decided to use a fitness evaluation budget of 3.0×10^4, a budget that is less than one sixth of the budget used by PUSHGP, and one tenth the budget used in G3P. In doing so, we restrict the number of problems that our system is able to solve, but we significantly increase our investigative agility. To choose which problems to use in our experiments, we first ran a series of tests with the decreased fitness evaluations on many of the benchmark problems, and chose three that provided a range of difficulty. These tests were done with GE and lexicase selection. We selected one easy – Median (MED), another moderately difficult – Smallest (SML), and a third hard – String Lengths Backwards (SLB). Per convention, the I/O dataset is split in two, one used during evolution, the training set, and one for out of sample testing, the testing set. For each of our selected problems, the training set consists of 100 test cases and the testing set consists of 1000 test cases.

Algorithm 4 Grammatical Evolution with Knobelty Selection
Parameters: KC: knob control method ∈ `Static`, `Dup_Adapt`,`Gen_Adapt`

1: $P \leftarrow \iota()$ ▷ Initialize population
2: $C \leftarrow \emptyset$ ▷ Initialize cache
3: **if** KC = `Static` **then** ▷ **Static** κ
4: $\kappa \leftarrow k$ ▷ Set κ to a static value
5: $f(g(P))$ ▷ Map and evaluate population
6: $C \leftarrow C \cup P$ ▷ Add population to cache
7: **for** $t \in [1, \ldots, T]$ **do** ▷ Iterate over generations
8: **if** KC = `Gen_Adapt` **then** ▷ **Generation based κ update**
9: $\kappa \leftarrow 2^{-\lambda t}$ ▷ Update κ based
10: **if** KC = `Dup_Adapt` **then** ▷ **Duplication sensitive κ adaptation**
11: $\kappa \leftarrow \Delta(\text{inefficiency}(P), \text{inefficiency}(P_{t-1}))$ ▷ Update κ, see Sec 3.3
12: $P' \leftarrow S(P, C, \kappa)$ ▷ **Knobelty Selection, Alg. 3**
13: $P' \leftarrow \chi(P')$ ▷ Crossover individuals
14: $P' \leftarrow \mu(P')$ ▷ Mutate individuals
15: $f(g(P'))$ ▷ Map and evaluate population
16: $C \leftarrow C \cup P'$ ▷ Add population to cache
17: $P \leftarrow P'$ ▷ Replace population
18: **return** $max(C)$ ▷ **Return best performing solution**

We report results on 100 runs. We report program synthesis performance in terms of how many runs out of 100 resulted in one or more programs that solved all the out of sample (test) cases. All other reported values are averages over 100 runs. We ran all experiments on a cloud VM with 24 cores, 24GB of RAM, and 16GB of disk.

To determine an efficient population to generation ratio, we swept the ratios while keeping fitness evaluations constant and found that a population size of 1500 with 20 generations produced better results on all three problems than other ratios. This contrasts significantly with the population to generation ratio that PUSHGP and G3P use, with our ratio of population size:generations 1500:20 = 75:1 vs PUSHGP and G3P of 1000:300 = 3.3:1. We believe that this is another example of diversity having an impact on performance. When choosing the original population the seeding operator is able to effectively space out individuals throughout the search space. Our results imply that this high initial diversity followed by a small number of generations to evolve is more effective than a smaller and thus less diverse initial population that has more generations to evolve.

Our implementation originates from the grammar based genetic programming repository PonyGE2 [34]. Building on PONYGE2, we added lexicase selection to create a conventional GE algorithm for program synthesis that uses lexicase selection. This algorithm, `GE_Perf`, uses performance based selection. We then designed and developed our various `knobelty` algorithm variants (see

Table 1: Baseline performance for different GP variants on the MED, SLB and SML program synthesis problems. N.B. GE uses an order of magnitude lower fitness evaluation budget.

	Test Performance		
Heuristic	**MED**	**SLB**	**SML**
GE_Perf	85	8	74
G3P [3]	79	68	94
PUSHGP [20]	55	94	100

Algorithm 4 and Table 3)[2]. The set of parameters we used throughout all our experiments is listed in Table 2.

Table 2: Experimental settings

Parameter	**Value**
Codon size	100,000
Elite size	15 ($0.01P$)
Replacement	generational
Initialisation	PI grow
Init genome length	200
Max genome length	500
Max init tree depth	10
Max tree depth	17
Max tree nodes	250
Max wraps	0
Crossover	single point
Crossover probability	0.9
Mutate duplicates	False
Mutation	int flip
Mutation probability	0.05
Novelty archive sample size (C)	100
Novelty tournament size (ω)	7

Table 1 presents the program synthesis performance of our baseline algorithm `GE_Perf` with those reported by PUSHGP and G3P, for each of the three test cases we chose. All three algorithms use lexicase selection. The "Perf" is implicit in PUSHGP and G3P, as they are both entirely concerned with performance. We make it explicit in `GE_Perf` since we will later use GE for our `knobelty` algorithms.

Regarding the test performance of the baselines, bear in mind that `GE_Perf` uses between 1/6th and 1/10th fitness evaluations per run compared to PUSHGP and G3P. With this handicap, it performs moderately worse on SML, significantly worse in SLB, but is actually able to outperform the other approaches on

[2] The code is available at `https://github.com/samnovelty/noveltyprogsys`

MED. From this point forward, we will solely use the results from `GE_Perf` as our baseline.

Table 3: `knobelty` algorithm variants, see Algorithm 4

Abbreviation	Explanation
`GenoNovelty`	GE with `Knobelty` selection using genotype novelty approximation
`DTreeNovelty`	GE with `Knobelty` selection using derivation tree novelty approximation
`PhenoNovelty`	GE with `Knobelty` selection using phenotype novelty approximation
`OutputsNovelty`	GE witn `Knobelty` selection using standard outputs novelty approximation

All `knobelty` algorithm variants approximate novelty with Algorithm 1. One of its parameters is the distance measure it uses. We abbreviate the variants by this distance measure, see Table 3. The approximation's sampling size of 100 was experimentally set by a sweep that identified the lowest size that is stable over 1000 sample tests. The tournament size of `knobelty` selection tournaments is $\omega = 7$. For the exponentially decreasing novelty, we used $\lambda =$ Number of Generations$/10 = 2$. We did no experimentation with the range of λ.

Experimental Approach We proceed in two steps.

1. We set $\kappa = 1.0$ and run `OutputsNovelty` to closely approximate the spirit of the original novelty[5]. We consider how this compares to `GE_Perf`, our baseline. Then, with $\kappa = 1.0$, we try `DTreeNovelty`, `PhenoNovelty`, `GenoNovelty` and compare to `GE_Perf` and `OutputsNovelty`.
2. We then use `DTreeNovelty` with `Static` knob control to conduct a sensitivity analysis of κ by sweeping it across a range of values for the MED problem. We look at program size, duplication, best performance and novelty. Then we run all three problems (MED, SLB and SML) using two novelty algorithms - `DTreeNovelty`, `OutputsNovelty`, with 3 knob controls – `Static`, `Gen_Adapt` and `Dup_Adapt`.

4.2 Results

Proceeding with step 1, Table 4 compares `GE_Perf` to `PhenoNovelty`, `GenoNovelty`, `DTreeNovelty` abd `OutputsNovelty`, run with a κ of 1.0, i.e. "pure" novelty. From these results, we can draw three insights. The first is that using pure novelty search with `DTreeNovelty`, `PhenoNovelty`, and `GenoNovelty` is unsuccessful. This observation is arguably to be expected. Genotypes in GE do not express behavior. Derivation trees and the programs defined by their leaves express behavior, but there is a many to one mapping between program and output behavior, so the search space of derivation trees and program overwhelms search based on novelty. The second observation is that `OutputsNovelty`, in contrast,

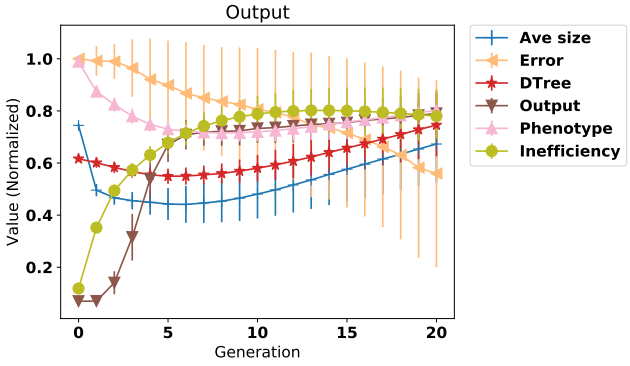
does reasonably well, beating the baseline on two of the three problems. This can be explained by the observation that, while `OutputsNovelty` is not directly related to fitness, taking the Hamming Distance between two solutions' binary test case error vectors expresses their relative performance. Novelty search is favoring different but mutually compatible solutions that could be combined successfully with crossover. In fact, Figure 1, which plots the average novelty at each generation for `OutputsNovelty` (1a), `DTreeNovelty` (1b), and `GE_Perf` (1c), shows an inverse correlation between the output novelty and error trajectories. Since lower values of error are better, we observe that increasing `OutputsNovelty` also relates to better fitness. This finding is strengthened because for two out of three test cases, `OutputsNovelty` leads to better performance than `GE_Perf`.

Table 4 and Figure 1 present *inefficiency*. Inefficiency is the ratio of the number of duplicate solutions to the product of number of fitness evaluations and generations. The third insight from Table 4 is that the inefficiency of `GE_Perf` is significantly higher then that of `PhenoNovelty` and `DTreeNovelty`. This means that `GE_Perf` can't generate novel solutions, while searching based on novelty can. While pure `DTreeNovelty` or `PhenoNovelty` fail, their ability to effectively explore the search space motivates the exploration of a combination of `GE_Perf` and novelty. Since `DTreeNovelty` is the algorithm that is the least inefficient, and thus explores the most solutions, and `OutputsNovelty` performs well on two of the three problems, we decide to move forward with them in our `knobelty` experiments.

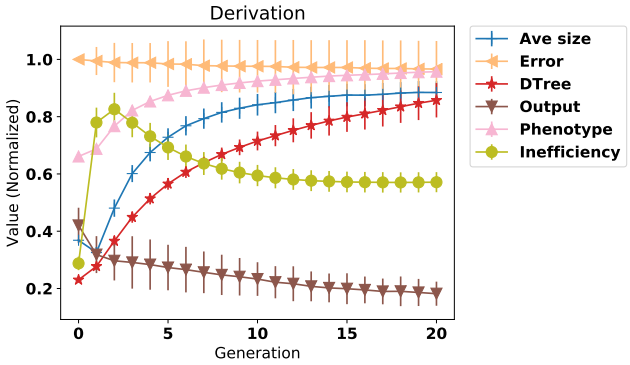
Table 4: (Pure) Novelty Experiments. We set $\kappa = 1.0$ and run `OutputsNovelty`, `DTreeNovelty`, `PhenoNovelty`, `GenoNovelty`. We show `GE_Perf`, our baseline which uses performance based selection, for comparison, above each problem's results.

		Fitness				Novelty (Total)			
Problem	**Distance – `knobelty` Alg**	**Train**	**Test**	**Time**	**Ineff.**	**Geno.**	**Pheno.**	**DT**	**Ave size**
MED	`GE_Perf`	86	85	789.34	71	0.89	0.29	9.73	24.52
MED	genotype – `GenoNovelty`	0	0	848.71	74	1.00	0.27	7.54	13.02
MED	phenotype – `PhenoNovelty`	0	0	1392.13	26	0.99	0.53	16.12	46.88
MED	derivation – `DTreeNovelty`	0	0	785.11	19	0.99	0.47	25.15	62.47
MED	outputs – `OutputsNovelty`	5	5	600.95	64	0.98	0.28	8.78	22.58
SLB	`GE_Perf`	8	8	1446.47	67	0.95	0.22	7.55	22.80
SLB	genotype – `GenoNovelty`	1	1	2039.14	67	1.00	0.18	6.67	14.06
SLB	phenotype – `PhenoNovelty`	0	0	2335.94	36	0.99	0.39	11.85	37.28
SLB	derivation – `DTreeNovelty`	0	0	2890.80	21	0.99	0.34	19.57	51.34
SLB	outputs – `OutputsNovelty`	13	13	2120.16	53	0.98	0.23	7.89	23.78
SML	`GE_Perf`	74	74	1350.96	81	0.93	0.30	8.45	19.68
SML	genotype – `GenoNovelty`	0	0	918.89	73	1.00	0.27	7.53	13.16
SML	phenotype – `PhenoNovelty`	0	0	1019.02	26	0.99	0.53	16.08	47.23
SML	derivation – `DTreeNovelty`	0	0	841.54	19	0.99	0.47	24.91	63.60
SML	outputs – `OutputsNovelty`	97	97	421.48	56	0.96	0.30	10.02	28.56

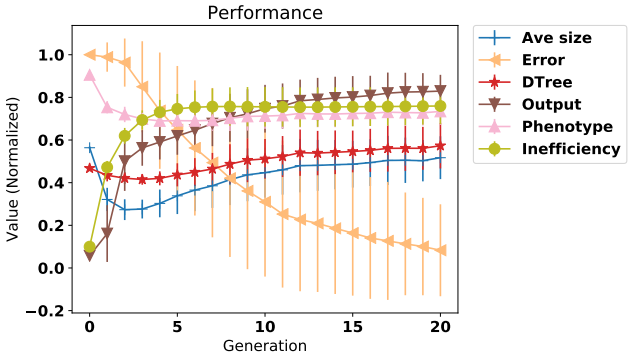
We now proceed with step 2, where we start by analysing `DTreeNovelty` with `Static` knob control. We conduct a sensitivity analysis of κ by sweeping it across a range of values from 0.0 to 1.0 on 0.1 intervals for the MED problem.



(a) MED `OutputsNovelty`



(b) MED `DTreeNovelty`



(c) MED `GE_Perf`

Fig. 1: Measurements per generation for MED. Y-axis is normalized measurement value and x-axis shows generation. Average and standard deviation over 100 runs.

Figure 2 shows the results from these experiments. We can see that with low values of κ the performance improves compared to our baseline `GE_Perf` ($\kappa = 0.0$), confirming our hypothesis that pairing fitness selection with novelty selection can improve performance. The parameter setting of $\kappa = 0.2$ seems to display the best trade-off in performance, efficiency and novelty. As expected, average program size and novelty, specifically `DTreeNovelty`, increases as κ increases, and inefficiency decreases.

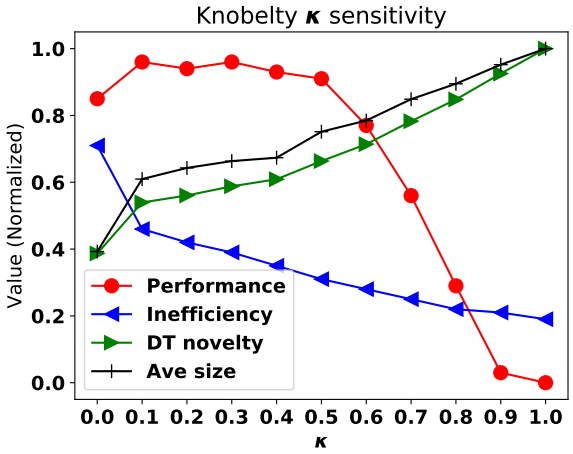


Fig. 2: Sensitivity analysis of κ constant for knobelty selection for the MED problem. Y-axis shows ratio value and x-axis shows κ. One line is performance, one line is DTree novelty, one line is inefficiency. The $\kappa = 0.2$ seems to display the best trade-off in performance, efficieny and novelty

We select $\kappa = 0.2$ to go forward and we run all three problems (MED, SLB and SML) using the two novelty algorithms we picked from analysis of Table 4 – `DTreeNovelty`, `OutputsNovelty`. We ran the experiments with all three ways to set the parameter κ that controls the balance between using novelty based and performance based selection – `Static`, `Gen_Adapt` and `Dup_Adapt`. Table 5 shows the results obtained from these experiments. We see from these results that in all cases, using knobelty selection based on either `DTreeNovelty` or `OutputsNovelty` does better than our `GE_Perf`. We also see that despite the fact that `OutputsNovelty` did significantly better when run with a κ of 1.0 than `DTreeNovelty` (Table 4), when κ is less than 1.0, i.e. when each uses parents selected with a mixture of novelty and performance, they produce similar results. We also see that three of the algorithm plus κ control combinations yield perfect solutions 100% of the time. The `DTreeNovelty` algorithm for `Dup_Adapt` solves SML and MED perfectly, and the `OutputsNovelty` algorithm for `Gen_Adapt` solves SML perfectly. Additionally, on SLB, the number of test cases solved by

just `GE_Perf` is increased by 100 percent by many of the algorithm and control combinations. These results on three problems strongly support our hypothesis that explicit diversity control can play a role in improving performance. They suggest going forward with more problems, and doing a more comprehensive comparison that includes other methods.

Table 5: Two `knobelty` algorithms, one basing novelty on derivation trees and the other on outputs are compared on the three problems for the three different ways to control the `knobelty` knob.

		Test Performance		
Algorithm	**Control of κ**	**MED**	**SLB**	**SML**
`GE_Perf`	-	85	8	74
`DTreeNovelty`	`Static`, $\kappa = 0.2$	+9	+10	+24
`OutputsNovelty`	`Static`, $\kappa = 0.2$	+12	+7	+23
`DTreeNovelty`	`Dup_`Adapt	+15	0	+26
`OutputsNovelty`	`Dup_`Adapt	+12	+13	+22
`DTreeNovelty`	`Gen_Adapt`	+11	+10	+24
`OutputsNovelty`	`Gen_Adapt`	+12	+9	+26

In the next section we conclude and present possible directions for future work.

5 Conclusions & Future Work

The contributions of this paper are:

1. We introduce a computationally tractable approximation of novelty for GP. It samples the cache rather than exhaustively referencing every item in it. This dispenses with a complex cache management policy.

2. We introduce novelty measures on genotype, derivation tree and program representation domains for GE.

3. Using these measures, we explore GE with a conventional performance objective, pure novelty, and `knobelty` for program synthesis.

4. We find evidence that `knobelty` can successfully balance a population's proportions of novel and high performing solutions, thus program synthesis can be improved in performance accuracy, speed and efficiency. Since these successful results are only based on three judiciously chosen problems, further investigation is merited.

There are a number of possible directions for future work. One is to mutate duplicate solutions, to increase the search space visited and drive the inefficiency to zero. Another is to evaluate the `knobelty` operators across more problems, with an increased number of fitness evaluations. These results would then be comparable with the results of PUSHGP and G3P. A third is to try `knobelty` with tree based operators and see how the results compare. In these tree based

experiments, the effect that novelty has on bloat would be especially interesting to monitor, as tree based operators are known to have problems with bloat. Finally it would be interesting to investigate other representations of program behavior, including program traces, and other static program and derivation tree representations.

References

1. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. pp. 1039–1046. ACM (2015)
2. Spector, L.: Autoconstructive evolution: Push, pushgp, and pushpop. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001). vol. 137 (2001)
3. Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: Towards understanding and refining the general program synthesis benchmark suite with genetic programming. In: 2018 IEEE Congress on Evolutionary Computation (CEC). pp. 1–6. IEEE (2018)
4. Helmuth, T., McPhee, N.F., Spector, L.: Lexicase selection for program synthesis: a diversity analysis. In: Genetic Programming Theory and Practice XIII, pp. 151–167. Springer (2016)
5. Lehman, J., Stanley, K.O.: Exploiting open-endedness to solve problems through the search for novelty. In: ALIFE. pp. 329–336 (2008)
6. López-López, V.R., Trujillo, L., Legrand, P.: Novelty search for software improvement of a slam system. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. pp. 1598–1605. ACM (2018)
7. Doucette, J., Heywood, M.I.: Novelty-based fitness: An evaluation under the santa fe trail. In: European Conference on Genetic Programming. pp. 50–61. Springer (2010)
8. Naredo, E.: Genetic Programming Based on Novelty Search. Ph.D. thesis, ITT, Instituto tecnologico de Tijuana (2016)
9. Ryan, C., O'Neill, M., Collins, J.: Introduction to 20 years of grammatical evolution. In: Handbook of Grammatical Evolution, pp. 1–21. Springer (2018)
10. Thorhauer, A., Rothlauf, F.: On the locality of standard search operators in grammatical evolution. In: International Conference on Parallel Problem Solving from Nature. pp. 465–475. Springer (2014)
11. Booth, T.L.: Sequential machines and automata theory (1967)
12. O'Neill, M., Ryan, C.: Evolving multi-line compilable C programs. In: Poli, R., Nordin, P., Langdon, W.B., Fogarty, T.C. (eds.) Genetic Programming, Proceedings of EuroGP'99. LNCS, vol. 1598, pp. 83–92. Springer-Verlag, Goteborg, Sweden (26-27 May 1999)
13. Lucas, S.: Exploiting reflection in object oriented genetic programming. In: Keijzer, M., O'Reilly, U.M., Lucas, S.M., Costa, E., Soule, T. (eds.) Genetic Programming 7th European Conference, EuroGP 2004, Proceedings. LNCS, vol. 3003, pp. 369–378. Springer-Verlag, Coimbra, Portugal (5-7 Apr 2004)
14. Agapitos, A., Lucas, S.M.: Learning recursive functions with object oriented genetic programming. In: Collet, P., Tomassini, M., Ebner, M., Gustafson, S., Ekárt, A. (eds.) Proceedings of the 9th European Conference on Genetic Programming. Lecture Notes in Computer Science, vol. 3905, pp. 166–177. Springer, Budapest, Hungary (10 - 12 Apr 2006)

15. Yu, T., Clack, C.: Recursion, lambda-abstractions and genetic programming. In: Poli, R., Langdon, W.B., Schoenauer, M., Fogarty, T., Banzhaf, W. (eds.) Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming. pp. 26–30. CSRP-98-10, The University of Birmingham, UK, Paris, France (14-15 Apr 1998)
16. Wan, M., Weise, T., Tang, K.: Novel loop structures and the evolution of mathematical algorithms. In: Silva, S., Foster, J.A., Nicolau, M., Machado, P., Giacobini, M. (eds.) Genetic Programming - 14th European Conference, EuroGP 2011, Torino, Italy, April 27-29, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6621, pp. 49–60. Springer (2011), https://doi.org/10.1007/978-3-642-20407-4_5
17. Weise, T., Tang, K.: Evolving distributed algorithms with genetic programming. IEEE Trans. Evolutionary Computation 16(2), 242–265 (2012), https://doi.org/10.1109/TEVC.2011.2112666
18. Weise, T., Wan, M., Tang, K., Yao, X.: Evolving exact integer algorithms with genetic programming. In: 2014 IEEE Congress on Evolutionary Computation (CEC). pp. 1816–1823 (July 2014)
19. Krawiec, K.: Behavioral program synthesis with genetic programming, vol. 618. Springer (2016)
20. Helmuth, T., McPhee, N.F., Spector, L.: Program synthesis using uniform mutation by addition and deletion. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 1127–1134. ACM (2018)
21. Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: A grammar design pattern for arbitrary program synthesis problems in genetic programming. In: European Conference on Genetic Programming. pp. 262–277. Springer (2017)
22. De Jong, K.A.: Analysis of the behavior of a class of genetic adaptive systems (1975)
23. Mengshoel, O.J., Goldberg, D.E.: The crowding approach to niching in genetic algorithms. Evolutionary computation 16(3), 315–354 (2008)
24. Hornby, G.S.: Alps: the age-layered population structure for reducing the problem of premature convergence. In: Proceedings of the 8th annual conference on Genetic and evolutionary computation. pp. 815–822. ACM (2006)
25. Mitchell, M., Thomure, M.D., Williams, N.L.: The role of space in the success of coevolutionary learning. In: Artificial life X: proceedings of the Tenth International Conference on the Simulation and Synthesis of Living Systems. pp. 118–124. MIT Press Cambridge, MA (2006)
26. Gomez, F.J.: Sustaining diversity using behavioral information distance. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation. pp. 113–120. GECCO '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1569901.1569918
27. Burke, E.K., Gustafson, S., Kendall, G.: Diversity in genetic programming: An analysis of measures and correlation with fitness. IEEE Transactions on Evolutionary Computation 8(1), 47–62 (2004)
28. Sudholt, D.: The benefits of population diversity in evolutionary algorithms: A survey of rigorous runtime analyses. arXiv preprint arXiv:1801.10087 (2018)
29. Burks, A.R., Punch, W.F.: An analysis of the genetic marker diversity algorithm for genetic programming. Genetic Programming and Evolvable Machines 18(2), 213–245 (2017)
30. Affenzeller, M., Winkler, S.M., Burlacu, B., Kronberger, G., Kommenda, M., Wagner, S.: Dynamic observation of genotypic and phenotypic diversity for different

symbolic regression gp variants. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. pp. 1553–1558. ACM (2017)
31. Shahrzad, H., Fink, D., Miikkulainen, R.: Enhanced optimization with composite objectives and novelty selection. arXiv preprint arXiv:1803.03744 (2018)
32. Goldsby, H.J., Cheng, B.H.: Automatically discovering properties that specify the latent behavior of uml models. In: International Conference on Model Driven Engineering Languages and Systems. pp. 316–330. Springer (2010)
33. Cuccu, G., Gomez, F.: When novelty is not enough. In: European Conference on the Applications of Evolutionary Computation. pp. 234–243. Springer (2011)
34. Fenton, M., McDermott, J., Fagan, D., Forstenlechner, S., O'Neill, M., Hemberg, E.: Ponyge2: Grammatical evolution in python. CoRR abs/1703.08535 (2017), http://arxiv.org/abs/1703.08535