Investigating Genetic Programming with Novelty and Domain Knowledge for Program Synthesis

by

Jonathan Gregory Kelly

S.B. Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author .....
Department of Electrical Engineering and Computer Science
May 24, 2019

Certified by .....
Una-May O'Reilly
Principal Research Scientist
Thesis Supervisor

Certified by .....
Erik Hemberg
Research Scientist
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chairman, Master of Engineering Thesis Committee

Investigating Genetic Programming with Novelty and Domain Knowledge for Program Synthesis

by

Jonathan Gregory Kelly

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Low population diversity is recognized as a factor in premature convergence of evolutionary algorithms. We investigate program synthesis performance via grammatical evolution using 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` which parametrically balances exploration and exploitation. We also recognize that programmers solve coding problems with the support of both programming and problem specific knowledge. We attempt to transfer insights from such human expertise to genetic programming (GP) for solving automatic program synthesis. We draw upon manual and non-GP Artificial Intelligence methods to extract knowledge from synthesis problem definitions to guide the construction of the grammar that Grammatical Evolution uses and to supplement its fitness function. Additionally, we investigate the compounding impact of this knowledge and novelty search. The resulting approaches exhibit improvements in accuracy on a majority of problems in the field's benchmark suite of program synthesis problems.

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist

Thesis Supervisor: Erik Hemberg
Title: Research Scientist

Acknowledgments

I would like to give a huge thank you to Erik Hemberg and Una-May O'Reilly for their seemingly endless patience, anecdotes, guidance, puns (Erik), and for the opportunity to work in such a great lab. I have learned so much and have so many good memories in the last two years, many of which are due to them and the whole ALFA lab. I would also like to thank the rest of the lab, and especially Shashank Srikant, Abdullah Al-Dujaili, and Nicole Hoffman for all that they've done for me in my time working with ALFA. Finally I'd like to thank my parents and family for their endless support, without which I'd have never have made it through MIT.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Program synthesis is an open, relevant and challenging problem domain within genetic programming, (GP) [15]. It is also an important challenge for the GP community to work on [29, 31] because it is arguable that any progress GP can contribute to program synthesis will emerge from important advancements in GP. Further, this progress will increase GP's relevance to Artificial Intelligence (AI) and machine learning. Recently, a program synthesis benchmark suite [15] has focused efforts around a common set of problems and has enabled results to be compared and reproduced. Multiple efforts, referencing the suite, have resulted in noteworthy advancements. However no methods have, to date, been able to solve the complete suite of program synthesis benchmark problems mentioned above.

One possible explanation to the question of why program synthesis has been such a challenging problem, is convergence to local optima during search. 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 this work, we present a broader study of diversity in GP with Grammatical Evolution (GE) and program synthesis.

Another possible explanation to what elevates program synthesis to such a high challenge status, is the the exceptional level of human expertise that is required by programming. Programming is rated a super human intelligence task by [35] and the nature of the extraordinary expertise it requires has been distilled into the notion

of computational thinking which is now taught in schools. While it is difficult for a machine to be trained with the same level of background knowledge that a student has, it is reasonable to provide it all the information present in the programming challenge itself. To achieve this, we investigate different ways to provide the domain knowledge present in programming questions to our algorithms working to generate solutions to these questions.

1.1 Background

1.1.1 Novelty Search

An approach called novelty search, introduced in [20] provides us with partial inspiration in how to combat low solution diversity. In novelty search, rather counter-intuitively, 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 [20, 23, 8]. 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 [8, 27, 23].

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. We evaluate `knobelty` using the above techniques, and conclude that it is successful on at least a narrow subset of the benchmark problems. We note that both the crossover and mutation operators have a large impact on the overall performance of our search, and so next work on optimizing the combination of crossover, mutation, and selection (novelty search) operators. We also note that a prominent problem with novelty search, as well as tree based crossover and mutation operators is bloat. Bloat is the unnecessary increase of solution size, which often ends up leading to worse performance. We combat bloat in a number of different ways, and find that limiting solution size leads to a boost in performance.

1.1.2 Domain Knowledge

As mentioned above our goal is to improve GP's performance on program synthesis. Today, some GP problems are harder to solve than others. To probe this, we hand coded solutions to the benchmark problems and set up a grammar providing the functionality of a general purpose programming language. We then calculated how hard it would be to find each of these hand coded solutions by going through the grammar and selecting each of the production choices necessary. By multiplying the probability of each chosen production choice we are able to give an estimate of the search space size for each problem, which can be seen in the second column of Table 1.1. One immediately notices that the hand-coded solutions to problems with magic numbers or string constants, e.g. problems Grade and Small or Large, are not just hard to find, but the likelihoods are almost nil. This exercise merely quantifies information already known to the GP program synthesis community. It is nonetheless also thought provoking because working with such constant values is not difficult for a student completing the problems. When we added the constants to the grammar and

Table 1.1: Search space size estimates for different problem grammars. Values estimated by computing the probability that a random set of grammar production choices will generate a simple and correct solution for each problem.

Problem	**Basic Grammar**	**Human Grammar**
Checksum	1e11	1e6
Compare String Lengths	1e7	1e5
Count Odds	1e7	1e5
Digits	1e5	1e5
Double Letters	1e36	1e7
Even Squares	1e10	1e7
For Loop Index	1e7	1e6
Grade	1e17	1e7
Last Index of Zero	1e6	1e5
Median	1e5	1e5
Mirror Image	1e5	1e4
Negative to Zero	1e7	1e5
Number IO	1e4	1e3
Small or Large	1e21	1e5
Smallest	1e5	1e5
String Lengths Backwards	1e7	1e6
Sum of Squares	1e7	1e6
Super Anagrams	1e5	1e5
Syllables	1e10	1e6
Vector Average	1e6	1e5
Vectors Summed	1e7	1e5

pared away language elements not needed to solve it, the occurrence likelihood of the human written solution within the grammar became orders of magnitude greater than the original grammer (Table 1.1, column 3). The results, again expected, nonetheless draw attention to the asymmetry between our assumptions of how a student would solve a programming problem and our expectations of what GP should be able to do. The problem, as it is being presented to GP, is like posing a circuit design problem to a student that requires resistors and other components while insisting the student invent resistors. The abstraction levels of the different tasks, from a programming perspective are vastly different. Expecting GP to find a magic number or constant *already present in the problem definition* is very different from asking it to combine generic, problem-agnostic, program statements into correct control logic.

A teacher does not solve the problem for a student but makes sure the student has read the problem and can relate it to their programming language knowledge. Equivalently, one could ask how well GP can solve the problem, with information

we assume a student would already have? This answer would be helpful for the GP community. It would provide us with some sort of upper bound on how well GP could perform if it had the reasoning power of such a student; we propose this bound be referred to as the *student-information* bound. It expresses GP's performance, given the most knowledge we believe is legitimate for GP to integrate and a state of art algorithm. We compare this bound to previous work.

This exercise further suggests that it is arguably legitimate to *automatically* identify the knowledge about program languages and the problem description and provide it to GP. We therefore work on identifying and incorporating this knowledge that can be automatically extracted from program synthesis problem descriptions using alternate AI methods into an improved Genetic Programming/Grammatical Evolution approach.

1.2 Research Questions

While improving the state of the art on the program synthesis problem was a goal in our research, another major goal was to improve GE by demonstrating the success of new techniques on a challenging problem. With that in mind, the research questions we want to address in this thesis are presented below.

Novelty:

- Can diversity be controlled within GP to improve synthesis performance and prevent premature convergence?
- What is the benefit of having separate search objectives for performance and novelty?
- How can novelty be tuned to avoid bloat?
- Which algorithm settings provide the best program synthesis performance?
- How can legitimate information about a programming language and problem description be transferred automatically to GP? What information is possible

and useful to extract?

- Will GP be able to solve more problems on the benchmark suite with the use of automatically extracted domain knowledge.

Domain Knowledge:

- How can legitimate information about a programming language and problem description be transferred automatically to GP? What information is possible and useful to extract?

- Will GP be able to solve more problems on the benchmark suite with the use of automatically extracted domain knowledge.

1.3 Contributions

With our research questions in mind, the contributions presented in this thesis are outlined below. The first four are discussed in Chapter 3, the following three in Chapter 4, and the final three in Chapter 5.

1. Introduction of 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. Introduction of novelty measures on genotype, derivation tree and program representation domains for GE.

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

4. 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.

5. Addition of a set of modifications to our GE algorithm using `knobelty` to prevent bloat.

6. A sensitivity analysis of the `knobelty` selection operation.

7. The identification of an effective set of crossover, mutation, and selection operators that have been optimized for performance on the program synthesis benchmark.

8. The identification and estimation of the *student-information* bound.

9. The creation of a tool that automatically parses information from problem statements to inform the grammar and fitness function.

10. An evaluation of the best set of parameters, grammars, and fitness functions found in this research with state of the art results on the program synthesis benchmark.

1.4 Structure Of Thesis

The rest of this thesis is structured as follows. Chapter 2 discusses previous work in the fields of genetic programming, program synthesis, novelty search, and automatic incorporation of domain knowledge. Chapter 3 then presents initial work done with diversity, discussing experiments done with novelty search. It also introduces `knobelty` and demonstrates its effect on the program synthesis domain. Chapter 4 then further explores the use of `knobelty`, as well as more complex crossover and mutation operators. It evaluates the effect bloat can have with different sets of operators, and discusses the efficacy of various techniques designed to combat bloat. Chapter 5 discusses a different approach to improving GE for program synthesis, the incorporation of domain knowledge. It presents the results from a grammar that represents the *student-information* bound (described in Section 1.1.2), as well as the results from various automation techniques designed to automatically enhance GEs performance with information parsed from the problem statement. Finally, Chapter 6 discusses the conclusions that can be drawn from this thesis, as well as potential future work.

Chapter 2

Related Work

The foundations of this thesis are grammar based GP – in the form of Grammatical Evolution, program synthesis, search convergence analysis through the lens of diversity and novelty search, and incorporation of domain knowledge. 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 [33]. 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 [37].

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, $S \in N$ [3].

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.

Programs in GE are evaluated like they are in all GP systems. For each test case, a value is assigned based on the distance between the desired outputs and program outputs when executed with the associated inputs. Selection, also following GP, is based on phenotype behavior, i.e. performance of program on required task.

GE's genotype-phenotype mapping implies crossover and mutation operators can operate on the genotype or the derivation tree. One crossover simply exchanges integers between crossover points on the genotype. Here, while the interpretation step raises some issues of locality, the grammar and the rewriting assure crossover will result in syntactically valid offspring [37]. Others operate on the derivation tree [9]

Like in GP, though in grammatical form in GE, the abstraction of functions and terminals, i.e. the language from which solutions are composed, impacts the solutions that constitue the search space. Further, the language *biases* the likelihood of generating solutions within the search space [31]. Biasing, in grammatical terms, can be explained by first introducing the notion of a grammar's structure. The structure of a grammar is its number of different production rules plus the number and values

of each right hand side production in a rule. A grammar's structure interacts with rewriting during decision tree derivation. Because of the specific number of rules and productions, and genotype redundancy, rewriting is implicitly biased to generate some derivations more than others. This bias has large impact on algorithm performance. To alter the likelihood of generating solutions within the search space, one can therefore alter (bias) a GE grammar. Alteration could include adding or subtracting right hand side productions or adding or deleting entire production rules. We will explain how we exploit bias alteration as a means of introducing domain knowledge into GP in Chapter 5.

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, [30, 24, 2, 42, 38, 41, 40]. Other approaches consider implicit fitness sharing, co-solvability, trace convergence analysis, pattern-guided program synthesis, and behavioral archives of subprograms [19].

A watershed moment was the introduction of a program synthesis benchmark suite of 29 problems, systematically selected from sources teaching introductory computer science programming [15]. Three studies are most relevant to this thesi: the original benchmarking done using `PUSHGP` [15], the most recent `PUSHGP` results using various mutation operators [14], and the most recent grammar guided GP efforts [10]. Henceforth we refer to these as $\texttt{PushGP}_{BM}$, $\texttt{PushGP}_{MU}$, `G3P`, and respectively.

Diversity was measured in $\texttt{PushGP}_{BM}$ 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 [11]. Another `G3P` effort analyses test set generalization [10].

Transparency is a desirable property of GP program synthesis solutions in addition to performance. We use *transparency* to refer to human readability and interpretability. While GP solutions are *executably* transparent, achieving this definition of transparency is challenged by bloat. As well, some representations are more transparent than others, e.g. `PUSHGP` code is less transparent (to an average university student) than grammars with rules allowing `Python`-like functions and terminals.

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" [7, 25] 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 [17]. Spatial separation of solutions is used in coevolutionary learning [26]. Behavioral information distance sustains diversity on the Tartarus problem [13]. In GP, diversity measures and diversity's correlation with fitness were studied as early as [4], and diversity has continually been demonstrated to play an important role in premature convergence [36]. Surprisingly, [5] 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 [1]. 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 [20] 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 [27] are comparable on different axes: problem domains and new methods. Problem domains include robot controller navigation, symbolic regression, classification among others [8, 27, 23, 34, 12]. New methods include crowding selection methods and a weighted combined fitness and novelty score [6]. Similarly, [8] combine diversity and performance into one objective convergence in a robot controller problem.

2.4 Domain Knowledge with GP

There are many ways in which domain knowledge can be used in GP, and it is often incorporated subconsciously. When setting up GP to solve a specific problem, humans design the fitness function, determine which function are available to the algorithm, and in the case of GE, set up the grammar. In each of these stages, human

knowledge is integrated. Because GP is an AI method, it is important to be consistent in the inclusion of problem solving knowledge to avoid making certain problems unfairly easier for GP to solve. An example of a consistent inclusion of knowledge is the weighting of AST paths for program repair [39]. Another, considering program synthesis, is the use of different stacks for $\texttt{PushGP}_{BM}$ [15].

Although not explicitly stated, $\texttt{PushGP}_{BM}$, $\texttt{PushGP}_{MU}$ and `G3P` each use varying levels of problem knowledge. For each problem, humans manually decide which data types to use and how to set up the fitness functions. In a slightly less consistent manner, for a certain subset of problems such as `Small or Large` and `Syllables` which are difficult to solve without problem knowledge, specific string keywords that are useful to solve the problem are introduced. These inclusions are very similar to the explicit method we call constant hunting in Section 5.1.2. The information role of GP's fitness function is undeniably important. Multiple studies investigate fitness functions and their impact on the problem at hand, for a survey see e.g. [32].

In summary, there has been work done using novelty search, and much work done on program synthesis, but never has novelty been applied to the program synthesis problem. As generating programs leads to an essentially infinite search space, exploring this cleverly and with as few evaluations as possible is vital, which is why we believe novelty search might be able to help. Additionally, as there is much information available to students in the problem statement itself when doing a coding challenge, we believe it is fair and reasonable to try and provide our programs with that information as well. In fact, not only do we believe that it is reasonable, but we believe that without the information in the problem statement, many problems become realistically impossible to solve. With this in mind, we discuss our methods, experiments, and results in the next three chapters.

Chapter 3

Novel Exploration - Exploitation Control

This chapter discusses a way to explicitly combat diversity, and thus attempt to avoid premature search convergence. We first introduce our methods, outline the experiments and setup, and finally discuss results. The research done in this section was used in a paper that we presented at the EuroGP 2019 conference [18].

3.1 Method

Our method consists of three main parts, a distance measure (see 3.1.1), a novelty measure, and a statistical summarization of novelty for a population (see 3.1.2).

3.1.1 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 made up of a count of the common and different nodes in the derivation tree. Ignoring structure, we collapse the tree structure

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

```
n ← 0                                   ▷ Initial novelty
for i ∈ [1, ..., N] do                  ▷ Draw individuals to compare against
    O ←~ U(C)                           ▷ Uniform sampling with replacement
    n ← n + d(I, O)                     ▷ Get distance
return n/N                              ▷ Return average distance
```

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.

3.1.2 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.

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.

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

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

1: $\tau \leftarrow S_t(\omega, P)$ ▷ Randomly choose competitors for a tournament
2: **for** $i \in \tau$ **do** ▷ Calculate novelty of each competitor
3: $i_\nu \leftarrow A(i, d, C, N)$ ▷ **Approximate novelty, see Alg. 1**
4: **return** $\max(\tau_\nu)$ ▷ Pick most novel competitor

3.1.3 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. We hypothesize 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

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

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

definitions and `knobelty` selection we proceed to experimental evaluation.

3.2 Experiments

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 [15]. As our intent was not to match the previous standards set by related work done by `PushGP` or `G3P`, 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.

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

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 3.1.3
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**

out of sample (test) cases. All other reported values are averages over 100 runs. We ran all experiments on a cloud (`OpenStack`) VM with 24 cores, 24GB of RAM using Intel(R) Xeon(R) CPU E5-2450 v2 @ 2.50GHz.

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 [9]. Building on PonyGE2, we added lexicase selection to create

Table 3.1: 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` [10]	79	68	94
`PushGP` [14]	55	94	100

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 Algorithm 4 and Table 3.3)[2]. The set of parameters we used throughout all our experiments is listed in Table 3.2.

Table 3.1 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 MED. From this point forward, we will solely use the results from `GE_Perf` as our baseline.

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 3.3. 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 $\lambda = \text{Number of Generations}/10 = 2$. We did no experimentation with the range of λ.

[2]The code is available at `https://github.com/flexgp/novelty-prog-sys`

Table 3.2: 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
Population size (P)	1,500
Number of generations	20

Table 3.3: `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

3.2.1 Experimental Approach

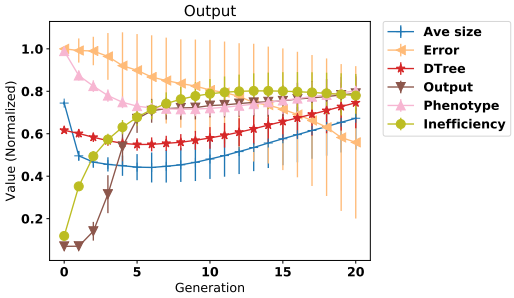
We proceed in two steps.

1. We set $\kappa = 1.0$ and run `OutputsNovelty` to closely approximate the spirit of the original novelty[20]. 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`.

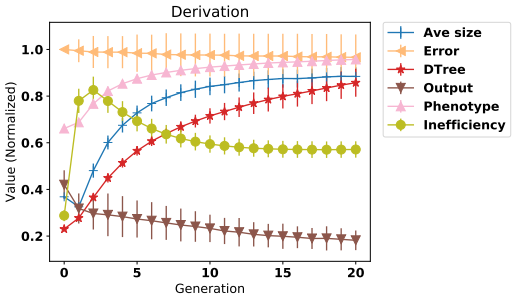
2. We then use `DTreeNovelty` with `Static` knob control to conduct a sensitivity analysis of κ by sweeping it across a range of values for the MED problem. We look at program size, duplication, best performance and novelty. Then we run all three problems (MED, SLB and SML) using two novelty algorithms - `DTreeNovelty`, `OutputsNovelty`, with 3 knob controls – `Static`, `Gen_Adapt` and `Dup_Adapt`.

3.3 Results and Discussion

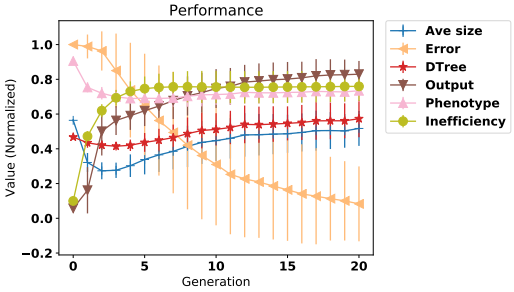
Proceeding with step 1, Table 3.4 compares `GE_Perf` to `PhenoNovelty`, `GenoNovelty`, `DTreeNovelty` abd `OutputsNovelty`, run with a κ of 1.0, i.e. "pure" novelty. From these results, we can draw three insights. The first is that using pure novelty search with `DTreeNovelty`, `PhenoNovelty`, and `GenoNovelty` 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 3-1, which plots



(a) MED `OutputsNovelty`



(b) MED `DTreeNovelty`



(c) MED `GE_Perf`

Figure 3-1: Measurements per generation for MED. Y-axis is normalized measurement value and x-axis shows generation. Average and standard deviation over 100 runs.

the average novelty at each generation for `OutputsNovelty` (3-1a), `DTreeNovelty` (3-1b), and `GE_Perf` (3-1c), 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 3.4 and Figure 3-1 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 3.4 is that the inefficiency of `GE_Perf`

Table 3.4: (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

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.

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. Figure 3-2 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.

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 3.4 – `DTreeNovelty`, `OutputsNovelty`. We ran the experiments with all three ways

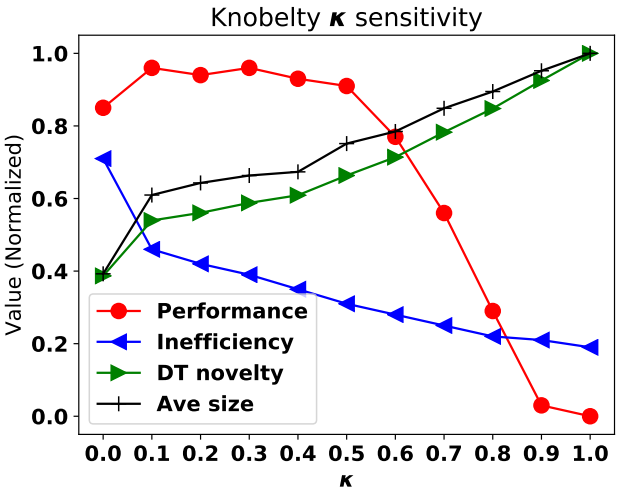


Figure 3-2: Sensitivity analysis of κ constant for knobelty selection for the MED problem. Y-axis shows ratio value and x-axis shows κ. One line is performance, one line is DTree novelty, one line is inefficiency. The $\kappa = 0.2$ seems to display the best trade-off in performance, efficieny and novelty

to set the parameter κ that controls the balance between using novelty based and performance based selection – `Static`, `Gen_Adapt` and `Dup_Adapt`. Table 3.5 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 3.4), 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.

We have learned that while using only novelty to drive search is not effective to increase performance, it does reduce inefficiency implying it is able to explore more search space. With this insight in mind, we were able to create a knob that balances novelty search and traditional performance driven search. We evaluate `knobelty` on

Table 3.5: 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

three problems, and our results 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. This additional and more comprehensive study of `knobelty` is presented in the next chapter.

Chapter 4

Refinement and Enhancement of Novelty Search

In this chapter we further investigate the effect `knobelty` can have on the program synthesis problem. Specifically we focus on the use of tree based crossover and mutation operators, and handling bloating. We first introduce our methods, outline the experiments and setup, and finally discuss results.

4.1 Method

After establishing in Chapter 3 that using `knobelty` search can be an effective technique in the program synthesis domain, we proceed to evaluate it with more complex operators across a wider range of problems. We first evaluate how effective novelty search is when using tree based operators, as well as different combinations of tree based and genomic operators. Next, as bloat is a known issue with tree based operators and with novelty search, we explore methods of limiting solution size. Finally, we evaluate the best set of parameters using novelty search on the entire suite of problems from the program synthesis benchmark in order to compare with previous work.

4.1.1 Duplication

One further area of investigation is the effect that duplicate solutions have during evolution. Theoretically, one way to increase novelty is to keep track of previous solutions in a cache and whenever a new solution is generated, check to see if it is unique. If it is not unique, meaning it matches something in the cache, continue to mutate it until it is unique. We call this technique mutating duplicates, since we continue to perform the mutation operator on any solution that is a duplicate of a previously discovered solution, and test its effect when searching with `knobelty` as well as with pure lexicase selection.

4.1.2 Handling Bloat

A known side effect of novelty search is its tendency to cause solutions to bloat, as solutions that are longer appear to be more novel [22, 21]. Additionally, tree based operators also have a tendency to bloat, generating solutions that are on average much larger than when using genomic operators. In our work, we look at handling bloat in four main ways, each described below.

Lexicase with Size Penalty

We first limit bloat by making a minor alteration in the lexicase algorithm. During normal lexicase selection, if a number of solutions perform equally on all test cases, a random solution from those remaining will be selected. We alter this to penalize solution size, by always choosing the solution that has the smallest number of tree nodes, rather than by random selection. S are the solutions with equal fitness and $|x|$ is the tree-node size of x, so $s = \min_{x \in S} |x|$.

Penalize fitness score

We also investigate a more harsh method of limiting size of solutions, by directly penalizing the fitness score. We do this by first normalizing the number of nodes, by dividing the number of nodes an individual has by the max number of nodes specified

in the algorithmic parameter. We then add this normalized value to the individuals fitness score for each test case. In doing so, during lexicase selection individuals with a smaller number of nodes will more aggressively be selected. This new fitness score can be represented as $g(x) = |x|/C + f(x)$ where x is an individual in the population and C is the algorithms max node size parameter.

Max Node Size

Another very easy way of limiting bloat is by reducing the max size of individual solutions. To do this, we directly reduce the max tree node parameter, and evaluate the effect that that has on solution size, as well as overall performance.

Best Solution chosen by Size

The final way we attempt to limit bloat, is in our choice of the best solution. In the basic implementation of `PonyGE` for program synthesis, the first individual to solve the training cases with zero training error will be the individual that eventually is selected to be run on the test cases. This means that even if other unique solutions are discovered in later evolutions that also solve the training cases with zero training error, they will be thrown out. We change this such that in the case that two generated solutions perform equally well on the training cases, the solution with the smallest number of tree nodes is chosen. In doing so, we hope to limit the number of solutions that timeout on the test cases by choosing the solution with the least amount of extraneous code.

4.2 Experiments

As discussed in Chapter 2, program synthesis has been a topic of study for decades and the suite of program synthesis questions has been a performance benchmark for much recent work. Thus, to further understand the effect that novelty search can have we not only experiment with different operators and bloat handling techniques, we also work with some different problems then the experiments done in Chapter 3.

In particular, in the subsequent experiments we focus on five problems: SLB (String Lengths Backwards), LIZ (Last Index of Zero), SOL (Small or Large), FLI (For Loop Index), and GRD (Grade). For a full description of the problems, refer to [15].

4.2.1 Setup

Just as in the previous chapter, we report experimental 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. Similar to our previous experiments, we ran all experiments on a cloud (`OpenStack`) VM with 24 cores, 24GB of RAM using Intel(R) Xeon(R) CPU E5-2450 v2 @ 2.50GHz.

Our implementation originates from the grammar based genetic programming repository `PonyGE2` [9]. 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 use derivation tree `knobelty` as described in Section 3.1.3, and use an exponentially decaying knob as this combination was shown to be effective in the previous chapter[1]. The set of parameters we used throughout all our experiments is the same as in the previous chapter in Table 3.2, with a few exceptions. As described in the experiments below, we use tree based crossover and mutation operators instead of the genomic ones used in Chapter 3. The other exception is that aside from the first preliminary experiment with results in Table 4.1, we set mutate duplicates to True and thus don't allow individuals that have been previously seen to appear in future populations.

As listed in the table of parameters, 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 $\omega = 7$.

[1]The code is available at `https://github.com/flexgp/novelty-prog-sys`

Table 4.1: Efficiency Experiments – Genomic operators vs Tree based operators.

Problem	Distance	Innef.	Geno.	Pheno.	DT	Ave size
SOL Genomic	`GE_Perf`	52.56	0.94	0.30	10.43	32.53
SOL Genomic	`knobelty`	39.56	0.97	0.33	12.05	34.84
SOL Tree	`GE_Perf`	9.27	0.93	0.50	31.68	199.54
SOL Tree	`knobelty`	8.11	0.96	0.55	41.89	204.59

4.3 Results and Discussion

4.3.1 Tree based Operators

In our work in Chapter 3, `knobelty` was shown to be effective when paired with genomic operators. While this is promising, tree based algorithms have been shown to be effective in the program synthesis space [11, 10]. Thus, to better understand the impact that novelty search can have, we evaluate `knobelty` when using tree based crossover and mutation operators. In a preliminary set of experiments run with tree based operators on the problem `Small or Large`, we quickly learn a few interesting things. The experiment was run with both genomic and tree based operators, and with and without `knobelty`, and the results are in Table 4.1. From this initial experiment we can see that the inefficiency when using tree based operators is greatly reduced, meaning that the algorithm is rarely generating individuals that have been seen in previous generations. This is a good sign for performance, as it means that more unique individuals have been explored. Another result we are able to quickly see is that the average size of individuals when using tree based operators is drastically larger, approximately six times as large. This demonstrates that indeed the tree based operators are causing solutions to bloat. Lastly we can see that in addition to an increase in efficiency and an increase in solution size, there's also a large increase in total novelty.

With these results in mind, we go on to test the performance of tree based operators in comparison to genomic operators, as well as a comparison between purely performance driven search, and search with `knobelty`. To evaluate, we choose the five test problems described above, and test with `GE_Perf` as well as with `knobelty`. The results are in Table 4.2. From this set of experiments we again learn many useful

Table 4.2: Novelty experiments using tree based operators instead of genomic operators.

		Fitness		Novelty (Total)			
Problem	**Distance**	**Train**	**Test**	**Geno.**	**Pheno.**	**DT**	**Ave size**
SLB	`GE_Perf`	33	33	0.91	0.44	27.70	158.06
SLB	`Knobelty`	23	20	0.95	0.50	38.40	164.56
LIZ	`GE_Perf`	8	8	0.94	0.50	29.69	162.15
LIZ	`Knobelty`	3	3	0.97	0.55	37.35	169.08
SOL	`GE_Perf`	3	3	0.93	0.50	32.09	201.65
SOL	`Knobelty`	0	0	0.95	0.55	41.75	205.71
FLI	`GE_Perf`	2	1	0.96	0.53	30.69	188.06
FLI	`Knobelty`	0	0	0.97	0.55	36.70	180.42
GRD	`GE_Perf`	0	0	0.94	0.51	33.48	195.41
GRD	`Knobelty`	0	0	0.96	0.55	42.52	190.41

insights. The first is that in comparison with the genomic operators (not shown in the table), tree based operators perform significantly better. While this is a positive result, we also see that the use of `knobelty` does strictly worse than `GE_Perf`, contradicting our previous hypothesis that `knobelty` could improve performance. We also see again that that average size of the solution is very large, and that even when not driven by novelty search directly, `GE_Perf` has very high levels of total novelty. We believe that due to the bloating of solutions, there is already so much diversity in the population that spending evaluations on novelty search to increase diversity ends up being a waste and thus detrimental.

Before we tackle bloat, we test the combination of a tree based crossover operator with a genomic mutation operator, and a tree based mutation operator with a genomic based crossover operator. This can help us determine an optimal set of operators, as well as inform us about which operators affect performance and bloating the most. We test using four of the problems from the previous set of experiments, and the results are in Table 4.3. While we see that none of the combinations do better than using subtree operators for both crossover and mutation, we do learn that the mutation operator is the operator that's driving the excessive solution bloat.

4.3.2 Handling Bloat

We hypothesize that bloat can lead to sub-optimal performance. One potential reason that bloat might be hurting performance is because the number of training cases is

Table 4.3: Combining Genomic and Tree operators. Run while enforcing a size penalty during lexicase selection (same as in Table 4.4) as described in Section 4.1.2.

Problem	Distance	Operators	Train	Test	Ave size
SLB	GE_Perf	CG MS	37	35	86.73
SLB	GE_Perf	CS MG	43	34	26.13
SLB	Knobelty	CG MS	39	34	83.07
SLB	Knobelty	CS MG	33	33	28.96
SOL	GE_Perf	CG MS	1	1	114.77
SOL	GE_Perf	CS MG	0	0	23.39
SOL	Knobelty	CG MS	1	1	112.74
SOL	Knobelty	CS MG	1	1	26.23
FLI	GE_Perf	CG MS	2	0	119.04
FLI	GE_Perf	CS MG	0	0	38.8
FLI	Knobelty	CG MS	0	0	109.36
FLI	Knobelty	CS MG	0	0	39.77
GRD	GE_Perf	CG MS	1	1	105.28
GRD	GE_Perf	CS MG	0	0	12.43
GRD	Knobelty	CG MS	0	0	105.50
GRD	Knobelty	CS MG	0	0	14.89

Table 4.4: Size penalty experiments. Penalized during lexicase selection phase as described in Section 4.1.2

Problem	Distance	Train	Test	Ave size
SLB	GE_Perf	40	40	100.16
SLB	Knobelty	45	43	94.98
LIZ	GE_Perf	2	2	122.34
LIZ	Knobelty	8	6	131.91
SOL	GE_Perf	3	3	124.73
SOL	Knobelty	5	5	151.55
FLI	GE_Perf	1	1	156.13
FLI	Knobelty	2	1	153.50
GRD	GE_Perf	0	0	122.24
GRD	Knobelty	2	2	132.39

significantly less than the number of test cases, and so bloat might cause solutions to pass the training cases but then timeout on the test cases due to unnecessary looping. In essence, this is GP's version of overfitting. Thus, we evaluate the effect that bloat has by limiting it in a number of different ways. The first way is through penalizing the size during lexicase selection, which we evaluated on the five test problems. The results are in Table 4.4. We find that with this size penalty not only are solutions using `GE_Perf` generally improved, but now the use of `knobelty` also becomes beneficial once again. This helps to confirm our hypothesis that novelty search can be a valuable tool in effectively exploring the search space, we just needed to reduce the noise generated by the bloat to levels where novelty search is not being drowned out.

While this first set of bloat control experiments is promising, it leads to the ques-

Table 4.5: Extreme size penalty experiments. Penalized by adding size to fitness score as in as described in Section 4.1.2

Problem	Distance	Train	Test	Ave size
SLB	`GE_Perf`	33	31	87.47
SLB	`Knobelty`	37	37	87.77
SOL	`GE_Perf`	0	0	96.36
SOL	`Knobelty`	0	0	97.89
FLI	`GE_Perf`	0	0	97.76
FLI	`Knobelty`	1	1	103.21
GRD	`GE_Perf`	0	0	105.31
GRD	`Knobelty`	2	2	105.28

tion of whether an even stricter size penalty would create even better results. To answer this question, we introduce a stricter size penalty, one applied to the fitness score of an individual. The results from four of the problems are displayed in Table 4.5. In these results we see that while once again it leads to `knobelty` being more successful than `GE_Perf`, the size penalty is too much and the results are worse than when using the weaker size penalty applied during lexicase selection.

The last experiments we did in order to combat bloat were directly limiting size in the algorithms parameters, and always storing the smallest individual as the best. The first of these two, directly limiting size, led to significantly worse performance on all five problems. The second of the two led to more nuanced results. While it didn't improve the training performance, it was able to improve the algorithms ability to generalize to the test set in some cases. This demonstrates that the algorithm does at times find multiple correct solutions, and that some of the correct solutions are superior to others. While it doesn't lead to an improvement in what solutions are found, it still is a good technique to use to generate higher quality solutions.

We have seen that tree based operators are in general more effective than genomic ones, but that they have the unfortunate side effect of bloating. To combat this we evaluated a few techniques, and discovered that a penalty during lexicase selection works the best. Finally, we see that `knobelty` is again an effective technique when used appropriately with tree based operators. While this is another step towards solving the program synthesis problem, we see that there is still much work to do. The next chapter takes another step with the incorporation of domain knowledge.

Chapter 5

Incorporation of Domain Knowledge

This chapter discusses the effect domain knowledge can have on the programming synthesis problem, and discusses techniques to automatically incorporate information from the problem statement to better inform search. We first introduce our methods, and then outline the experiments and setup. Finally we present the results and discuss our findings. The research done in this section was used in a paper that we presented at the GECCO 2019 conference [16].

5.1 Method

We present our methods in the following order. We start, in Section 5.1.1, by describing our base grammatical evolution program synthesis algorithm, `PSGP`. Our hypothesis is that there is knowledge that can be automatically extracted from program synthesis problem descriptions using alternate AI methods that can be successfully integrated into `PSGP`. In Section 5.1.2 we discuss how legitimate domain knowledge – programming language and problem description information, can be transferred. We then, in Section 5.1.2, present the methods by which we hunt out specific information from problem descriptions and specifically change `PSGP`.

5.1.1 Program Synthesis Algorithm: PSGP

An overview of PSGP is shown in Algorithm 5. Our PSGP implementation originates from the grammar based genetic programming repository of PonyGE2 [9]. Building on PonyGE2, we have added lexicase selection and a knobelty selection operator, defined in Section 3.1.3. Our code and all grammars are available [1].

We introduce minor adaptations to the knobelty selection operator, due to the results from Chapter 4. First, sometimes a variation operator will generate a new solution that matches the cache sample. Rather than discard this solution, we continue to mutate it until it is novel, per [28]. Second, as discuessed in the previous chapter, GE with tree based operators and knobelty can cause bloat. To combat this, PSGP gives a preference to smaller solutions during lexicase selection and additionally when promoting elites to the next generation, if two solutions solve the same number of test cases, we choose the one with the smallest number of tree nodes. These techniques are further described in Section 4.1.

The algorithm includes a data structure we call a knowledge base. It represents that domain knowledge that has been extracted by any of our knowledge extraction methods. We introduce a new function named createGrammar that has logic which translates information in the knowledge base into grammar changes. We will elaborate on these, where appropriate, in each knowledge extraction method.

5.1.2 Problem Description Knowledge Extraction

We prepared PSGP to run with a variety of grammars. Its base grammar was prepared using an automatic grammar generator and has the basic functionality of Python. It is able to produce solutions to any coding problem, given the datatypes (e.g. boolean, float) that are used in the problem. This is similar to the grammar of [10]. We call the combination of PSGP with this grammar, variant G_{Base}.

This base grammar is intended to reflect the background programming knowledge a programmer has before reading the problem. When the programmer proceeds to

[1] http://github.com/flexgp/programsynthesishunting

Algorithm 5 `PSGP`(D, I, K, Θ) **Program Synthesis GP**
Parameters: D test cases, I problem statement, K knowledge base, Θ hyper parameters

$G_P \leftarrow$ createGrammar(I, K) ▷ Input grammar
$F_R \leftarrow$ getFitnessFunction(I, K) ▷ Set up fitness function
$P \leftarrow$ initialize(Θ, G_P) ▷ Intialize population
$P \leftarrow$ evaluate(P, Θ, F_R) ▷ Evaluate pop fitness
$B \leftarrow \{x : x \in P, x_f = |D|\}$ ▷ Save perfect solutions, x_f is the fitness of a solution x
for $t \in [1, \ldots, \Theta_T]$ **do** ▷ Iterate over generations
    $P' \leftarrow$ selection(P, Θ) ▷ Select new population
    $P' \leftarrow$ variation(P', G_P, Θ) ▷ Subtree mutation and crossover without duplicates
    $P' \leftarrow$ evaluate(P', Θ, F_R) ▷ Evaluate pop fitness
    $B \leftarrow B \cup \{x : x \in P', x_f = |D|\}$ ▷ Save perfect solutions
    $P \leftarrow$ replacement(P', Θ) ▷ Update population
$i^* \leftarrow \min(\{|x| : x \in B\})$ ▷ Retrieve smallest solution
return i^* ▷ Return best solution

read the problem, we presume this helps them to effectively eliminate large parts of the language that will not be necessary so they can focus only on utilizing program elements that are relevant to the problem. As an example, one of the problems on the program synthesis benchmark reads [15]:

```
Small or Large (Q 4.6.3):
Given an integer n, print "small" if n < 1000 and "large"
if n >= 2000 (and nothing if 1000 <= n < 2000).
```

We presume that when programmers read this, they recognize that they will not need certain programming elements, e.g. the `#` or the built in `Python strip()` function. Instead, they recognize their program needs to work with the strings `"small"` and `"large"`, the numbers `1000` and `2000` and the `<` and $\geq$ operators. Effectively we presume some sort of computational thinking that results in new abstractions that are more specific to solving the problem and the winnowing of primitive abstractions from which the solution is not expected to be composed.

In the introduction, we also describe this more minimal grammar, G_{Human}. It is prepared manually by removing any unnecessary string, character and integer productions, and unnecessary comparison operators. For some problems, python functions such as `"strip"` or `"insert"` are also removed if they are not useful. By doing this,

we create a grammar that is able to start a problem in a similar way to that of a human programmer.

How `PSGP` transfers knowledge: Given `PSGP` uses grammatical evolution, our strategy for introducing domain knowledge is to change G_{Base} by altering its bias. As we discuss in Section 2.1, this changes the likelihood of solutions being generated during rewriting. We either explicitly increase the probability of selecting productions that domain knowledge indicates are useful by adding them to the grammar or we reward solutions that contain certain terminals when fitness is calculated.

What domain knowledge is transferred: Humans retrieve information from the problem description that is invaluable to solving the problem. In each of the methods that follow we extract some proposed units of knowledge from the problem description, given some knowledge of programming languages. Our following variants currently vary in their degree of automation. We reasonably assume that their manual efforts could be automated. Should a method prove useful and not be fully automated, we plan to go forward and automate that aspect.

Hunting for Words that Map to Built-In Functions

Variant $G_{Base+W2F}$ recognizes that there is a set of text words and phrases that map directly to built-in language functions. For example, the text word `"minimum"` maps to the `min` function. We initialize the `knowledge base` with these mappings and, completely automatically, parse the problem description for text words or phrases in the `knowledge base`. After parsing, the `createGrammar` function modifies G_{Base} by increasing the likelihood of productions that have a corresponding text word or phrase in the problem description. For example, if the word `"length"` is found in the text, the grammar is automatically updated to have a higher probability of the `len` function. Despite the possibility that the parsing is imperfect, because `createGrammar` only increases the probability that certain productions are picked and does not explicitly reduce the probability of any production, the modified grammar always retains its ability to express a correct solution.

Hunting for Constants, Operators and What Not To Use

While hunting for a set of text words that match built-in functions is helpful, it does not help with words in the description that cannot be anticipated. This is an issue in problems such as `Small or Large` (above) where string constants or magic number constants appear in the description. A student programmer would directly declare: `"small"`, `"large"`, `1000` and `2000` as constants. However, with a general purpose language such as G_{Base}, `PSGP` must find these constants. The probability of generating them is very low, making problems with constants essentially impossible to solve. This is the "invent resistors while circuit designing" issue. To avoid it, variant $G_{Base+COW}$ integrates knowledge of constants. For each problem (at this point) we manually identify the constants in the problem text and, add them to our `knowledge base`. Then in the `createGrammar` function they are automatically added to G_{Base}. For example in the problem `Small or Large` the grammar production rule for strings starts out as:

```
<str_c_part>::=<str_c_part><str_lit>|<str_lit>
```

This is able to generate any possible string, as `<str_c_part>` points to all the ASCII characters. However, after the identification of the string constants and the subsequent automatic insertion into the grammar, this production rule in the grammar simply becomes:

```
<str_c_part>::="small"|"large"
```

Additionally, because we know that $G_{Base+W2F}$ is an imperfect parse, in $G_{Base+COW}$ we also look directly for necessary arithmetic and relational operators in the problem description. Then, and different from $G_{Base+W2F}$, `createGrammar` will automatically *remove* from G_{Base} any arithmetic and relational operators that are *not* identified to be in the problem description. This simultaneously tells $G_{Base+COW}$ what program elements are unnecessary while increasing the likelihood that useful operators end up in solutions.

Hunting for Fitness Function Knowledge

Grammatical evolution, as with any evolutionary algorithm, depends heavily on its fitness function to identify good solutions. Thus, problems where the fitness function has little ability to judge the quality of the candidate solution until it gets very close to a correct solution are often difficult to solve. Since the fitness functions generally used with program synthesis problems only look at the output of an individual's code, problems that return boolean values or a counter become challenging to solve. An example of a problem like this is [15]:

```
Compare String Lengths (Q 4.11.13):
Given three strings n1, n2, and n3, return true
if length(n1) < length(n2) < length(n3),
and false otherwise.
```

To try and increase fitness information, we add a second component to the fitness function that directly inspects the program for trivial program elements that must be included in any correct solution. While we currently manually identify these elements, we believe that it would be possible to parse them automatically from the problem description. As an example, for the `Compare String Lengths` problem (above), we have identified `len`, `<`, `True`, and `False` as necessary correct solution elements. During fitness evaluation, this variant, named $G_{Human+FF}$, inspects the program and increases the fitness score for every listed element it finds.

5.2 Experiments

As aforementioned, since [15] introduced the suite of program synthesis benchmark questions, performance of many program synthesis algorithms has been tested against this benchmark. Of the 29 problems introduced in the benchmark, 25 of them have received the most attention. The problems named `Collatz Numbers`, `String Differences`, `Wallis Pi`, and `Word Stats` have proven to be so challenging with current techniques [15, 10] that they are not attempted. Of the remaining 25 problems,

we focus on 21. We do not work with `Pig Latin`, `X-Word Lines`, `Replace Space with Newlines`, and `Scrabble Score` because our grammar does not currently support splitting strings, joining strings, dictionaries, or newline characters.

5.2.1 Setup

As in previous experiments, we report results on 100 runs and 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 with the same compute setup listed in the experimental setup section of the previous chapters.

The set of static parameters we use throughout all our experiments is listed in Table 5.1. When we use a fitness budget of 30,000 evaluations, we use a population size of 1,500 and run for 20 generations. This ratio of population to generation was reported to be effective in the experiments of Chapter 3. As in those experiments, we decided to use a reduced budget of 30,000 fitness evaluations mainly to increase investigative agility and to recognize the impact of additional domain knowledge.

When using `knobelty`, the archive's sampling size of 100 was experimentally set by a sweep that identified the lowest size that is stable over 1,000 sample tests. The tournament size of `knobelty` selection tournaments is $\omega = 7$. We measure novelty based on the derivation tree, and used an exponentially decaying knob. For the exponential decay, we set λ to be Generations/10, as that was shown to be effective in the results from Chapter 3.

We use the variants of `PSGP` listed in Table 5.2.

5.3 Results and Discussion

This section reports the impact on performance given the varying degrees of domain knowledge our variants integrate into `PSGP` and compares them to previous work.

We start by manually creating a grammar that was minimally powerful to express correct solutions. Understanding how GP performs using this grammar, G_{Human},

Table 5.1: Experimental settings for `PSGP`

Parameter	**Value**
Generations	20 (60 $G_{Human+FF+N}$)
Population Size (P)	1,500 (5,000 $G_{Human+FF+N}$)
Elite size	$0.01P$
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 probability	0.9
Mutate duplicates	True
`Knobelty` archive sample size (C)	100
`Knobelty` tournament size (ω)	7
`Knobelty` function	Exponential [18]
`Knobelty` λ	Generations/10

Table 5.2: Variant abbreviations

Abbreviation	**Variant**
G_{Base}	Base grammar
$G_{Base+W2F}$	Words to Functions (Section 5.1.2)
$G_{Base+COW}$	Constants, operators and what not to use (Section 5.1.2)
G_{Human}	Human's grammar (Section 5.1.2)
$G_{Human+N}$	Human's grammar and `knobelty` operator (Section 3.1.3)
$G_{Human+FF}$	Human's grammar and augmented fitness function (see Section 5.1.2)
$G_{Human+FF+N}$	Human's grammar, augmented fitness function and `knobelty` operator

Table 5.3: Results on 21 benchmark problems using human domain knowledge. The number of runs (out of 100) that solved all test cases is reported. The best results are **bold**. G_{Human} run with a total of 30,000 fitness evaluations. $\texttt{PushGP}_{MU}$ [14], $\texttt{PushGP}_{BM}$ [15], G3P [10] use a total of 300,000 fitness evaluations.

Problem	$\texttt{PushGP}_{MU}$	G3P	$\texttt{PushGP}_{BM}$	G_{Human}
Checksum	**5**	0	1	2
Compare String Lengths	42	2	5	**60**
Count Odds	20	12	5	**22**
Digits	19	0	10	**79**
Double Letters	**20**	0	1	0
Even Squares	0	**1**	0	**1**
For Loop Index	2	**8**	0	6
Grade	1	**31**	1	7
Last Index of Zero	**72**	22	29	55
Median	66	79	54	**100**
Mirror Image	**100**	0	87	63
Negative to Zero	**82**	63	72	81
Number IO	**100**	94	**100**	**100**
Small or Large	9	7	5	**67**
Smallest	**100**	94	97	**100**
String Lengths Backwards	**94**	68	74	53
Sum of Squares	**26**	3	3	5
Super Anagrams	4	21	0	**82**
Syllables	**51**	0	24	2
Vector Average	**92**	5	43	38
Vectors Summed	11	**91**	1	21

provides us with an approximate upper bound on how well GP could perform if given the reasoning power of a programmer.

We compare this result to previously reported numbers from both $\texttt{PushGP}_{BM}$ and G3P. We compare with the best results from $\texttt{PushGP}_{MU}$'s most recently published paper [14] (where they used a variety of new mutation operators), G3P's most recently reported results [10], and $\texttt{PushGP}_{BM}$'s results on the benchmark [15]. In the comparison in Table 5.3 it is important to note that G_{Human} is limited to an order of magnitude less fitness evaluations. With this reduced budget, it was able to outperform or equal $\texttt{PushGP}_{MU}$'s [14] latest results on 12 out of the 21 problems, G3P on 17 of the 21 problems, and the original $\texttt{PushGP}_{BM}$ benchmark [15] on 16 out of the 21 problems. These results testify to the impact of adding domain knowledge.

Table 5.4: Results on 21 benchmark problems using the techniques described in Sections 5.1.2 and 5.1.2. The number of runs (out of 100) that solved all test cases is reported. The best results are **bold**. Done with a total of 30,000 fitness evaluations.

Problem	G_{Base}	$G_{Base+W2F}$	$G_{Base+COW}$	$G_{Base+W2F+COW}$
Checksum	0	0	0	0
Compare String Lengths	30	32	38	**48**
Count Odds	0	0	0	**1**
Digits	70	**77**	72	74
Double Letters	0	0	0	0
Even Squares	0	0	0	0
For Loop Index	0	**3**	0	**3**
Grade	0	0	**2**	**2**
Last Index of Zero	13	18	32	**35**
Median	99	99	99	99
Mirror Image	25	30	25	**32**
Negative to Zero	32	35	56	**58**
Number IO	100	100	100	100
Small or Large	0	1	**67**	**67**
Smallest	100	100	100	100
String Lengths Backwards	15	**17**	15	**17**
Sum of Squares	0	0	**3**	**3**
Super Anagrams	40	61	60	**71**
Syllables	0	0	**1**	**1**
Vector Average	0	0	**1**	**1**
Vectors Summed	1	3	**5**	**5**
Improvement over G_{Base}	-	21/21	21/21	21/21

5.3.1 Problem Description Knowledge Extraction

We next evaluate two variants of `PSGP` that transfer domain knowledge to GP in an automated or semi-automated way. Table 5.4 shows results for $G_{Base+W2F}$ and $G_{Base+COW}$ in columns 3 and 4. Both methods outperform G_{Base} on all 21 problems. They don't always solve the same problems equally as well. This implies the knowledge of the two methods is complementary to some degree. We therefore combine them, see column 5. With $G_{Base+W2F+COW}$ the combination of both methods leads to even better performance, although it is still not quite as good as the upper bound set by G_{Human}. As expected the problems that are impacted the greatest by the use of these methods are those that have important phrases in the problem description like `"reverse"`, `"small"`, or `"zero"`.

5.3.2 Novelty & Fitness Function

We next investigate whether the performance of G_{Human} improves when `knobelty` or a domain-knowledge informed fitness function is added. We see in Table 5.5 that `knobelty` can be an effective operator in improving performance. When used with $G_{Human+N}$, performance improves or stays the same on 15 of the 21 problems compared to using only G_{Human}. Interestingly, the problems where adding novelty search does not improve performance also generally have smaller search spaces, evident in Table 1.1. In fact, all six problems that do worse with the addition of novelty search have a search space size on the order of 10^5, only one order of magnitude more than the $3 * 10^4$ fitness evaluations used. We believe that this is directly linked towards the balance of exploration and exploitation. Novelty search is used due to its ability to promote solutions that escape local minima and that better explore the search space. However, if the search space is small it is less likely that all solutions will get stuck in local minima, and thus spending time exploring the space with novelty search rather than following the gradient to the best solution with lexicase selection becomes detrimental to performance.

In comparison to the previous novelty experiments discussed in the previous two

Chapter, novelty does not have quite as strong a beneficial effect when used with tree based operators. However with the use of G_{Human} we are able to significantly outperform on all three problems (Median, String Lengths Backwards, Smallest) that were worked on in GE_{NOV}, with the same number of fitness evaluations. These results show that while new operators can be used to improve performance, incorporating domain knowledge has the potential to yield a significantly larger boost in performance.

Adapting the fitness function to not just look at the output of a generated program, but also its code has an additional beneficial effect on performance. When combining the new fitness function with G_{Human}, similarly to adding novelty, performance improves or stays the same on 15 of the 21 problems compared to only G_{Human}, see Table 5.4. An additional observed correlation is that 5 of the 6 problems where performance declines using the novelty operator, performance also declines using the augmented fitness function. As novelty and the fitness function should be largely unrelated in their affect on performance, we believe this might be a coincidence and plan to investigate it further in future work.

5.3.3 Student-information Bound

In our final set of experiments, we empirically determine the student-information bound. We increase the fitness evaluation budget to the equivalent level of state of art methods and use a combination of novelty search and the new fitness function ($G_{Human+FF+N}$). This algorithm already outperforms or equals state of art methods on 9 problems with 3e4 fitness evaluation (Table 5.3). We therefore rerun $G_{Human+FF+N}$ on the 12 problems where it did not outperform both `PushGP` and `G3P`. Referencing Table 5.6 we see that with the increased fitness evaluations and the addition of novelty and the augmented fitness function to G_{Human}, the student-information bound is better or equal to $\texttt{PushGP}_{MU}$ on 19 of the 21 problems, able to outperform `G3P` on 20 of the 21 problems, and able to outperform or equal $\texttt{PushGP}_{BM}$ on all 21 problems. While $G_{Human+FF+N}$ was not able to outperform state of art methods on every problem, these results once again speak to the power of incorporating domain knowledge, and thereby promote further study into automating the process of parsing

Table 5.5: Results on 21 benchmark problems using human domain knowledge, novelty, and new fitness function. The number of runs (out of 100) that solved all test cases is reported. The best results are **bold**. Done with a total of 30,000 fitness evaluations.

Problem	G_{Human}	$G_{Human+N}$	$G_{Human+FF}$
Checksum	2	2	**20**
Compare String Lengths	60	63	**73**
Count Odds	**22**	16	18
Digits	79	**100**	98
Double Letters	0	0	0
Even Squares	**1**	**1**	0
For Loop Index	6	**7**	3
Grade	7	8	**11**
Last Index of Zero	**55**	43	52
Median	**100**	98	**100**
Mirror Image	63	71	**85**
Negative to Zero	**81**	72	53
Number IO	100	100	100
Small or Large	67	88	**92**
Smallest	100	100	100
String Lengths Backwards	53	**58**	**58**
Sum of Squares	5	8	**11**
Super Anagrams	**82**	80	72
Syllables	2	4	**10**
Vector Average	38	45	**48**
Vectors Summed	**21**	4	8
Improvement over G_{Human}	-	15/21	15/21

Table 5.6: Results on benchmark problems using domain knowledge, novelty, and augmented fitness function (that G_{Human} did not have best result on). The number of runs (out of 100) that solved all test cases is reported. The best results are **bold**. Done with a total of 300,000 fitness evaluations.

Problem	PushGP$_{MU}$	G3P	$G_{Human+FF+N}$
Checksum	5	0	**56**
Double Letters	**20**	0	2
Even Squares	0	1	**15**
For Loop Index	2	8	**65**
Grade	1	31	**92**
Last Index of Zero	72	22	**86**
Mirror Image	**100**	0	**100**
Negative to Zero	82	63	**98**
String Lengths Backwards	94	68	**99**
Sum of Squares	26	3	**41**
Syllables	**51**	0	26
Vector Average	92	5	**99**
Vectors Summed	11	**91**	41

domain knowledge from problem descriptions.

Chapter 6

Conclusions & Future Work

In this thesis, we evaluated the impact of an explicit exploitation - exploration trade off across a range of conditions. We find that having a `knobelty` control is largely successful, especially when using genomic operators as they often struggle to escape local optima. We also evaluate various ways to handle the bloating associated with novelty search and more complex tree based mutation and crossover operators. We find that a size penalty during lexicase selection is very effective, successfully increasing performance when using `GE_Perf` and limiting the size of the solutions. Additionally, as solutions become a more reasonable size, novelty search again is beneficial as it is no longer being drowned out due to the higher novelty that large solutions inherently possess.

We also evaluated the impact that the addition of domain knowledge can have on the program synthesis problem. We define the *student-information* bound, and manually create a grammar that is able to approximate the bound. We then work on methods to automatically parse information from the problem statement in order to attempt to reach the upper bound we set in the manually created grammar. We establish that the addition of domain knowledge has the capacity to be extremely effective, and demonstrate that finding solutions to some of the benchmark problems without the addition of domain knowledge is largely infeasible. We also demonstrate that automatically parsing information from the problem statement can be effective in pruning the grammar, as well as augmenting the fitness function. Lastly we find

that novelty search is not as effective on the pruned grammars, which we believe is due to the reduction in search space size. With a smaller space to explore, exploitation becomes more valuable to success.

There are a number of avenues for additional improvement that could be explored in future work. Firstly, it would be interesting to investigate other representations of program behavior to use as a novelty metric, including program traces, and other static program and derivation tree representations. Additionally, while using tree based operators showed improvement, $\texttt{PushGP}_{MU}$ has shown that other mutation operators could lead to even further improvement. It would be interesting to see how novelty search is able to impact an algorithm with these new mutation operators. In regards to domain knowledge, it would be useful to research whether incorporating other elements available in the `python` standard library will allow us to be successful on the remaining 8 problems on the benchmark. Also, the fruitful strategy of considering programming broadly could be extended to consider another angle. Programmers rarely get a program correct on their first try. They encounter syntax and semantic logic errors and then have to debug to fix them. Future work could look at how debugging could inform GP search strategies. We could also look at common patterns of bugs and their fixes to see if GP can be extended to identify and directly incorporate them, rather than leave them to search. Finally, we could compare against other program synthesis methods, and see the effect that novelty search and domain knowledge could have in improving those methods.

Bibliography

[1] Michael Affenzeller, Stephan M Winkler, Bogdan Burlacu, Gabriel Kronberger, Michael Kommenda, and Stefan Wagner. Dynamic observation of genotypic and phenotypic diversity for different symbolic regression gp variants. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1553–1558. ACM, 2017.

[2] Alexandros Agapitos and Simon M. Lucas. Learning recursive functions with object oriented genetic programming. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 166–177, Budapest, Hungary, 10 - 12 April 2006. Springer.

[3] Taylor L Booth. Sequential machines and automata theory. 1967.

[4] Edmund K Burke, Steven Gustafson, and Graham Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, 2004.

[5] Armand R Burks and William F Punch. An analysis of the genetic marker diversity algorithm for genetic programming. *Genetic Programming and Evolvable Machines*, 18(2):213–245, 2017.

[6] Giuseppe Cuccu and Faustino Gomez. When novelty is not enough. In *European Conference on the Applications of Evolutionary Computation*, pages 234–243. Springer, 2011.

[7] Kenneth Alan De Jong. Analysis of the behavior of a class of genetic adaptive systems. 1975.

[8] John Doucette and Malcolm I Heywood. Novelty-based fitness: An evaluation under the santa fe trail. In *European Conference on Genetic Programming*, pages 50–61. Springer, 2010.

[9] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Michael O'Neill, and Erik Hemberg. Ponyge2: Grammatical evolution in python. *CoRR*, abs/1703.08535, 2017.

[10] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. Towards understanding and refining the general program synthesis benchmark suite with genetic programming. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–6. IEEE, 2018.

[11] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael ONeill. A grammar design pattern for arbitrary program synthesis problems in genetic programming. In *European Conference on Genetic Programming*, pages 262–277. Springer, 2017.

[12] Heather J Goldsby and Betty HC Cheng. Automatically discovering properties that specify the latent behavior of uml models. In *International Conference on Model Driven Engineering Languages and Systems*, pages 316–330. Springer, 2010.

[13] Faustino J. Gomez. Sustaining diversity using behavioral information distance. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 113–120, New York, NY, USA, 2009. ACM.

[14] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. Program synthesis using uniform mutation by addition and deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1127–1134. ACM, 2018.

[15] Thomas Helmuth and Lee Spector. General program synthesis benchmark suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1039–1046. ACM, 2015.

[16] Erik Hemberg, Jonathan Kelly, and Una-May O'Reilly. On domain knowledge and novelty to improve program synthesis performance with grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2019.

[17] Gregory S Hornby. 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*, pages 815–822. ACM, 2006.

[18] Jonathan Kelly, Erik Hemberg, and Una-May O'Reilly. Improving genetic programming with novel exploration - exploitation control. In *European Conference on Genetic Programming*. Springer, 2019.

[19] Krzysztof Krawiec. *Behavioral program synthesis with genetic programming*, volume 618. Springer, 2016.

[20] Joel Lehman and Kenneth O Stanley. Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE*, pages 329–336, 2008.

[21] Joel Lehman and Kenneth O Stanley. Efficiently evolving programs through the search for novelty. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 837–844. ACM, 2010.

[22] Joel Lehman and Kenneth O Stanley. Novelty search and the problem with objectives. In *Genetic programming theory and practice IX*, pages 37–56. Springer, 2011.

[23] Víctor R López-López, Leonardo Trujillo, and Pierrick Legrand. Novelty search for software improvement of a slam system. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1598–1605. ACM, 2018.

[24] Simon Lucas. Exploiting reflection in object oriented genetic programming. In Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 369–378, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag.

[25] Ole J Mengshoel and David E Goldberg. The crowding approach to niching in genetic algorithms. *Evolutionary computation*, 16(3):315–354, 2008.

[26] Melanie Mitchell, Michael D Thomure, and Nathan L Williams. 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*, pages 118–124. MIT Press Cambridge, MA, 2006.

[27] Enrique Naredo. *Genetic Programming Based on Novelty Search*. PhD thesis, ITT, Instituto tecnologico de Tijuana, 2016.

[28] Miguel Nicolau and Michael Fenton. Managing repetition in grammar-based genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 765–772. ACM, 2016.

[29] Michael O'Neill and David Fagan. The elephant in the room: Towards the application of genetic programming to automatic programming. In *Genetic Programming Theory and Practice XVI*, pages 179–192. Springer, 2019.

[30] Michael O'Neill and Conor Ryan. Evolving multi-line compilable C programs. In Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 83–92, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[31] Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4):339–363, 2010.

[32] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming*. Lulu. com, 2008.

[33] Conor Ryan, Michael ONeill, and JJ Collins. Introduction to 20 years of grammatical evolution. In *Handbook of Grammatical Evolution*, pages 1–21. Springer, 2018.

[34] Hormoz Shahrzad, Daniel Fink, and Risto Miikkulainen. Enhanced optimization with composite objectives and novelty selection. *arXiv preprint arXiv:1803.03744*, 2018.

[35] Vinay Shashidhar, Nishant Pandey, and Varun Aggarwal. Automatic spontaneous speech grading: A novel feature derivation technique using the crowd. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 1085–1094, 2015.

[36] Dirk Sudholt. The benefits of population diversity in evolutionary algorithms: A survey of rigorous runtime analyses. *arXiv preprint arXiv:1801.10087*, 2018.

[37] Ann Thorhauer and Franz Rothlauf. On the locality of standard search operators in grammatical evolution. In *International Conference on Parallel Problem Solving from Nature*, pages 465–475. Springer, 2014.

[38] Mingxu Wan, Thomas Weise, and Ke Tang. Novel loop structures and the evolution of mathematical algorithms. In Sara Silva, James A. Foster, Miguel Nicolau, Penousal Machado, and Mario Giacobini, editors, *Genetic Programming - 14th European Conference, EuroGP 2011, Torino, Italy, April 27-29, 2011. Proceedings*, volume 6621 of *Lecture Notes in Computer Science*, pages 49–60. Springer, 2011.

[39] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

[40] T. Weise, M. Wan, K. Tang, and X. Yao. Evolving exact integer algorithms with genetic programming. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 1816–1823, July 2014.

[41] Thomas Weise and Ke Tang. Evolving distributed algorithms with genetic programming. *IEEE Trans. Evolutionary Computation*, 16(2):242–265, 2012.

[42] Tina Yu and Chris Clack. Recursion, lambda-abstractions and genetic programming. In Riccardo Poli, W. B. Langdon, Marc Schoenauer, Terry Fogarty, and Wolfgang Banzhaf, editors, *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming*, pages 26–30, Paris, France, 14-15 April 1998. CSRP-98-10, The University of Birmingham, UK.