On Domain Knowledge and Novelty to Improve Program Synthesis Performance with Grammatical Evolution

Erik Hemberg
Massachusetts Institute of Technology
Cambridge, Massachusetts
hembergerik@csail.mit.edu

Jonathan Kelly
Massachusetts Institute of Technology
Cambridge, Massachusetts
jgkelly@mit.edu

Una-May O'Reilly
Massachusetts Institute of Technology
Cambridge, Massachusetts
unamay@csail.mit.edu

ABSTRACT

Programmers solve coding problems with the support of both programming and problem specific knowledge. They integrate this domain knowledge to reason by computational abstraction. Correct and readable code arises from sound abstractions and problem solving. 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. We examine the impact of using such knowledge on 21 problems from the GP program synthesis benchmark suite. 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.

CCS CONCEPTS

• **Computing methodologies** → *Multi-agent systems.*

KEYWORDS

Genetic Programming, grammar, program synthesis, novelty

ACM Reference Format:
Erik Hemberg, Jonathan Kelly, and Una-May O'Reilly. 2019. On Domain Knowledge and Novelty to Improve Program Synthesis Performance with Grammatical Evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference 2019 (GECCO '19).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Program synthesis has been a long standing challenge problem of AI. It is the exceptional level of human expertise that is required by programming that elevates it to such a high challenge status. Programming is rated a super human intelligence task by [18] and the nature of the extraordinary expertise it requires has been distilled into the notion of computational thinking which is now taught in schools.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
GECCO '19, July 13–17, 2019, Prague, Czech Republic
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Program synthesis is an intuitively natural problem for genetic programming (GP). It is also an important challenge for the GP community to work on [13, 15] 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 [8] 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. One set of efforts has demonstrated the merits of lexicase selection and used program synthesis to motivate a new mutation operator for `PushGP` [6, 7]. Another has introduced a grammar-guided approach and analyzed the suite's problems to reflect upon general grammar design [4, 5]. More recently, program synthesis has supported the investigation of how tunable selection-based novelty can help balance search exploration and exploitation. Simultaneously, narrow program synthesis performance improvement was also demonstrated [9]. As these are some of the most recent contributions to the program synthesis community, for the rest of the paper we refer to them as state of art.

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. 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 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, 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

Table 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

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. The first contribution of this paper is to experimentally determine this bound. We do this by using the grammar we used in calculating column 3 of Table 1.

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 hypothesize that there is knowledge that can be automatically extracted from program synthesis problem descriptions using alternate AI methods that can be successfully integrated into an improved Genetic Programming/Grammatical Evolution approach. The central questions of this paper relate to this hypothesis. They are *(1)* How can legitimate information about a programming language and problem description be transferred automatically to GP? What information is possible and useful to extract? *(2)* Will GP be able to solve more problems in the benchmark suite, specifically if:

(i) We create a synonymous mapping between the built-in functions of a programming language and English words and then automatically parse a problem definition to see if these words are within it. For each word we identify, we increase the likelihood that GP uses that function in its search by altering the bias of a basic programming language grammar to favor it.

(ii) We permit GP to avoid having to "invent its own resistors", i.e. simultaneously solve lower-order problems at a disparate level of abstraction. We parse the problem description for magic numbers and string constants and add these to the grammar.

(iii) We inspect the problem definition for words describing arithmetic functions such as `sum` or the + sign and relational operators such as `less than` or $\leq$ and automatically update a basic programming language grammar to improve the likelihood that the equivalent arithmetic and logic operators of the programming language will be used.

(iv) We remove from the grammar elements of the programming language that the problem description implies are not necessary.

(v) We extract problem description information and use programming knowledge to inform GP's fitness function. If during evolution, an inspection of a candidate solution reveals program statements that the problem description sets up the expectation of being used, fitness is increased.

We implement each of these options and integrate them into an algorithm we name `PSGP`. Because they inform GP and to increase investigative agility, we reduce the evaluation budget available to them to 10% of state of art (3e4). With this budget, we evaluate them on 21 of the benchmark suite problems and compare them to each other and state of the art. Because they exhibit superior performance over a baseline, we try them in combination. Finally, we focus on the problems for which knowledge-based methods do not yield superior performance in comparison to previous work and provide them with the state of art budget (3e5). We provide them with the pared grammar, G_{HUMAN}, the domain knowledge-based fitness function information and enhance `PSGP` with `knobelty` selection which brings the algorithm up to state of art and the imported domain knowledge to a maximum. This sets a bound that we propose the community refer 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.

The paper is structured as follows. Section 2 presents background, Section 3 presents methods that translate our hypotheses into practice, Sections 4 and 5 present experiments and their results, and finally, Section 6 concludes and presents future work.

2 BACKGROUND

This section presents background. We cover program synthesis with GP in Section 2.1, selection operators for program synthesis in Section 2.2, and grammatical evolution in Section 2.3. In Section 2.4 we discuss the incorporation of domain knowledge into GP.

2.1 Program Synthesis with GP

In GP, program synthesis is formulated as an optimization problem akin to system identification: 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: $\mathbb{1}: q(x) \neq y$.

Some early work in program synthesis considered specific programming techniques, such as recursion, lambda abstractions and

reflection. Other work considered synthesis using a particular programming language such as C. Yet other work addressed specific coding problems such as caching, [1, 11, 14, 20, 22–24]. From a technique perspective, other approaches covered implicit fitness sharing, co-solvability, trace convergence analysis, pattern-guided program synthesis, and behavioral archives of subprograms, [10].

A watershed moment was the introduction of a program synthesis benchmark suite of 29 problems, systematically selected from sources teaching introductory computer science programming [8]. Four studies are most relevant to this paper: the original benchmarking done using PUSHGP [8], the most recent PUSHGP results using various mutation operators [7], the most recent grammar guided GP efforts [4], and, recently introduced, grammatical evolution with knobelty selection [9]. Henceforth we refer to these as $\texttt{PushGP}_{BM}$, $\texttt{PushGP}_{MU}$, G3P, and GE_{NOV} respectively.

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.

2.2 Selection Operators for Program Synthesis

Selection operators have demonstrated an influential role in program synthesis success. Lexicase selection does not base selection on a single fitness value [8]. Instead, it uses a random ordering of the training set to select individuals that perform as well as possible on a subset of the test cases, even if they perform poorly on other test cases. The goal of lexicase selection can be summarized as promoting into the next generation, parents that collectively solve different test cases.

Another is knobelty selection [9]. It parametrically balances exploration and exploitation by populating a mixed population of offspring. One subset it populates is selected based on performance quality and the other subset is selected based on diversity. A potential selected candidate is compared to samples drawn from a cache of all historical solutions and discarded if it is a duplicate. This sampling introduces heuristic efficiency in lieu of exhaustive correctness and dispenses with a complex cache management policy. 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 when, functionally, they are not [9]. Additionally, in GP, tree based operators also have a tendency to induce bloat [16].

2.3 Grammatical Evolution

Grammatical Evolution (GE) is a genetic programming algorithm where a Backus Naur Form (BNF) context free grammar is used in the genotype to phenotype mapping process [17]. The BNF expresses a production rule as a non-terminal left-hand side, a separator and a list of productions on the right hand side, each production can contain terminals and/or non-terminals. Formally, 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, $S \in N$ [2]. In GE the probability of selecting a production for a given non-terminal is approximately uniform, with the probability depending on the number of productions.

A grammar expresses the infinite, combinatorial language of a GP problem's functions and terminals and their combinations. It also provides flexibility because once grammar decoding and grammar-based operators are implemented, a grammar and the implementation of its symbol can be inserted without changing anything else in the GE algorithm.

The grammar is a starting point for a two step sequence of mappings that decode a genotype to its phenotype or program:

1) **Genotype to derivation tree**: The genotype, an integer sequence, is referenced, one integer at a time, to control production rule rewriting. This rewriting generates a rule production sequence which can be represented as a derivation tree. At each rewriting step, the integer "gene" dictates which specific element in the list of right hand side productions rewrites the current non-terminal. Typically the production at the (gene modulo number of productions in the rule) position is selected.

2) **Derivation tree to phenotype/program**: The leaves (terminals) of the derivation tree constitute the executable code or program that GE can evaluate.

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 [19]. Others operate on the derivation tree [3]

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 [15]. 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 Section 3.2.

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 [21]. Another, considering program synthesis, is the use of different stacks for PushGP$_{BM}$ [8].

Although not explicitly stated, PushGP$_{BM}$, 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 3.2.2

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. [16].

To the best of our knowledge we have not seen studies regarding incorporating explicit domain knowledge in the grammar and fitness function by forming a knowledge base and inspecting the problem statement and the solutions. The next section introduces the methods we develop for this investigation.

3 METHOD

We present our methods in the following order. We start, in Section 3.1, 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 3.2 we discuss how legitimate domain knowledge – programming language and problem description information, can be transferred. We then, in Sections 3.2.1–3.2.3, present the methods by which we hunt out specific information from problem descriptions and specifically change PSGP.

3.1 Program Synthesis Algorithm: PSGP

An overview of PSGP is shown in Algorithm 1. Our PSGP implementation originates from the grammar based genetic programming repository of PonyGE2 [3]. Building on PonyGE2, we have added lexicase selection and a `knobelty` selection operator [9]. Our code and all grammars are available [1].

We introduce minor adaptations to the `knobelty` selection operator. 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 [12]. Second, GE with tree based operators and `knobelty` can cause bloat. Bloat interferes with the simplicity and runtime complexity of solutions.

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

Algorithm 1 PSGP(D, I, K, Θ) **Program Synthesis GP**
Parameters: D test cases, I problem statement, K knowledge base, Θ hyper parameters

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

PSGP gives a preference to smaller solutions during lexicase selection. 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.

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.

3.2 Problem Description Knowledge Extraction

We prepared PSGP to run with a variety of grammars. Its base grammar, G_{BASE} 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 [4]. 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 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 [8]:

```
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 ≥ 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 2.3, 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.

3.2.1 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.

3.2.2 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.

3.2.3 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 [8]:

```
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.

4 EXPERIMENTS

Since [8] 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 [4, 8] 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.

Table 2: 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 [9]
Knobelty λ	Generations/10

Table 3: Variant abbreviations

Abbreviation	Variant
G_{Base}	Base grammar
$G_{Base+W2F}$	Words to Functions (Section 3.2.1)
$G_{Base+COW}$	Constants, operators and what not to use (Section 3.2.2)
G_{Human}	Human's grammar (Section 3.2)
$G_{Human+N}$	Human's grammar and knobelty operator (Section 2.2)
$G_{Human+FF}$	Human's grammar and augmented fitness function (see Section 3.2.3)
$G_{Human+FF+N}$	Human's grammar, augmented fitness function and knobelty operator

4.1 Setup

We report results on 100 runs. We report program synthesis performance in the same way as previous work, in terms of how many runs out of 100 resulted in one or more programs that solved all the out of sample (test) cases. All other reported values are averages over 100 runs. We ran all experiments on a cloud (OpenStack) VM with 24 cores, 24GB of RAM using Intel(R) Xeon(R) CPU E5-2450 v2 @ 2.50GHz.

The set of static parameters we use throughout all our experiments is listed in Table 2. 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 GE_{NOV} [9]. 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 GE_{NOV} [9].

We use the variants of PSGP listed in Table 3.

5 RESULTS

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.

Table 4: 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. PushGP$_{MU}$ [7], PushGP$_{BM}$ [8], G3P [4] use a total of 300,000 fitness evaluations.

Problem	PushGP$_{MU}$	G3P	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

We start by answering the first question of the introduction. We manually created a grammar that was minimally powerful to express correct solutions. Understanding how GP performs using this grammar, G_{HUMAN}, 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 PushGP$_{BM}$ and G3P. We compare with the best results from PushGP$_{MU}$'s most recently published paper [7] (where they used a variety of new mutation operators), G3P's most recently reported results [4], and PushGP$_{BM}$'s results on the benchmark [8]. In the comparison in Table 4 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 PushGP$_{MU}$'s [7] latest results on 12 out of the 21 problems, G3P on 17 of the 21 problems, and the original PushGP$_{BM}$ benchmark [8] on 16 out of the 21 problems. These results testify to the impact of adding domain knowledge.

5.1 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 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".

Table 5: Results on 21 benchmark problems using the techniques described in Sections 3.2.1 and 3.2.2. 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

Table 6: 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

5.2 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 6 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. 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 previous work by GE_{NOV} using genomic operators, 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. 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 Student-information Bound

To end, 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 4). We therefore rerun $G_{Human+FF+N}$ on the 12 problems where it did not outperform both `PushGP` and `G3P`. Referencing Table 7 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 domain knowledge from problem descriptions.

6 CONCLUSIONS AND FUTURE WORK

By introducing domain knowledge gathered from the problem we attempted to solve programming synthesis problems with the same

Table 7: 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

knowledge a human programmer brings to them. Ideally all knowledge extraction work would be done automatically without any human intervention. We think this is a plausible goal and point out that it requires AI but not GP. We plan to make $G_{Base+W2F}$ and $G_{Base+COW}$ more robust and broader in the future. While we think that working with 21 of the benchmark problems is adequate in understanding the effect that our work can have on the general program synthesis landscape, we intend to continue expanding our grammar. We plan 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. Additionally we plan to expand the representation to allow non-uniform probabilities in the grammar, so as to more easily bias frequent code fragments to appear more often. PushGP$_{MU}$ has shown a significant improvement can be achieved from using better mutation operators. We plan to incorporate these mutation operators in future work as we believe that a similar boost can be achieved when paired with our human grammar as well as novelty search and the augmented fitness function.

This 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. Our community 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.

REFERENCES

[1] Alexandros Agapitos and Simon M. Lucas. 2006. Learning Recursive Functions with Object Oriented Genetic Programming. In *Proceedings of the 9th European Conference on Genetic Programming (Lecture Notes in Computer Science)*, Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt (Eds.), Vol. 3905. Springer, Budapest, Hungary, 166–177.

[2] Taylor L Booth. 1967. Sequential machines and automata theory. (1967).

[3] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Michael O'Neill, and Erik Hemberg. 2017. PonyGE2: Grammatical Evolution in Python. *CoRR* abs/1703.08535 (2017). arXiv:1703.08535 http://arxiv.org/abs/1703.08535

[4] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2018. Towards Understanding and Refining the General Program Synthesis Benchmark Suite with Genetic Programming. In *2018 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 1–6.

[5] Stefan Forstenlechner, Miguel Nicolau, David Fagan, and Michael O'Neill. 2016. Grammar design for derivation tree based genetic programming systems. In *European Conference on Genetic Programming*. Springer, 199–214.

[6] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2016. Lexicase selection for program synthesis: a diversity analysis. In *Genetic Programming Theory and Practice XIII*. Springer, 151–167.

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

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

[9] Jonathan Kelly, Erik Hemberg, and Una-May O'Reilly. 2019. Improving Genetic Programming with Novel Exploration - Exploitation Control. In *European Conference on Genetic Programming*. Springer.

[10] Krzysztof Krawiec. 2016. *Behavioral program synthesis with genetic programming*. Vol. 618. Springer.

[11] Simon Lucas. 2004. Exploiting Reflection in Object Oriented Genetic Programming. In *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings (LNCS)*, Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule (Eds.), Vol. 3003. Springer-Verlag, Coimbra, Portugal, 369–378.

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

[13] Michael O'Neill and David Fagan. 2019. The Elephant in the Room: Towards the Application of Genetic Programming to Automatic Programming. In *Genetic Programming Theory and Practice XVI*. Springer, 179–192.

[14] Michael O'Neill and Conor Ryan. 1999. Evolving Multi-Line Compilable C Programs. In *Genetic Programming, Proceedings of EuroGP'99 (LNCS)*, Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty (Eds.), Vol. 1598. Springer-Verlag, Goteborg, Sweden, 83–92.

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

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

[17] Conor Ryan, John James Collins, and Michael O'Neill. 1998. Grammatical evolution: Evolving programs for an arbitrary language. In *European Conference on Genetic Programming*. Springer, 83–96.

[18] Vinay Shashidhar, Nishant Pandey, and Varun Aggarwal. 2015. 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)*, Vol. 1. 1085–1094.

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

[20] Mingxu Wan, Thomas Weise, and Ke Tang. 2011. Novel Loop Structures and the Evolution of Mathematical Algorithms. In *Genetic Programming - 14th European Conference, EuroGP 2011, Torino, Italy, April 27-29, 2011. Proceedings (Lecture Notes in Computer Science)*, Sara Silva, James A. Foster, Miguel Nicolau, Penousal Machado, and Mario Giacobini (Eds.), Vol. 6621. Springer, 49–60. https://doi.org/10.1007/978-3-642-20407-4_5

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

[22] Thomas Weise and Ke Tang. 2012. Evolving Distributed Algorithms With Genetic Programming. *IEEE Trans. Evolutionary Computation* 16, 2 (2012), 242–265. https://doi.org/10.1109/TEVC.2011.2112666

[23] T. Weise, M. Wan, K. Tang, and X. Yao. 2014. Evolving exact integer algorithms with Genetic Programming. In *2014 IEEE Congress on Evolutionary Computation (CEC)*. 1816–1823. https://doi.org/10.1109/CEC.2014.6900292

[24] Tina Yu and Chris Clack. 1998. Recursion, Lambda-Abstractions and Genetic Programming. In *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming*, Riccardo Poli, W. B. Langdon, Marc Schoenauer, Terry Fogarty, and Wolfgang Banzhaf (Eds.). CSRP-98-10, The University of Birmingham, UK, Paris, France, 26–30.