Using Existing Knowledge for Transfer and Regularization for Program Synthesis with Genetic Programming

by

Jordan Wick

B.S., Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author .....
Department of Electrical Engineering and Computer Science
May 18, 2020

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

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

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

Using Existing Knowledge for Transfer and Regularization for Program Synthesis with Genetic Programming

by

Jordan Wick

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In normal Genetic Programming (GP), test case performance is the only signal the population has to improve on. However, human programmers use other signals to guide them - they know what "good" code looks like. They often reuse program patterns across multiple functions, showing **Transferability** of knowledge between programming tasks. Pieces of code also become more or less likely in different contexts - you don't see many *For* loops nested 4 layers deep in codebases generated by humans. In this thesis, Transferability is explored in the context of Grammatical Evolution, looking at how a population of problems being optimized to solve one problem can be used to aid in solving a similar problem. To do this, methods were created to parse existing solutions from a codebase into the Grammatical Evolution representation, and operators were implemented that switch the GE objective throughout the process of evolution. A "humanlike" objective was defined which takes into account the distribution of AST nodes within different program contexts. This was used as Regularization during GE, in that programs that strayed further from the humanlike distribution of nodes received a penalty.

It was found that optimizing for one problem first can make it easier to find a solution to other similar problems, especially when the solution to one problem is used in the other - however, the amount of pre-optimization and the choice of problem are of great imprtance. Additionally, optimizing directly for code to become more "humanlike" via the defined measure was not effective in allowing the population solve test cases more efficiently, although selecting directly for these metrics did change the distribution of these metrics in the resulting populations. This shows that while surrogate objectives can improve performance, they need to be chosen carefully.

Thesis Supervisor: Erik Hemberg
Title: Research Scientist

Thesis Supervisor: Una-May O'Reilly

Title: Principal Research Scientist

Acknowledgments

I'd like to thank Erik Hemberg for his guidance and support throughout the process, staying up to help write papers, and finding time in between all of his meetings, and Una-May O'Reilly for her support and in making the lab a welcoming environment. I'd also like to thank Dirk Schweim for his help and effort on our project in making the code generated by GE more humanlike.

I'd also like to thank my professors for helping me learn all that I did this year. Thank you to Gerry Sussman and Jack Wisdom for teaching me a new way of thinking about classical mechanics and filling lab sections with interesing conversations, Pulkit Agrawal for teaching me Reinforcement Learning and discussing ideas with me during office hours, and Chris Rackauckas for teaching me about Numerical Methods and getting me excited about the Julia Language.

And finally, I'd like to thank all the friends who helped me throughout the year - from working on assignments together to talking into the late hours of the night, they made MIT an amazing experience that I'll look back on fondly.

This material is based upon work supported by the DARPA Advanced Research Project Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under Contract No. N66001-18-C-4036.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Code is time-consuming to write - it requires years of training to be able to write effectively, many bugs take a significant amount of time to debug, and algorithms can get relatively complex. The field of program synthesis aims to reduce this burden by synthesizing software automatically [12, 16]. While there are many ways to tackle this problem, this thesis focuses on one variant of Genetic Programming called **Grammatical Evolution (GE)** [25].

1.1 Background

Grammatical Evolution [25] is a technique in which a population of programs is evolved to solve a particular programming task. Whether or not a program "solves" a task is usually quantified by the program passing all of a given suite of test cases. The search space is defined by a Formal Grammar, where the overall solution to the programming task is able to be expressed as a string within this grammar. Each program is then represented as a list of integers, which are used to determine which production rules are used within the grammar in order to generate a program.

In GE, a population of programs is first initialized (*Initialization*). Afterwards, this population repeatedly goes through a process of being evaluated to determine a numerical fitness value (*Evaluation*), has a subset of the population selected to survive into the next generation (*Selection*), and then individuals are combined and

mutated before proceeding into the next generation (*Crossover* and *Mutation*). Over time, the population becomes more adept at solving the task defined in the fitness function.

Each program has an associated Abstract Syntax Tree (AST), which defines the structure of the program. ASTs are composed of different types of nodes, some of which are more likely to appear in some contexts than others. For example, the probability that a *For* node will show up in the context of 5 For loops is much lower than if it were in the top-level namespace of a program. This work defines a "humanlike" score based on the likelihood of a particular type of node appearing at different contexts, relative to some baseline which is learned from a corpus of programs.

1.2 Motivation and Challenges

This section covers the main areas of motivation of this thesis, which is focused on Knowledge Transfer from a corpus of programs to a population of programs being evolved with GE.

1.2.1 Transferring knowledge between similar problems

Human programmers do not learn how to solve programs in isolation. Instead, they transfer knowledge, and even code, when they solve similar problems. In contrast, Genetic Programming (GP) methods for Program Synthesis traditionally solve a single problem at a time [12, 10, 13]; the GP system is given a suite of problems and optimizes for each task independently, despite the fact that many program solutions share subprograms, e.g. iterating over the elements in a list.

Recent work in program synthesis with GP have studied simultaneous synthesis of multiple functions using genetic programming with scaffolding [4] and parallel evolution of string library functions [30]. However, none of these have considered transferring existing knowledge from the start of the synthesis, as is done by a human programmer.

Because similar problems have solutions that share subprograms, a population optimized to solve one problem first will likely have information that's useful for solving another problem given that the problems have similar structure. With this in mind, we explore how alternating the GE objective between different but similarly-structured problems affects a population's ability to solve a given task. The effects of starting a population only from solutions to a similar problem are also explored.

One of the main challenges of this is to design a grammar that will be able to capture these similarities in structure between the problems, and in general to allow for the transfer of information from one solution to another.

1.2.2 Transferring knowledge from corpus of human-generated programs to GE

While there is valuable information within populations of solutions that solve a similar problem to the one being studied, there is also likely untapped information found within large corpora of programs - these can provide baselines for what a program "should" look like, if it were written by a human. When looking at code from GE runs, many of them have a lack of readability, in that the generated program contains extra symbols that provide no value to solving the problem at hand, or don't get evaluated, and solutions tend to grow unbounded in size [24]. For example, `if` statements get produced which either have nothing meaningful in their body or always have their condition evaluate to `False`, or the number 0 will be added to another variable repeatedly. GE also doesn't explicitly take context-sensitivity into account - while multiple nested loops are relatively unlikely in high-quality human-generated code, if loops are allowed in GE, multiple nested loops could form, increasing runtime and decreasing performance. These problems can be manually solved by changing the grammar to only allow loops in certain locations, but this adds constraints and limits the types of solutions that can be generated. In an ideal world, we'd like to be able to optimize not only for fitness, but also for readibility and sensibility of code.

While optimizing for the number of test cases solved is the most common optimization objective in GE, several other surrogate objectives have shown promise as potential alternative optimization objectives. One such example is Novelty Search [15, 13], which defines 'novelty' as individuals which are more distant, on average, to the other members of a population, defined by some distance metric. Optimizing for this formulation of novelty alongside test case fitness has been shown to allow GE to solve all test cases quicker.

While novelty is one surrogate objective that can be optimized for, there are several other aspects of code that can be optimized for that would ideally make the code more "humanlike", such as McCabe complexity, Tree Size, and Lexical Diversity [29]. These and other metrics have been proposed to measure various aspects of code quality, readability, and complexity. In addition, properties of a program's AST can give some insight into the structure of the program [29]. In this work, we define a **Humanlike Score** for programs, which is defined as the similarity between the distribution of AST nodes found at different contexts within a program, relative to a target population of programs.

With this in mind, we propose to make the ASTs of programs look similar to those generated by humans, based on the likelihood of the occurence of nodes in the tree, relative to a baseline corpus of programs. This provides one measure in which we can make GE code similar to humanlike code - the main challenge, and main question, is just if this provides enough signal to be useful for making the code more effective or readable.

1.3 Research Questions

The first set of research questions is related to the transferability of information between populations of programs (individuals). For this set of research questions, the aim is to understand how information can be transferred between problems with similar structure.

- **RQ1.1**: Can optimizing a GP program synthesis population for one task make it more efficient at solving a similar task?
- **RQ1.2**: Can GP program synthesis solve complex tasks by combining the solutions to simpler ones?
- **RQ1.3**: Does GP program synthesis benefit from the use of intermediate goals?
- **RQ1.4**: How can existing knowledge in the form of existing programs be incorporated in the GP search?

The questions above focused on tranferring knowledge between programs that solve similar tasks. Our second set of research questions aims to see how knowledge can be transferred between high-quality programs and GE search in general, not restricting only to programs with similar function. Specifically, they aim to understand how optimizing for code that match the AST Node distribution of a given target population affects the outcomes of GE. This is examined both in terms of the readability and interpretability of generated solutions, along with the final test case performance.

- **RQ2.1**: Does optimizing directly for a desired value for program metrics actually lead to a population where these metrics take on the desired value?
- **RQ2.2**: Can optimizing directly for these code metrics result in increased test case performance?
- **RQ2.3**: Does optimizing for these code metrics make our solutions more humanlike?
- **RQ2.4**: Is is easier to find solutions when starting from pieces of code that have been optimized to be more humanlike?

1.4 Contributions and Thesis Structure

In this thesis, the following contributions were made:

1. A method for parsing programs into the GE representation, to initialize populations from existing programs.

2. We find that without tuning operators and grammatical structure to capitalize on information transfer between similar problems, performance gains aren't found when initializing a population with solutions to similar problems **RQ1.2, 1.4**.

3. A framework for switching the optimization objective throughout a GE run.

4. It is found that when optimizing for a program to solve multiple problems, it can be beneficial to start with the harder problem as a subgoal as opposed to the easier problem, and that optimizing for subproblems first leads to a much greater chance of solving the given problem **RQ1.1, 1.3**.

5. The development and investigation of a context-sensitive Humanlike Score, which measures how similar a program's AST is to a target corpus of programs.

6. We find that optimizing for Humanlike score can lead to programs that are readable, while also showing that certain settings can lead to degenerate solutions, but directly optimizing for only this Humanlike score performs significantly worse than Random Search **RQ2.2, 2.3**.

7. We find that initializing from a population optimized for the Humanlike score doesn't have significant improvements to test case performance, although further tuning of this metric could be required **RQ2.4**

The rest of this thesis will be structured as follows. Chapter 2 will walk through the background and related work. Chapter 3 will cover the experiments in the Transferability of knowledge between populations of related solutions, aimed at answering **RQ1.1-1.4**. Chapter 4 will cover the experiments in optimizing for this Humanlike Score in the selection phase, aimed at answering **RQ2.1-2.4**. Chapter 5 will walk through the main conclusions reached in this work, along with directions for future research.

Chapter 2

Background and Related Work

In this chapter, we'll cover the background relevant to this work and describe the datasets used in the experiments. Chapter three will cover our experiments on the transferability of GE solutions to similar problems, Chapter four will go over our methods for making code more humanlike, and Chapter five will cover the main conclusions we reached.

2.1 Background

Here we cover the background for the techniques discussed in this thesis.

2.1.1 Program Synthesis with Genetic Programming

GP cares about automatic programming [23]. In GP, one manifestation of this is as program synthesis, which 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 $\mathbb{D} = [X, Y]^N, x \in X, y \in Y$, with $q : X \leftarrow Y$. Typically an indicator function measures error on a single case: $\mathbb{1}$: $q(x) \neq y$. The program q can be represented by some language L. We can formulate the program synthesis problem as $\arg\min_{q \in Q} q(X) - Y$

Work in program synthesis has considered specific programming techniques, such as recursion, lambda abstractions and reflection. Some other work focused on particular programming language such as C, while others studied specific coding problems such as caching, [21, 19, 2, 35, 32, 34, 33]. From a technique perspective, other approaches covered implicit fitness sharing, co-solvability, trace convergence analysis, pattern-guided program synthesis, and behavioral archives of subprograms, [16]. Important progress for program synthesis in GP was the introduction of a benchmark suite of 29 problems, systematically selected from sources teaching introductory computer science programming [12, 10, 13].

Transparency is a desirable property of GP program synthesis solutions in addition to performance. Previous work use *transparency* to refer to human readability and interpretability [13]. We extend this to also consider transferability, the ability of a program synthesis solution to be transferred to other program synthesis tasks.

2.1.2 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 [25]. 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, $S \in N$ [5]. In GE the probability of selecting a production from a rule is approximately uniform, with the probability depending on the number of productions.

The sentences a grammar can produce forms the language L. The grammar is a starting point for a two step sequence of mappings that decode a genotype to a program(phenotype):

1) **Genotype to derivation tree**: The genotype, an integer sequence, is read, one integer at a time, to rewrite non-terminals to terminal via the rules. This generates a rule production sequence which can be represented as a derivation tree. At each step in the rewriting the integer "gene" determines which specific element in the list of right hand side productions expands 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 in the same way as in 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, is based on performance of program on required task. GE's genotype-phenotype mapping allows crossover and mutation operators to operate on the genotype or the derivation tree [7]. One point crossover exchanges integers between crossover points on the genotype. Here, the interpretation step raises locality issues, however the grammar and the rewriting assure crossover will result in syntactically valid offspring [31]. 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 constitute the search space. Further, the language and grammar *biases* the likelihood of generating solutions within the search space [22].

GE has been used to investigate dynamic environments, defined as a spectrum from problems where the change is predictable to problems which is completely random [6]. The work on dynamic environments deals with multiple tasks, but the aim is often to solve the task at hand well, instead of finding a general solution as for multiple tasks, i.e. max q_i^t instead of max $\mathbf{q}$. Additionally, the impact of providing existing similar solutions and reverse mapping of solutions has not been investigated for program synthesis. In the next section we describe the methods we use for investigating transferability in program synthesis.

2.1.3 Multi Task Learning

Multi-task learning and the related optimization is concerned with how to solve multiple optimization problems (tasks) simultaneously to improve on the performance of solving each task independently. The assumption is that there exists some useful knowledge in common for solving related tasks and that the useful knowledge acquired when solving one task can assist in solving another task if they are similar [36]. In GP there have been multiple studies on GP regarding modularity, e.g. a survey of modularity in genetic programming [8], without explicit focus on multiple task.

Previous work in GP has focused on other domains than program synthesis. For example, multi task visual learning using genetic programming [14] creates populations of individuals where each individual is composed of trees that solves different visual tasks. Automatic generation and exploitation of related problems in genetic programming [18] studies symbolic regression and methods to deal with multiple tasks. Functional modularity for genetic programming [17] looks at boolean functions. More recent work studies automating knowledge transfer with multi-task optimization [28] for boolean problems with Cartesian GP.

There is some recent work for program synthesis with GP. For example, a study of simultaneous synthesis of multiple functions using genetic programming with scaffolding [4]. The parallel hierarchical evolution of string library functions [30] uses string manipulation in the HERCL programming language. Finally, there is an evolutionary framework for HPC [27] investigating Python code for 29 array/vector problems that uses a grammar which is a subset of Python.

One important area for multi-task learning is transferability. This is not solely concerned with learning multiple tasks within the search process itself; it also covers the utilization of existing solutions to guide the initial solutions. There has been work showing the importance of initialization [20]. In addition, there are studies about favorable biasing of function sets using run transferable libraries [26] which studies boolean problems and how to transfer solutions into libraries between runs. We investigate how to take existing programs(phenotypes) and "reverse" map them

to genotypes that we can add to the population.

Formally, multi-task learning deals with multiple programs $\mathbf{q} = \{q_0, \ldots, q_n\}$ Similar tasks have a low distance according to some measure, $\delta = f(q_i, q_j), 0 \leq \delta \leq C$. The language L must be capable of expressing all the programs in $\mathbf{q}$, $L_i \subseteq L$. Domain knowledge is expressed as existing solutions, $D = \{q_i, \ldots, q_j\}$.

2.2 Datasets Used

Here we describe the datasets used for the experiments in this thesis, along with covering the differences between them.

2.2.1 6.00.1x Dataset

The selection criteria of the GP benchmark suite was mostly concerned with problems that had input/output examples, multiple solutions, no synthesis method bias, and which represent actual programming tasks [11]. We need to expand on these criteria in order to investigate transferability and generalization in program synthesis. Thus, we identify programming problems that are similar according to human experts, composable into a combined program, obtained from learning designers and instructors, and that provide us with access to a corpus of correct and incorrect human solutions.

We select two new program synthesis problems that are identified as being similar by human experts. In our case, the instructor and learning scientists analyzed the learning design of *MITx 6.00.1x Introduction to computer science and programming in Python* (*6.00.1x*), a MOOC offered on the EdX platform [3]. They are also from an introductory programming course, so they should be as complex as the problems in the existing GP program synthesis benchmark suite [12].

The learning design for *6.00.1x* is such that the problems are gradually built on information presented in lecture videos and optional practice for students in short exercises(finger exercises). We focus on the first two problem sets *Problem Set 1-1* and *Problem Set 1-2*, these problem sets check the students understanding of control flow. We have submission history data from a large-scale scraping effort from 2016

Term 2 and 2017 Term 1. We consider only consider the behavior of the 3,485 certified students.

We also measure similarity based on python keywords. Based on the Pearson correlation the correct student submissions for the two problems we focus on are all quite similar to each other, and the "gold standard" solution. Around 3% of the correct solutions are dissimilar to the "gold standard" based on a similarity threshold of 50% [3].

Problem set 1-1: Count Vowels

```
def problem_set_1_1(s: str) -> int:
    """
    Assume `s` is a string of lower case characters.

    Write a program that counts up the number of vowels contained in
    the string `s`. Valid vowels are: `'a', 'e', 'i', 'o'`, and
    `'u'`. For example, if `s = 'azcbobobegghakl'`, your program
    should print:

    `Number of vowels: 5`
    """
    ctr = 0
    for i in s:
        if i == "a" or i == "i" or i == "o" or i =="e" or i == "u":
            ctr = ctr + 1
    print("Number of vowels:", ctr)
    return ctr
```

Existing solutions are not many. We reverse map $\approx 2,000$ solutions and end up with 2 distinct solutions.

Problem set 1-2: Count bob

```
def problem_set_1_2(s: str) -> int:
    """
    Assume `s` is a string of lower case characters.

    Write a program that prints the number of times the string `'bob'`
    occurs in `s`. For example, if `s = 'azcbobobegghakl'`, then your
    program should print

    `Number of times bob occurs is: 2`
```

```
    """
    ctr = 0
    for i in range(len(s) - 2):
        if s[i] == "b" and s[i + 1] == "o" and s[i + 2] == "b":
            ctr = ctr + 1
    print("Number of times bob occurs is:", ctr)
    return ctr
```

Existing solutions are even fewer. We reverse map $\approx 1,000$ solutions and end up with 1 distinct solution.

Combination of Problem Set 1-1 and 1-2 We create a combination of *Problem Set 1-1* and *Problem Set 1-2* called *Problem Set 1-1, 1-2 Combo*. Note that there are no existing solutions for this combination since we create this problem for `Multi-task Program Synthesis GP`.

```
def problem_set_1_1_2_combo(s: str) -> Tuple[int, int]:
    """
    Assume `s` is a string of lower case characters.

    Write a program that prints the number of vowels and number of
    times the string `'bob'` occurs in `s`. For example, if `s =
    'azcbobobegghakl'`, then your program should print

    ```
    Number of vowels: 5
    Number of times bob occurs is: 2
    ```
    """
    ctr_1 = 0
    ctr_2 = 0
    for i in range(len(s)):
        if i < (len(s) - 2) and s[i] == "b" and s[i + 1] == "o" and s[i + 2] == "b":
            ctr_2 = ctr_2 + 1
        if s[i] == "a" or s[i] == "i" or s[i] == "o" or s[i] == "e" or s[i] == "u":
            ctr_1 = ctr_1 + 1
    print("Number of vowels:", ctr_1)
    print("Number of times bob occurs is:", ctr_2)
    return ctr_1, ctr_2
```

We used the grammar in Figure 2-1 to both reverse map and generate solutions.

```
!start : initial_assign | "i0 = int(); i1 = int(); s0 = str();
 res0 = int(); res1 = int()\n" initial_assign
!initial_assign : (int_var equals num "\n" initial_assign)
                | (int_var equals num "\n" code)
                | (string_var equals "str()\n" initial_assign)
!equals : " =" | "=" | "= " | " = "
!plusequals : " +=" | " += " | "+=" | "+= "
!code :  (code statement "\n") | (statement "\n")
!statement : assign | compound_stmt
!compound_stmt :  for | if
!assign :  int_assign |  inc
!inc :  int_var plusequals int
!for :  for_iter_string
!bool :  bool_string | (bool_string  bool_op  bool)
!bool_op : " and "|" or "
!bool_string :  string_cmp | in_string
!in_string : "s0 in " str_tuple
!str_tuple : "(" s_or_comma ")"
!s_or_comma : string_alpha_low | (string_alpha_low ", " s_or_comma)
!if : ("if " bool ":{:\n" code ":}") | ("if (" bool "):{:\n" code ":}")
!number : num
!num : "0"|"1"|"2"
!int_var : "i0"|"i1"|"res0"|"res1"
!int_assign :  int_var "=" int
!int :  int_var | ("int(" number ".0)") | num
!string_var : "in0[i1+" num "]" | "s0" | "in0[i1]"
!string_cmp :  string_var string_equals string_alpha_low
!string_equals : "==" | " == " | "== " | " =="
!for_iter_string : ("for s0 in in0:{:\n" if "\n:}")
                 | ("for i1 in range(len(in0)-" num "):{:\n" if "\n:}")
!string_alpha_low : "'b'"|"'a'"|"'e'"|"'i'"|"'o'"|"'u'"
```

Figure 2-1: EBNF grammar for *Problem Set 1-1*, *Problem Set 1-2* and *Problem Set 1-1, 1-2 Combo*. Some of the formatting symbols were specific options to the Lark parser that was used [1].

2.2.2 GitHub Dataset

In order to generate a distribution for what metrics humanlike code has, we needed to get software from real-world, high-quality projects. There are many high-quality open-source codebases available online on GitHub - with this in mind, we decided to use the same GitHub corpus used in [29], which minded real-world, open-source projects online. From here on, we'll refer to this dataset as the **Github Corpus**. Only projects with over 150 stars on GitHub were mined, to ensure a higher quality of code. The GitHub corpus consists of programs that take in either 2, 3, or 4 arguments, and are used for a variety of applications. Class methods were excluded because they usually do relatively simple tasks. We used a total of over 40,000 Github programs, and each program consisted of the function definition and the corresponding code.

2.2.3 Comparison Between Datasets

Each of the datasets described above differs in terms of the number and diversity of the programs inside, along with the sources they were generated from.

The 6.00.1x dataset has the advantage of just looking at solutions to two problems that are very similar in structure, which are to count vowels and count the number of occurrences of the word "Bob". However, the one main drawback here is the fact that such a small number of programs were produced - only a few unique solutions were parsed for each problem, limiting the diversity. The Github dataset, on the other hand, contains thousands of programs, which is necessary in order to calculate a distribution of program metrics. Additionally, the code is high quality - each function was taken only from repositories with over 150 stars, which is quite a high bar. However, has its associated downsides. First, the dataset isn't specialized for a small subset of problems, as the 6.00.1x dataset is. Second, this diversity of programs has the potential to wash out meaningful information - if you compute the average over a large range of programs, then all the programs can start to blend in with each other.

Chapter 3

Transferability in GE Solutions

Traditional Genetic Programming methods for Program Synthesis have focused on solving a single problem at a time. However, many programs share subprograms - many programs utilize functions to iterate through the elements of an array, for example. In this chapter, we investigate the transferability of programs (solutions) for GP program synthesis. Specifically, we explore the effectiveness of optimizing a population for one task with GE before optimizing it for a similar, related problem, and investigate the use of intermediate goals for solving complex problems. We find that having a population solve one task before working to solve a related task can have positive results, and that choosing harder intermediate goals over easier ones can be beneficial.

We investigate the following research questions, restated from Chapter 1:

- **RQ1.1**: Can optimizing a GP program synthesis population for one task make it more efficient at solving a similar task?
- **RQ1.2**: Can GP program synthesis solve complex tasks by combining the solutions to simpler ones?
- **RQ1.3**: Does GP program synthesis benefit from the use of intermediate goals?
- **RQ1.4**: How can existing knowledge in the form of existing programs be incor-

porated in the GP search?

First we need to identify program synsthesis problems suitable for investigating transferability. The selection criteria of the existing GP benchmark suite was mostly concerned with problems that had input/output examples, multiple solutions, no synthesis method bias and represented actual programming tasks [11]. We need to expand on these criteria for our study of transferability in program synthesis, so we generate solutions for two python programming problems from *MITx 6.00.1x Introduction to computer science and programming in Python* (*6.00.1x*), a MOOC offered on the EdX platform. From this course we identify programming problems that are similar according to human experts, composable into a combined program, obtained from learning designers and instructors, and that provide us with access to a corpus of correct and incorrect human solutions [3].

We use a version of grammatical GP, `Multi-task Program Synthesis GP`, in which the task at hand is varied throughout the process evolution. We use information gained from solving one program synthesis problem (task) in another program synthesis problem (task). We look at the effectiveness of solving a task by initializing a population of existing programs that solve a similar task, for multiple combinations of similar tasks. The existing programs we use for initialization are combinations of random (normal GP initialization), reverse mapped from human solutions, and programs generated from previous runs of `Multi-task Program Synthesis GP`.

The chapter is structured as follows. Section 3.1 presents methods, Sections 3.2 and 3.3.4 present experiments and their results.

3.1 Method

We present our methods regarding transferability through multi-task learning and reverse mapping of existing solutions.

`Multi-task Program Synthesis GP` is shown in Algorithm 1. Our `Multi-task Program Synthesis GP` implementation is based on the `PonyGE2` [7] grammar guided genetic programming system, in particular the work by [13].

Algorithm 1 `Multi-task Program Synthesis GP`(D, S, D_K, G, Θ)
Parameters: $\mathbb{D}$ test cases, S task schedule, D existing solutions as knowledge base, G grammar, Θ hyper parameters
Return: Population

$P \leftarrow \emptyset$ ▷ Population
for $d \in D$ **do** ▷ Reverse map existing programs
    $P \leftarrow P \cup$ reverseMap(d, G) ▷ Reverse map, see Alg. 2
$P \leftarrow P \cup$ initialize(Θ, G) ▷ Initialize population
$F \leftarrow$ getTask$(S, 0)$ ▷ Get the fitness function for task(s)
$P \leftarrow$ evaluate(P, Θ, F) ▷ Evaluate pop fitness
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
    $F \leftarrow$ getTask(S, t) ▷ Get the fitness function for task(s)
    $P' \leftarrow$ evaluate(P', Θ, F) ▷ Evaluate population
    $P \leftarrow$ replacement(P', Θ) ▷ Update population
return P ▷ Return final population

The key GP extensions are the:

Multiple tasks Synthesizing programs for multiple tasks and switching between the different tasks. This is one way of providing more general knowledge between tasks.

Initialization with similar programs Initializing the population with programs that solve a similar task.

Reverse map of existing programs Parsing existing programs to an evolvable representation. This is one method of providing domain knowledge.

3.1.1 Multi task program synthesis scheduling

We look at both serial and parallel scheduling for multiple program synthesis. With serial schedule one task is first solved and then another. With a parallel schedule multiple tasks are solved at the same time. Existing solutions to tasks are also provided.

Serial One task, $q_i^{t=k}$ is solved, then another task, $q_j^{t=k+1}$ is solved. The set of tasks are $\mathbf{q}$ and $0 \leq i, j \leq |\mathbf{q}|, k \in \mathbb{Z}$

Parallel Multiple tasks are solved at the same time, $q_i \& q_j$

Serial & Parallel Tasks can be solved both in serial and parallel, $q_i^{t=k}$ and $q_i^{t=k+1} \& q_j^{t=k+1}$

Existing solutions Existing solutions to tasks are provided to the initial population *a)* Existing source code(solutions) for other programs, D^H *b)* Existing evolved solutions in the `Multi-task Program Synthesis GP` representation for other program synthesis task, D^S *c)* Randomly generated programs using the standard GP initialization procedures, R

There are many possible scheduling combinations for tasks. A few examples of task scheduling are shown in Figure 3-1.

Initialization with similar programs

Seeding the `Multi-task Program Synthesis GP` search with similar programs is one way of providing domain knowledge for the program synthesis. This seeding can be seen as an extreme variant of serial multi task switching, i.e. the first task has been synthesized to completion and provides a starting point for the next task. We provide two methods for initialization in addition to standard GP initialization methods.

Existing Human Source Code We parse existing source code, see Section 3.1.2, to the `Multi-task Program Synthesis GP` representation. This source code has been used to solve problem q_i.

Existing Evolved Source Code We run `Multi-task Program Synthesis GP` to solve a problem, q_i and use these solutions as the initial population for another problem q_j.

3.1.2 Reverse mapping of existing solution to `Multi-task Program Synthesis GP` representation

Overview of the reverse mapping of existing source code to `Multi-task Program Synthesis GP` representation is shown in Algorithm 2 and 3. The mapping first preprocess the program and refactors variables and function names to a consistent naming scheme. Then it recursively parses a program depth-first left-to-right and re-

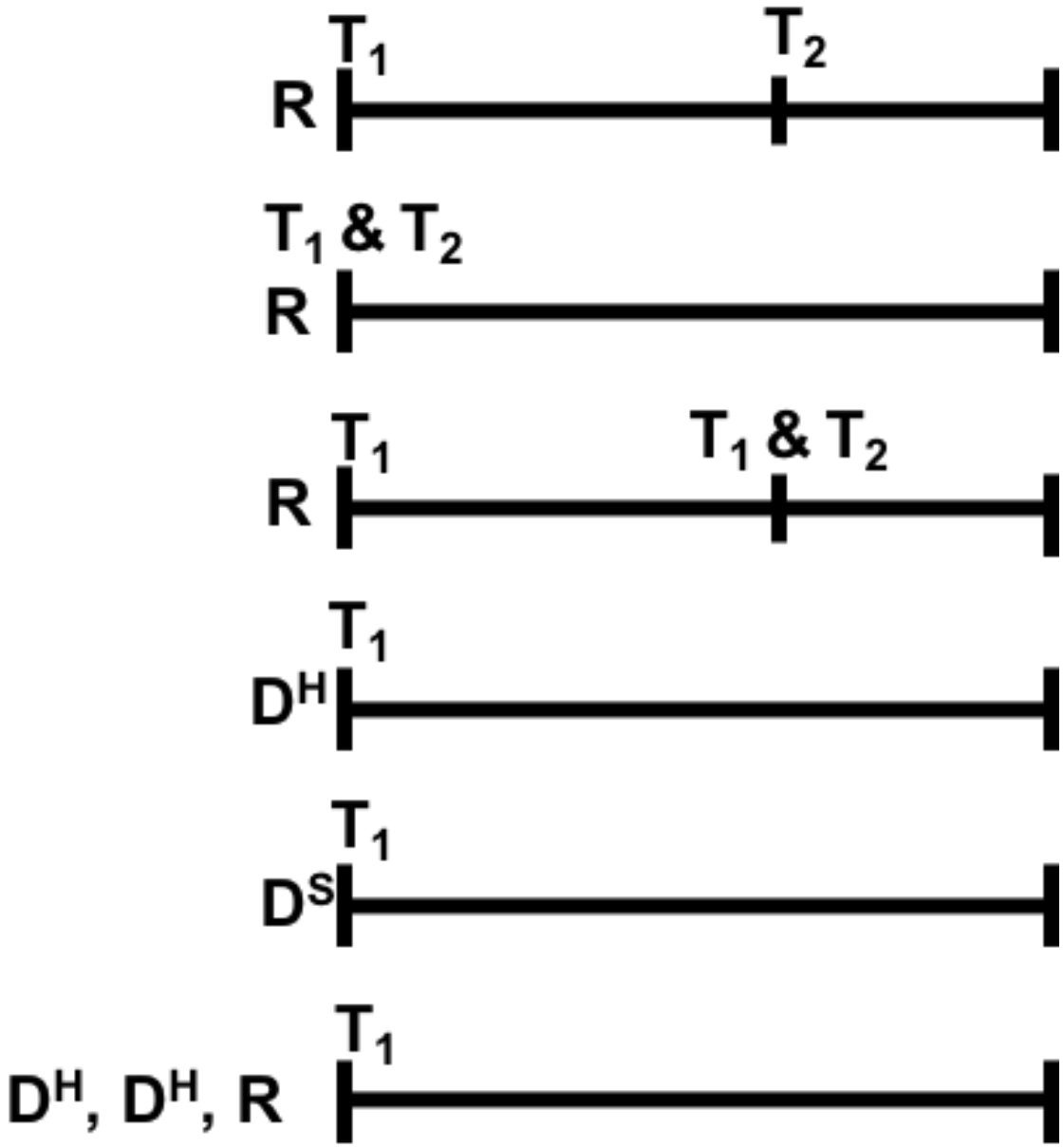

Figure 3-1: Examples of multi task scheduling with serial schedules, parallel schedules, and initialization with similar solutions. The examples are separated by the dotted lines.

Algorithm 2 $reverseMap(D_K, G, P)$
Parameters: D_K existing solution, G grammar, P parser
Return: List of integers

```
D'_K ← preprocessProgram(D_K)          ▷ Refactor variable and function names
S ← getStartSymbol(G)                  ▷ Get start symbol
p ← parse(D'_K, P, S)                  ▷ Parse solution to a tree
i ← reverseMapSubProgram(p, G, P)      ▷ Parse subprograms, see Alg. 3
return i                               ▷ Return GE representation
```

Algorithm 3 $reverseMapSubProgram(p, G, P)$
Parameters: p subprogram tree, G grammar, P parser
Return: List of integers

```
i ← []                                 ▷ Initialize GE genome
r ← getRules(p, G)                     ▷ Get rules
for i ∈ [0, ..., |r|] do               ▷ Iterate over rules
    r ← r_i                            ▷ Get rule
    M ← True                           ▷ Match variable
    if |r| == |p.children| then        ▷ Check number of children
        for j ∈ [0, ..., |r|] do       ▷ Iterate over productions
            M ← M&(r_j == p[j])        ▷ Does product match child node
    if M == True then                  ▷ Found match
        break                          ▷ Break iteration on first match
i ← i ∪ i                              ▷ Add production choice to genome
for c ∈ p.children do                  ▷ Recurse over children
    i ← i∪ reverseMapSubProgram(c, G, P)   ▷ Add production choice to genome
return i                               ▷ Return GE representation
```

turns a list of integers indicating production choices (GE genome). The first matching production will be returned.

In the next section we will describe the experimental setup and the results.

3.2 Experiments

We used the 6.00.1x dataset, described in Chapter 1 of this thesis, for the following experiments. Our setup and experimental design are designed in greater detail below.

Table 3.1: Experimental settings

Parameter	**Value**
Generations	200)
Population Size (P)	800
Elite size	8
Replacement	generational
Initialization	PI grow
Init genome length	200
Max genome length	500
Max init tree depth	15
Max tree depth	17
Crossover probability	0.8
Mutate duplicates	True
Novelty selection [15]	
Knobelty archive sample size (C)	100
Knobelty tournament size (ω)	6
Knobelty function	Exponential
Knobelty λ	Generations/10

3.2.1 Setup

We report results on 100 runs. We report program synthesis performance in the same way as previous work [12, 13], 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 3.1. We use subtree crossover. Fitness is the number of correct test cases solved during training, also same as [12].

3.2.2 Experimental Design

In this work, we explored two main experimental approaches: Multi Task Learning, and Reverse Mapping. For each approach, multiple variants were tested across a range of parameters. For Multi Task Learning, we specify both the Task Schedule (tasks to be solved, and in what order) and the relative amount of time to spend on each task. For Reverse Mapping, we specify which set of solutions we initialize the population with, along with the percentage of the population that is initialized from those solutions. The variants used in the experiments are described in Table 3.2. The

Table 3.2: Experimental Variants

Variant Name	Feature
	Multi task learning
EarlySingleSwitch	Change task after 25% of generations have passed
MedSingleSwitch	Change task after 50% of generations have passed
LateSingleSwitch	Change task after 75% of generations have passed
OneThenTwo	Initially optimize for *Problem Set 1-1* and switch to *Problem Set 1-2*
TwoThenBoth	Initially optimize for *Problem Set 1-1* and switch to *Problem Set 1-1, 1-2 Combo*
TwoThenOne	Initially optimize for *Problem Set 1-2* and switch to *Problem Set 1-1*
TwoThenBoth	Initially optimize for *Problem Set 1-2* and switch to *Problem Set 1-1, 1-2 Combo*
	Reverse mapping
HalfFromDir	Initialize 50% of the population from the given directory of solutions, generate the rest randomly
AllFromDir	Initialize 100% of the population from the given directory of solutions, generate the rest randomly
UserSolutions	Infuse population with student-submitted solutions
GeneratedSolutions	Infuse population with solutions evolved to solve the task by a previous GP run
NonDiverse	Infuse population with multiple copies of a single program that doesn't solve either task

names in the figures are concatenations of these. Baseline refers to a run without multi tasks and random initialization.

3.3 Results

Table 3.3 outlines the main experiments we ran, along with their results. We report the number of runs in which at least a single program passed all of the test cases by the last generation. Below we delve into the major takeaways and analysis from the experiments.

3.3.1 All problems are not created equally

Although the three problems are similar in structure, they have varying difficulties given our grammar. Figure 3-2 shows this difference - given the same population size and number of generations, going after the *Problem Set 1-1, 1-2 Combo* alone is extremely difficult, and *Problem Set 1-1* appears to be much more easily solvable than *Problem Set 1-2*. This could be due to the fact that each of the vowels in *Problem*

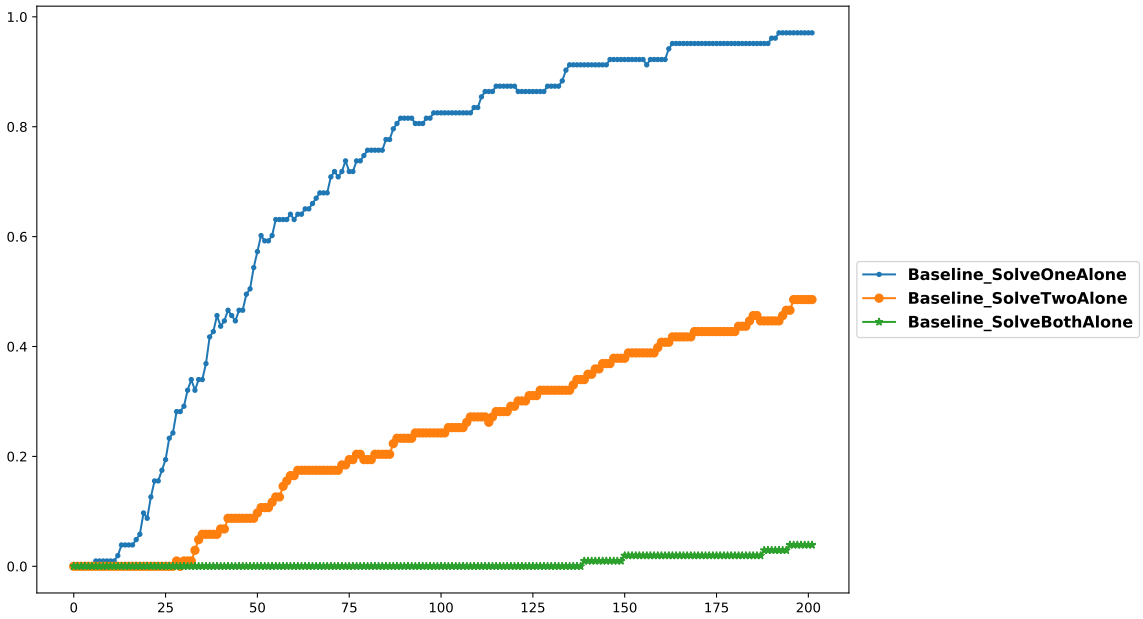

Figure 3-2: Percentage of runs which contained a program that solved all of the test cases, when trying to solve a single problem (*Problem Set 1-1*, *Problem Set 1-2*, or *Problem Set 1-1, 1-2 Combo*)

Set 1-1 gives an additional signal that the program is doing something right, a sort of gradient - every time the program stumbles upon an additional vowel incorporated into the letters it checks for, it solves additional test cases. This will be important in the following sections.

3.3.2 Solving Two Problems at the Same Time: Choose a Harder Intermediate Task

Here we try to answer **RQ1.2, 1.3**: How do intermediate goals affect the GE search, and can GE combine solutions to simpler tasks together to solve a more complex problem? For solving *Problem Set 1-1, 1-2 Combo*, useful intermediate tasks are to solve each of the *Problem Set 1-1* and *Problem Set 1-2* problems individually before moving to solve the combined problem. With this in mind, we performed multi-task learning by starting from either *Problem Set 1-1* or *Problem Set 1-2* and changing the task to solving *Problem Set 1-1, 1-2 Combo* at some point throughout the evolution. The percentage of runs in which at least a single program solved all given test cases

is given in Figure 3-3. In this plot, each of the runs was given 200 generations to run. However, they are displayed as being shifted in order to line up the point at which they started working on the final problem; for example, if a population started by spending 50 generations on *Problem Set 1-1* and then switched to spending 150 generations on *Problem Set 1-2*, then it would be shifted back 50 generations relative to the rest of the lines, so that there's a common point where all problems start the second problem.

One thing to notice is that spending more time optimizing for the intermediate task generally had positive results - populations that spent more time optimizing for the subtask (LateSingleSwitch) generally had a higher upward trajectory than those that spent less time optimizing for the subtask (EarlySingleSwitch). This helps us answer **RQ1.3**, in that GE can benefit from intermediate goals. It also helps us answer **RQ1.2**: because *Problem Set 1-1, 1-2 Combo* is modular and needs solutions to both *Problem Set 1-1* and *Problem Set 1-2*, this shows that when the grammar allows for the easy combination of multiple subproblems, a mopre complex task can be easier to solve. One reason for this could be that spending more time optimizing for the intermediate task infuses the population with more individuals that can competently solve test cases, so when the population is faced with the more complex problem, it's individuals are each more likely to already have the solution to the intermediate problem, and thus can solve the combination problem easier. This effect can be seen in Figure 3-4, which shows the number of test cases solved by the best individual in a run, averaged across all runs - keeping the problem constant, a longer time spent optimizing for the intermediate task allows a population to solve a higher number of test cases from *Problem Set 1-1, 1-2 Combo* earlier as compared to those that spent less time optimizing on the intermediate task. Additionally, this experiment revealed the fact that harder problems can sometimes be a better intermediate signal than easier ones - *Problem Set 1-1* is an easier problem to solve for the population than *Problem Set 1-2*, and Figure 3-3 shows that the performance is much better when optimizing for *Problem Set 1-2* before *Problem Set 1-1, 1-2 Combo*, as opposed to optimizing for *Problem Set 1-1* before *Problem Set 1-1, 1-2 Combo*. This could be

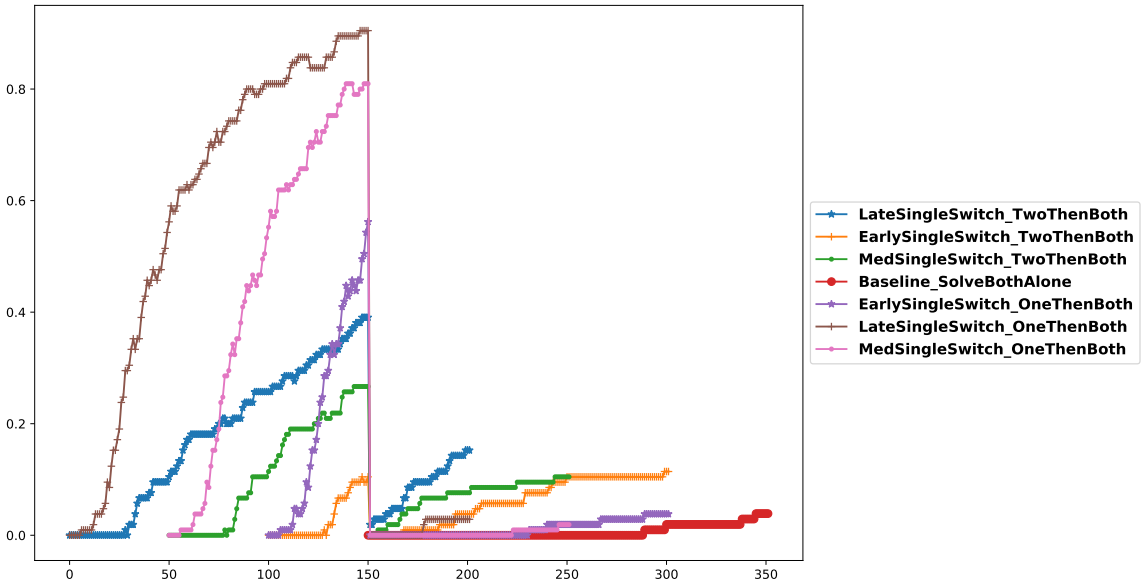

Figure 3-3: Percentage of runs which contained a program that solved all of the test cases for the multi task learning experiment, when switching from solving *Problem Set 1-1* to *Problem Set 1-1, 1-2 Combo*.

because harder problems take more evolutionary leaps or changes to reach, and having the constraint of keeping existing solutions to subproblems intact as you optimize for a new task can make it harder for a given individual to solve a new problem. That is, a population who only needs to solve *Problem Set 1-1* will likely be better off than a population who needs to solve both problems, but who already has a solution to *Problem Set 1-2*. With this in mind, the number of evolutionary leaps needed to reach *Problem Set 1-1, 1-2 Combo* starting from a solution to *Problem Set 1-2* is likely much less than what's needed when starting from *Problem Set 1-1*.

3.3.3 Transferring Information from One Task to Another

Here we explore **RQ1.1**: does optimizing GE for one task make it more effective at solving a similar task. The two problems we looked at were very similar, in that they involved initializing a count variable, iterated through a string, and added to that total count if some condition was met. Because they share similarities, our original hypothesis would be that optimizing for one task first would be beneficial when optimizing for the other. Specifically, a population that had previously been

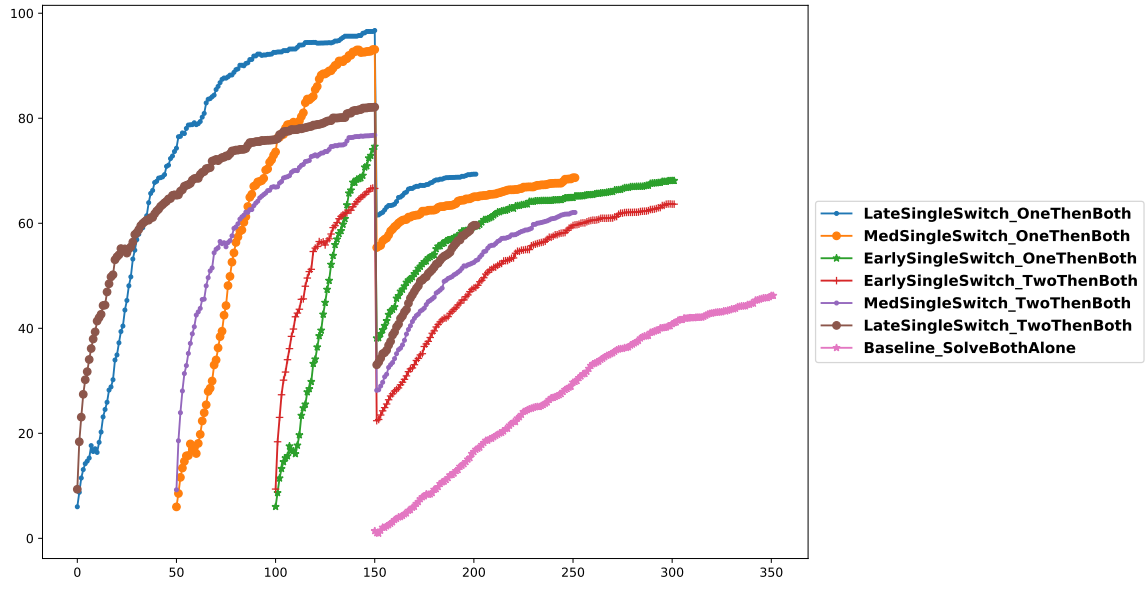

Figure 3-4: Average number of test cases solved by the best individual in each given population, when switching from solving *Problem Set 1-1* to *Problem Set 1-1, 1-2 Combo*.

optimized for solving *Problem Set 1-1* would be better equipped to solve *Problem Set 1-2* than a population solely focused on *Problem Set 1-2* from the beginning, and we tested this idea in Figures 3-6 and 3-5

From the results, we can see that the effect of optimizing for a previous task was different in each case. When going from *Problem Set 1-1* to *Problem Set 1-2* in Figure 3-6, pre-optimization seemed to have no effect on progress, as the population performed basically identically in each case. However, when going from *Problem Set 1-2* to *Problem Set 1-1* in 3-5, this pre-optimization seemed to produce positive results, as the population performed better than what it originally would have on the baseline in one case.

Looking at Figure 3-5, we can see that while too much pre-optimization on a related problem can be harmful, pre-optimizing the right amount can be beneficial - While the MedSwitch and LateSingleSwitch schemes perform worse than the baseline, the EarlySingleSwitch performed better than the baseline result, even when including the fact that 25% of the time was spent optimizing for a different problem. While further work needs to be done to study the cause for this one-way information transfer,

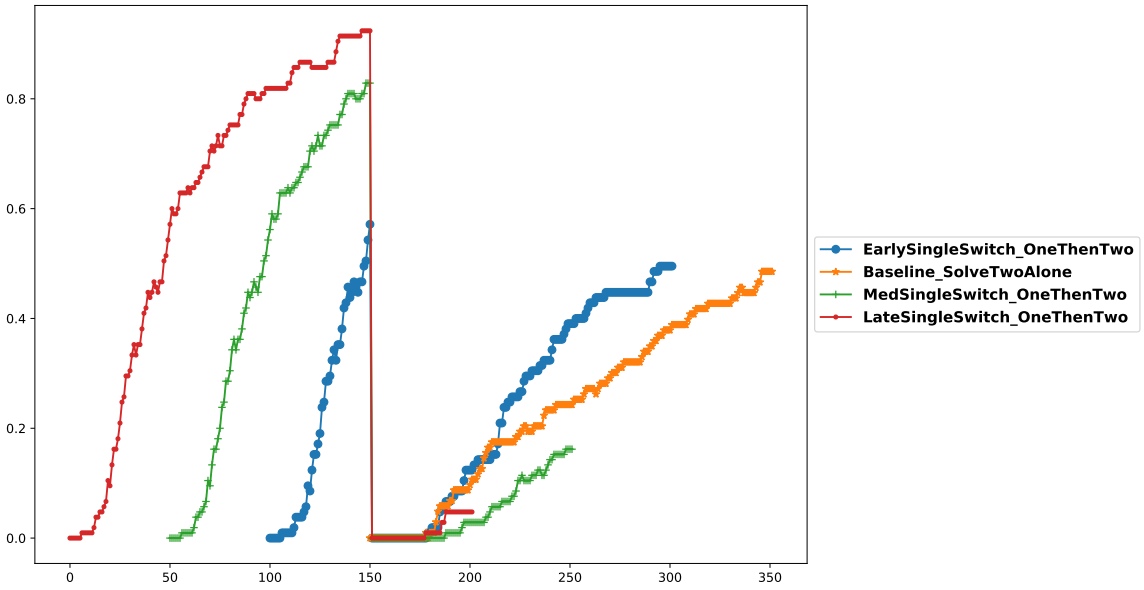

Figure 3-5: Percentage of runs which contained a program that solved all of the test cases, when the task was switching from solving *Problem Set 1-1* to *Problem Set 1-2*

for now it can be said at least that there can be one-way flow of information between populations optimized for different tasks; in other words, given problems A and B, optimizing for one A before B can be neutral, while optimizing for B before A can be beneficial. To answer **RQ1.1**: while pre-optimizing for a similar task can be beneficial to the search, many other factors are also involved, such as how long the population is optimized for the first problem.

3.3.4 Initializing with Preexisting Knowledge

Here we aim to answer **RQ1.4**: how can knowledge from existing programs be incorporated into GP? Because of the similarity in the solutions to each of the given problems, our initial hypothesis was that initializing populations with individuals that solved *Problem Set 1-1* would be better equipped at solving *Problem Set 1-2*, and vice versa; the thought was that submodules useful for solving one problem could be re-purposed for solving another problem, quicker than that feature being developed on its own. This idea was tested, and the results of these tests are given in Figures 3-7 and 3-8.

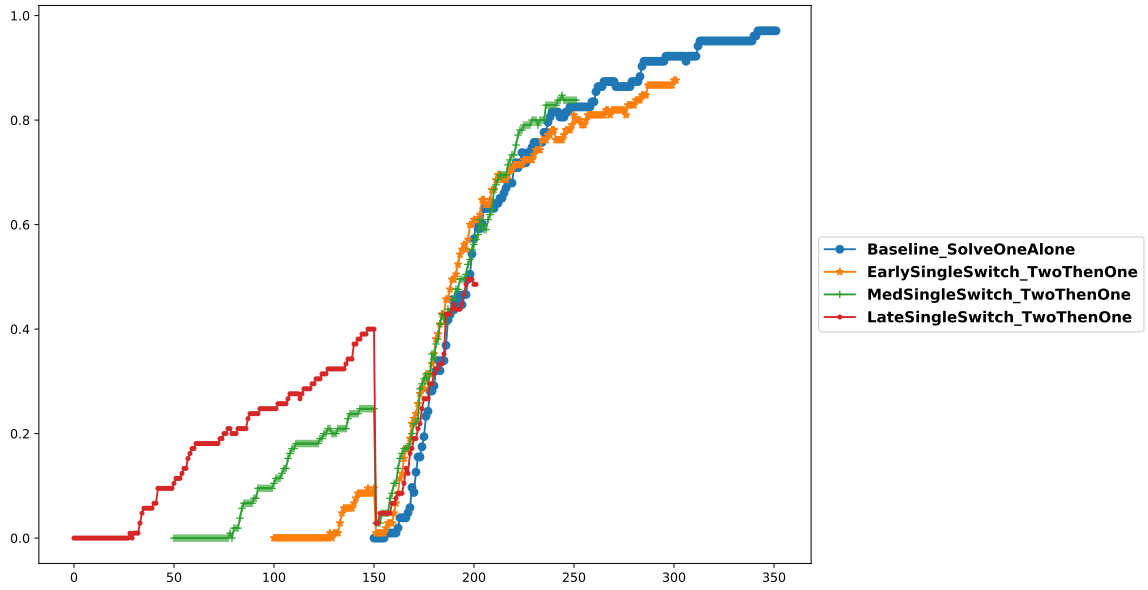

Figure 3-6: Percentage of runs which contained a program that solved all of the test cases, when the task was switching from solving *Problem Set 1-2* to *Problem Set 1-1*

One thing to notice here is that none of the runs initialized with solutions to one problem did any better than the runs in which random initialization methods were used. Additionally, when a single program which solved neither problem was chosen as a starting point, performance was not hindered - in fact, it even solved the problem in a higher number of runs than the baseline in Figure 3-7. While a decrease in diversity could be attributed to the decreased performance of the populations in which Human and Generated solutions were used during initialization, this doesn't seem likely given the fact that the NonDiverse initialization was competitive with the baseline.

To answer **RQ1.4**: while this experiment isn't able to conclude how existing programs can be used to accelerate GE, it doesn't rule out that they can be useful. One thing that's very likely is that the grammar and operators need to be further optimized to allow for the sharing of information between populations - while in this experiment, the grammar didn't easily allow for structure of one problem to transfer over to the structure of the other, section 3.3.2 shows the benefits of a grammar that can easily incorporate knowledge from one task in solving a more complex task.

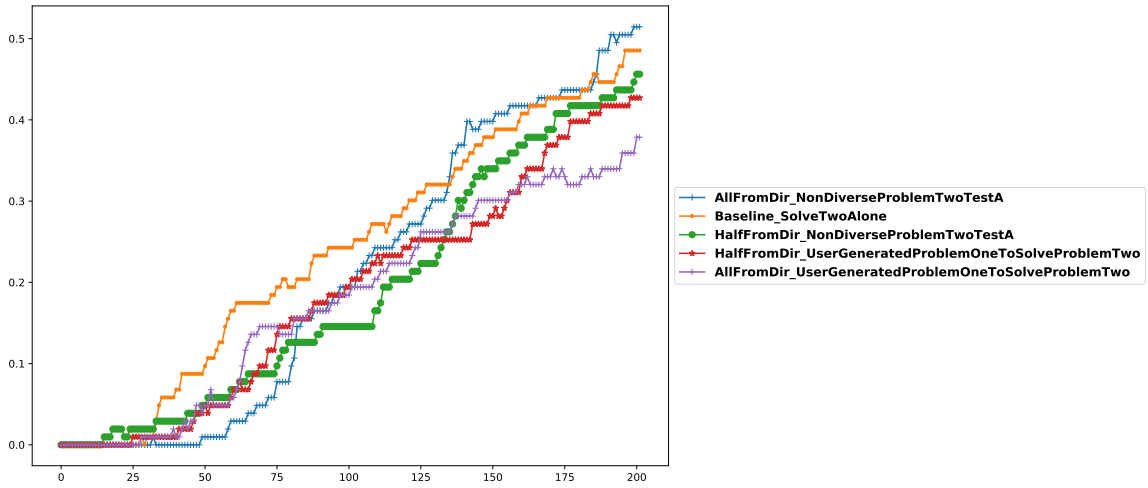

Figure 3-7: Percentage of runs which contained a program that solved all of the test cases, using various initialization strategies, when solving *Problem Set 1-2*.

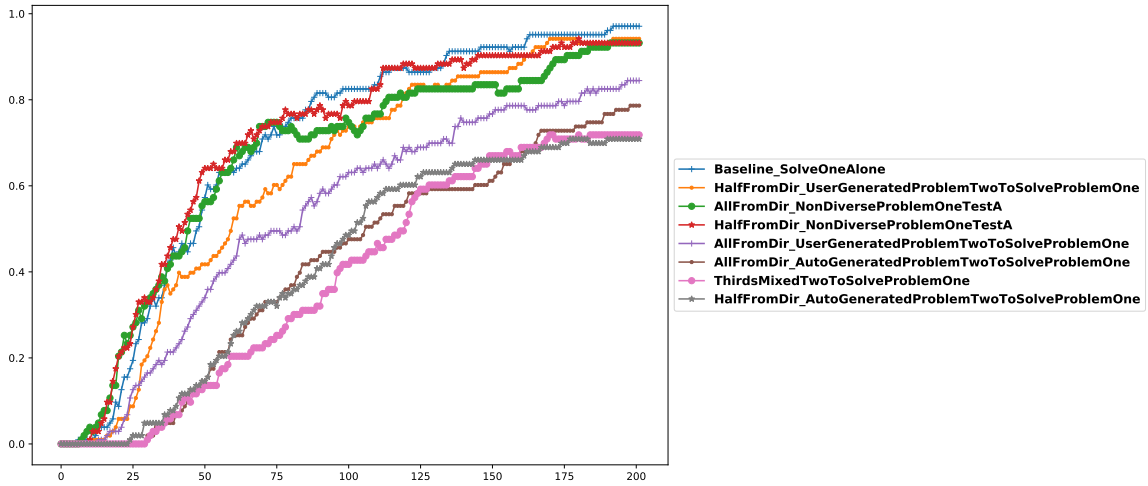

Figure 3-8: Percentage of runs which contained a program that solved all of the test cases, using various initialization strategies, when solving *Problem Set 1-1*.

Table 3.3: Experimental Results. Percent of runs solving all test cases at end is in the **Solved%** column for each variant.

Variant	**Solved%**
Baseline	
Problem Set 1-1	97 (Best for *Problem Set 1-1*)
Problem Set 1-2	48
Problem Set 1-1, 1-2 Combo	3
Switching	
Problem Set 1-2 then *Problem Set 1-1*, Early Switch	87
Problem Set 1-2 then *Problem Set 1-1*, Midway Switch	83
Problem Set 1-2 then *Problem Set 1-1*, Late Switch	48
Problem Set 1-1 then *Problem Set 1-2*, Early Switch	49 (Best for *Problem Set 1-2*)
Problem Set 1-1 then *Problem Set 1-2*, Midway Switch	16
Problem Set 1-1 then *Problem Set 1-2*, Late Switch	4
Problem Set 1-1 then *Problem Set 1-1, 1-2 Combo*, Early Switch	3
Problem Set 1-2 then *Problem Set 1-1, 1-2 Combo*, Early Switch	11
Problem Set 1-1 then *Problem Set 1-1, 1-2 Combo*, Midway Switch	1
Problem Set 1-2 then *Problem Set 1-1, 1-2 Combo*, Midway Switch	10
Problem Set 1-1 then *Problem Set 1-1, 1-2 Combo*, Late Switch	2
Problem Set 1-2 then *Problem Set 1-1, 1-2 Combo*, Late Switch	15 (Best for *Problem Set 1-1, 1-2 Combo*)
Initialization Schemes	
Problem Set 1-1, Human Solution, AllFromDir	84
Problem Set 1-1, Human Solution, HalfFromDir	94
Problem Set 1-1, Generated Solution, AllFromDir	78
Problem Set 1-1, Generated Solution, HalfFromDir	70
Problem Set 1-1, Human + Generated + Random (Third of Each)	71
Problem Set 1-1 Non Diverse Random, AllFromDir	93
Problem Set 1-1 Non Diverse Random, HalfFromDir	93
Problem Set 1-2, Human Solution AllFromDir	37
Problem Set 1-2, Human Solution HalfFromDir	42
Problem Set 1-2 Non Diverse Random, AllFromDir	51
Problem Set 1-2 Non Diverse Random, HalfFromDir	45

Chapter 4

Using properties of human-generated code to guide program synthesis with grammatical evolution

Program synthesis with GP produces code that looks different than human-generated code [29], and often includes extraneuous code [24]. Using the Github Corpus, we measure properties describing human-generated code, along with a context-sensitive measure. We use this knowledge to guide the search by incorporating a new selection scheme. The selection combines lexicase selection and a score based on the difference in code metrics and properties as well as context-sensitivity.

In this chapter, we aim to explore the following research questions:

1. **RQ2.1**: Does optimizing directly for a desired value for these code metrics actually lead to a population where these metrics take on the desired value?

2. **RQ2.2**: Can optimizing directly for these code metrics result in increased test case performance?

3. **RQ2.3**: Does optimizing for these code metrics make our solutions make our solutions more humanlike, by reducing the program bloat?

4. **RQ2.4**: Is is easier to find solutions when starting from pieces of code that

have been optimized to be more humanlike?

4.1 Methods

Below we describe the algorithm used to generate our new, context-sensitive code metric, based on AST Counts.

4.1.1 Context-Based AST Metrics

Here we describe how we used context-sensitive AST counts to compute a score for each of the programs. At a high level, our method looks at the distribution of AST node types (`BinOp`, `WhileNode`, `ForNode`, etc) found within a certain context (within a While loop, within a For loop, at the top-level of the program, etc), compares this distribution with a given distribution (the corresponding distribution from our corpus of GitHub programs), and the overall similarity between the two averaged across the entire program is given as the score. For the rest of this section, we'll first talk about how we compute each context, and then move into how we compute scores from these contexts.

Context Data Structures

Each program is composed of an AST with different types of nodes - nodes representing everything from While loops, For loops, Expressions, Binary Expressions, and anything else. For this work, we chose to use three of these node types - While loops, For loops, and Expression nodes, as the context, where all nodes beneath a While node, for example, would be considered to be in the context of that While node. We represented this using a trie-based data structure, outlined below.

Trie Based Data Structure Context is represented it as a Trie object which stores node counts. Each Trie represents a specific context, and has a list of children for each type of node we want to compute the context (in this case `For`, `While`, and `Expr`

nodes), which are Tries themselves. This representation has the advantage of storing the order of contexts (whether the program is inside a `While` loop inside of a `For` loop, and not the other way around, for example).

How to Calculate Humanlike Score

Once we calculate the number of each type of node in each context for a given program, we compare this to a given distribution of counts averaged across an entire corpus of programs. For each node in our AST, we calculate the probability of that node type occurring in the given context accross all programs in our chosen corpus. Formally, let C be the set of all contexts, T be the set of all node types, and define $N(t|c)$ to be the number of nodes of type t at a given context c. Then, this probability is written as

$$P_{corpus}(t|c) = \frac{N(t|c)}{\sum_{t_i \in T} N(t_i|c)} \tag{4.1}$$

.

Once we have the probabilities for each node type, there are multiple ways we can combine them together to compute a final score, detailed below.

Regular Score In this regime, we multiply all of the probabilities together to compute the final score, multiplied by a weighting function $W(t)$ (described in the next section). Because this resulting number will be very small and susceptible to underflow errors, we convert everything to log probabilities. Let Q_{corpus} be the set of all programs in a given corpus, and q be a program. Our score $S(q)$ for a given program q is then:

$$S(q) = \sum_{node \in q} log(W(t_{node}) * P_{corpus}(t_{node}|c_{node})) \tag{4.2}$$

where t_{node} is the type of the node, and c_{node} is the context of the node.

Normalized Score The previous regime favors programs which are shorter; to correct for this, we divide by the number of nodes in the AST of the program being

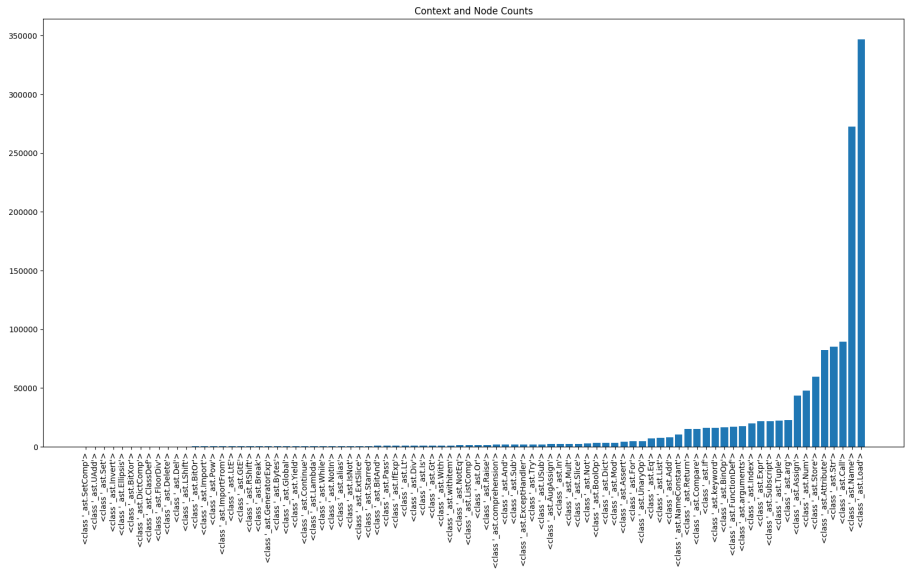

Figure 4-1: Frequency of each type of AST Node

looked at. This results in the following normalized score:

$$S(q) = \frac{\sum_{node \in q} log(W(t_{node}) * P_{corpus}(t_{node}|c_{node}))}{|q|} \tag{4.3}$$

where $|q|$ is the number of nodes in the given program's AST.

Node Importance

In english, some words carry more meaning than others - for example, whether or not the sentence contains the most common word, "the", gives no information, whereas the less common term "Laplacian" carries much more information. Different nodes also have vastly different frequencies - this is shown in Figure 4-1, which shows the different. With this in mind, we chose to incorporate two different weighting schemes:

Equal Importance In this scheme, each node type has just as much influence as any other type. This is given by

$$W(t_{node}) = 1 \tag{4.4}$$

Inverse Frequency Importance In this weighting scheme, each node is given an importance inversely proportional to the frequency with which it occurs across the github corpus of programs. This is given by

$$W(t_{node}) = \frac{\sum_{node' \in corpus} N(t_{node'})}{N(t_{node})} \tag{4.5}$$

where $N(t)$ is the total number of occurences of this node type across all contexts, across the entire corpus of programs.

Selection Based on Humanlike Score

Selecting for programs solely based on maximizing the above metrics likely wouldn't produce positive results - the goal in that case is no longer tied to the particular task at hand. With this in mind, we use a scheme that allows a variable level of importance to be placed on the humanlike score, using what we'll call our *H-Factor*. The *H-Factor* is a number between 0 and 1, and determines what percentage of the population will be evaluated solely based on test case lexicase selection, and which would be selected for based solely on the Humanlike score. For example, if *H-Factor=.6*, then 60% of the individuals in a population would be compared against each other and selected using only the Humanlike score, and the other 40% would be compared against each other using lexicase selection on the test cases. This factor allows us to tweak the importance of the score being used in the population. Algorithm 4 gives pseudocode for this method.

4.2 Experimental Setup

Two goals of our experiments were to see if optimizing directly for these properties would significantly change the distributions of these properties, and additionally if directly optimizing for these properties leads to better performance in solving the test cases. In order to answer these questions, we performed experiments in which we optimize for these metrics with varying *H-Factor*s. Then, we analyze the test case

Algorithm 4 *LexicaseAndHumanlikeSelection*(P, H, N, T)
Parameters: P: Population undergoing selection, H: *H-Factor*, N: Population size, T: Tournament size
Return: Population

$P_{new} \leftarrow \emptyset$ ▷ Population
while do$|P| < N$ ▷ Run until we've selected enough individuals
    $C \leftarrow sample(P, T)$ ▷ Sample T competitors C from P uniformly at random
    if $random() < H$ **then** ▷ With probability H, select based on Humanlike score
        $winner \leftarrow MostHumanlike(C)$ ▷ Get individual with highest Humanlike score
    else
        $winner \leftarrow LexicaseSelection(C)$ ▷ Perform lexicase tournament on C
    $P_{new} \leftarrow P_{new} \cup winner$ ▷ Add winner to the growing population
return P_{new} ▷ Return final population

performance and the distribution of scores in our population before and after our GE run, relative to a baseline run on the same problem. These relate to **RQ2.1, 2.2, 2.3**.

We also wanted to get a better sense of how these metrics compare relative to random search, to get an idea for how informative they are for solving the task at hand. This relates to **RQ2.2, 2.3**.

We want to see if code that has been optimized for these metrics provides a better starting point for GE than current initialization methods, while also seeing how these metrics change as we optimize for different objectives. This relates to **RQ2.3, 2.4**.

There were four main types of experiments we ran. Each experiment is marked with the main research questions it aims to answer.

1. Regularize for similarity to Github Corpus **RQ2.2, 2.3** - Running the experiments with various permutations Score Type and Node Importance schemes, across a set of given benchmark problems.

2. Regularize for similarity to Gold Standard Solution **RQ2.2, 2.3**- Instead of optimizing the population to match the distribution across an entire corpus of programs, just match the distribution to that generated by a single gold-standard solution.

3. Comparison to Random Search **RQ 2.1, 2.2, 2.3** - Compare GE with an *H-*

Variation	Meaning
NoFactor	*H-Factor=0*
SmallFactor	*H-Factor=0.1*
MedFactor	*H-Factor=0.2*
LargeFactor	*H-Factor=0.5*
HugeFactor	*H-Factor=1.0*
RegularScore	Return raw probability
NormalizedScore	Divide score by length of AST
AllImportant	All nodes equally weighted
InverseFrequency	Node weight inversely proportional to frequency
GoldStandard	The corpus for the Humanlike score is a single, gold-standard solution

Table 4.1: Setup Number Meanings. The values here are the values that are used in the experiments below. For example, an experiment with NormalizedScore in its name will use the Normalized Score described in the methods section.

Factor=1.0 to a run in which test case fitness is ignored entirely in selection, in order to determine how these metrics compare to random noise.

4. Humanlike Initialization **RQ2.4** - Running experiments while selecting only for the Humanlike score defined above, and then using the resulting population to initialize a GE run with the normal settings, given in Table 4.2

Table 4.1 lists the different parameter settings we used.

In this section, we used the Github dataset mentioned in Chapter 1. For each program, we generated an AST using Python's built-in AST module. From this, we were able to extract each of the stats mentioned above, along with the metrics that we'll define in the next section.

The parameters used in our experiments are shown in Table 4.1.

The bar charts in Figures 4-2, 4-3, 4-4, 4-5, and 4-6 show the percentage of the test cases solved by the best program in each run, averaged across all runs. So the higher the number, the higher the number of test cases the best individual in a population solved, on average. These results were averaged over 23 runs unless otherwise specified. We define the **Raw results** as the percent of test cases solved by the best program in each run, averaged across all runs for a given set of parameters. Raw results can be found in the Appendix A.

Parameter	Value
Crossover Probability	0.8
Crossover Type	Subtree Crossover
Mutation Probability	0.05
Mutation Type	Point Mutation
Generations	150
Population Size	1000
Selection	Lexicase
Mutate Duplicates	True
Elite Size	1

Table 4.2: Baseline Parameter Settings - these are the parameters used for all GE Runs, unless otherwise specified for a specific run.

Tables 4.4, 4.5, 4.7, and 4.9, and contain the rankings of each approach for each experiment, measured in terms of the number of test cases solved by the best individuals in the population, averaged over all of our runs. The setup that did the best gets rank 1, the second best gets rank 2, and so on, and if two individuals get the same result, they recieve the same rank. This is summed up across all tasks, so the setup with the lowest rank did roughly the best across all the problems. We recognize that this is a relatively noisy metric, however, which is why we still provide the raw scores for each of the runs.

Tables 4.3, 4.6, 4.8, 4.10, and 4.11 show the significance of our experiments, using the Mann-Whitney test for significance [9]. In these tables, our null hypothesis is that the median difference between the baseline and the experimental setup in question ($baseline - experiment$) is positive, and we list the **p-values** for this null hypothesis. When we reject this null hypothesis, the alternative hypothesis is that the median is negative, meaning the median value from the experiment is higher than the median value generated by the baseline setup. This means that when the null hypothesis is rejected, the median number of test cases the experimental variant solves is higher than the baseline. We set our confidence interval to by 95%, meaning a **p-value** has to be less than 0.05 in order to be considered significant.

4.3 Results

In this section, we'll walk through the results of our experiments, with a discussion of each. We'll go through the thought process behind the experiments that were conducted, and then discuss pitfalls and successes of each.

4.3.1 Regularize for similarity to Github Corpus

In this regime, we used the Humanlike score with various settings of the parameters described above. This was done in order to answer **RQ2.2** and **RQ2.3**: To see if optimizing for this metric can improve tst case performance and readability of the generated programs. When we conducted the experiments, few results were better than the baseline, and most were on par or worse. This can be seen from the raw test case results, the relative rankings of each setup, and the significance tests for each experiment. In the rankings specifically, we see that when we used *H-Factor = 0.5*, the rankings worsened significantly, which goes to show that optimizing too much for matching the humanlike distribution of code can be detrimental to the performance of the population. The significance test shows that none of the results were significantly better than the baseline with a confidence interval of 95%.

For the CountOdds, Digits, Median, and Smallest problems, performance seemed to be similar to the baseline when *H-Factor=0.1 - 0.2*.

One thing that Figure 4-2 showed was that factors helped when running the VectorAverage - in this problem, many of the setups had an apparent improvement over the baseline, which is a theme that will repeat itself later on. The results weren't significant according to Table 4.3, though.

Our thought from this was that even though the distribution of AST nodes across a large corpus of programs reaches some average, that individual programs each have a distribution that is different than the average. For example, while maybe the average tree depth across all programs is found to be 8, it may be the case that most solutions for a particular problem will end up having a tree depth of 13 because of increased problem complexity, causing our methods to lead the search astray. With this in

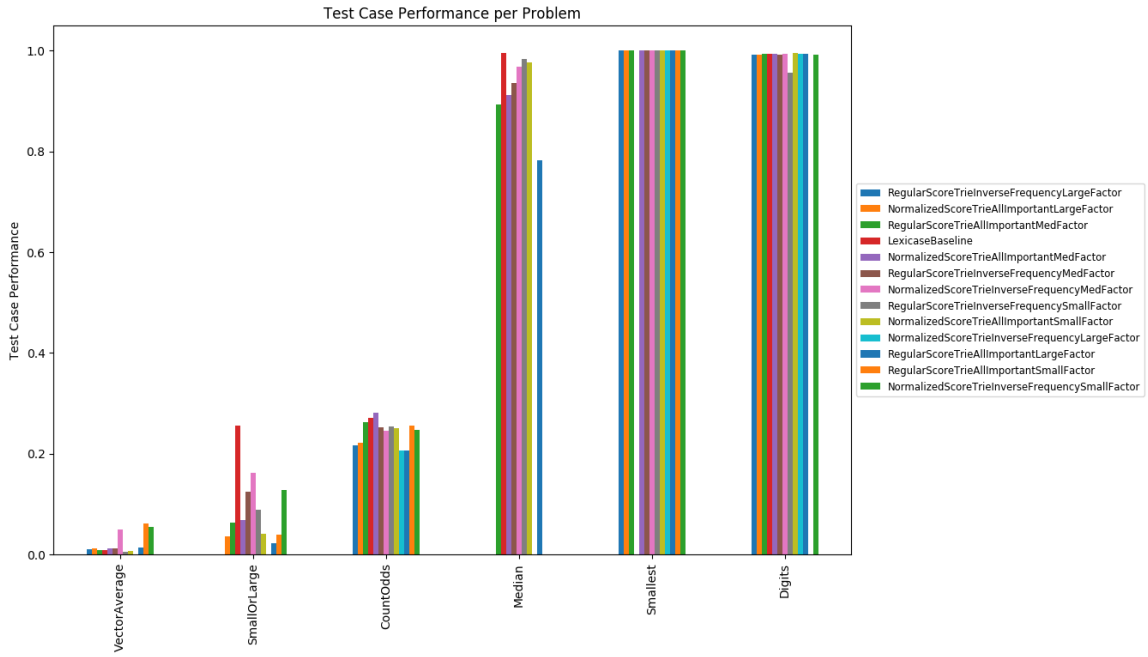

Figure 4-2: Results of Regularizing for similarity to Github Corpus. Measured in average best percentage of test cases solved per run.

mind, it could be the case that optimizing for the average distribution of AST nodes at different contexts could actually wash away information specific to the individual problem at hand, as the program doesn't need to be optimized for the average, but rather for some distribution specific to the problem at hand, which can vary widely. This observation motivated the next set of experiments.

Experiment Name	SmallOrLarge	Median	CountOdds	Digits	Smallest	VectorAverage
NormalizedScore, AllImportant, LargeFactor	0.9999	0.9362	1.0000	0.9997	N/A	0.3401
NormalizedScore, AllImportant, MedFactor	0.9927	0.6200	0.9982	0.9959	N/A	0.4952
NormalizedScore, AllImportant, SmallFactor	0.9921	0.5051	0.7290	0.9995	N/A	0.8455
NormalizedScore, InverseFrequency, LargeFactor	1.0000	0.8163	1.0000	1.0000	N/A	0.6173
NormalizedScore, InverseFrequency, MedFactor	0.9988	0.7261	0.9945	0.9589	N/A	0.5332
NormalizedScore, InverseFrequency, SmallFactor	0.9880	0.9362	0.5447	0.9920	N/A	0.6268
RegularScore, AllImportant, LargeFactor	1.0000	0.7261	1.0000	1.0000	N/A	0.0780
RegularScore, AllImportant, MedFactor	0.9370	0.8510	0.9997	0.9979	N/A	0.3566
RegularScore, AllImportant, SmallFactor	0.9775	0.2839	0.7290	0.9991	N/A	0.5237
RegularScore, InverseFrequency, LargeFactor	1.0000	0.9946	1.0000	1.0000	N/A	0.4345
RegularScore, InverseFrequency, MedFactor	0.9677	0.8865	0.9949	0.9742	N/A	0.2277
RegularScore, InverseFrequency, SmallFactor	0.9270	0.9103	0.7223	0.9952	N/A	0.8828

Table 4.3: Significance Testing: Regularizing to Github Corpus. Here we use the Mann Whitney method for hypothesis testing between the Baseline and each of the experimental settings, for each of the problems. The **p-value** is given for each problem. Full raw results can be found in the appendix, Table A.1.

Score Type	Node Weights	Factor	Total Rank
Baseline	Baseline	0.0	24
RegularScore	InverseFrequency	0.1	34
NormalizedScore	AllImportant	0.1	28
NormalizedScore	InverseFrequency	0.2	**21**
RegularScore	InverseFrequency	0.2	24
NormalizedScore	AllImportant	0.2	**18**
RegularScore	AllImportant	0.2	26
RegularScore	AllImportant	0.5	32
RegularScore	AllImportant	0.1	25
NormalizedScore	InverseFrequency	0.1	**23**
NormalizedScore	AllImportant	0.5	34
RegularScore	InverseFrequency	0.5	42
NormalizedScore	InverseFrequency	0.5	41

Table 4.4: Results Table: Regularizing for similarity to Github Corpus. Values shown are the sum of ranks of each method, across all problems. Full rankings per problem found in A.2.

4.3.2 Regularize for similarity to Gold Standard Solution

In the previous regime, we calculated the Humanlike score relative to an entire corpus of programs from Github. However, information can be lost inside of this average across thousands of programs, and we wanted to see what happens when we try to match the AST node distribution of a program that correctly solves the given problem. In this regime, instead of calculating the target distribution from the average across all programs in our corpus, we found a single solution to each of the problems in the benchmarks, calculated the distribution from that, and used this distribution in order to calculate the Humanlike score. In other words, we replaced our Github corpus with a single program that solves the task at hand. Similar to the previous experiment, this was done to answer **RQ2.2** and **RQ2.3**: To see if optimizing for this metric can improve tst case performance and readability of the generated programs.

From looking at the results, we found that it didn't seem to have much of an effect - while some runs showed an improvement over the baseline, it didn't seem to be consistent across multiple problems, and even if it was shown to be more promising on some problems, it wouldn't generalize to other problems. However, the VectorAverage problem also seemed to be improved with these methods. Figure 4.6 in the appendix

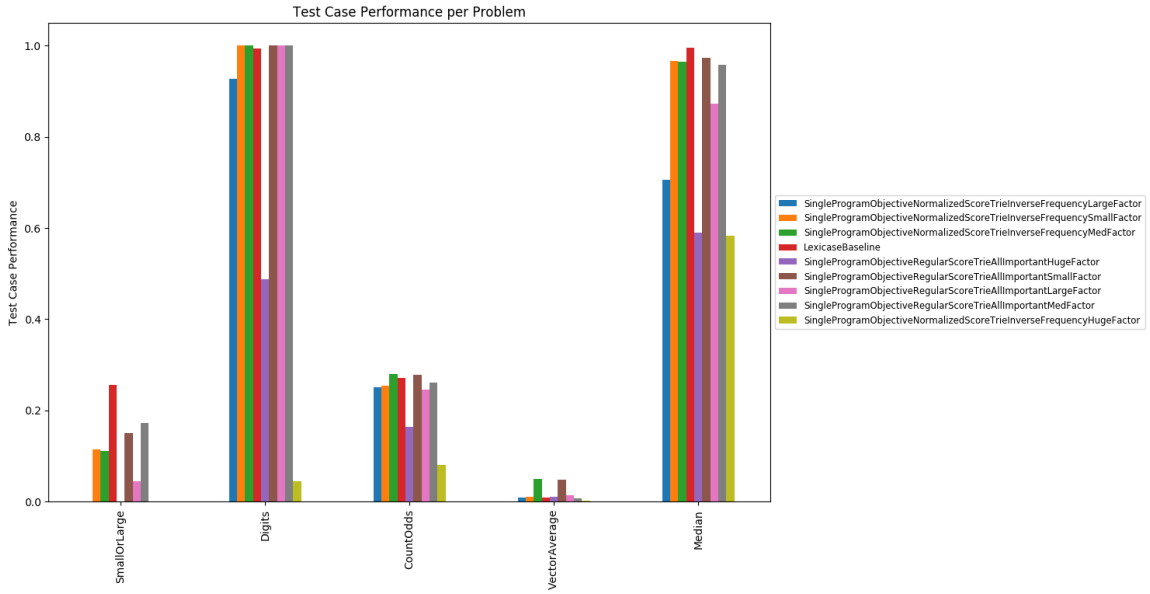

Figure 4-3: Results of Regularizing for similarity to Gold Standard Solution. Measured in average best percentage of test cases solved per run.

confirms that none of the results were significant.

It was also shown that solely optimizing for this metric wouldn't be good enough - when the *H-Facor = 1.0*, performance suffered dramatically. One experiment that would be interesting to run is to compare this with random search, and see if this is actually an improvement over that.

One thought for why this might be is that it turns out that test cases area actually one of the best signals you can get as to whether you've solved the problem or not - even if our measures are informative, in this test setup they would need to beat how informative test cases are in order to show positive results. This motivates our comparison of this scoring method with Random search, later in this section. Another experiment, which we perform in the next section, is to initialize the populations with a population that has been optimized for the Humanlike metrics.

4.3.3 Humanlike Initialization

Here we use the solutions generated from a GE run with an $H - Factor = 1.0$ as the initial population during initialization. We do this to see if first optimizing the

Score Type	Node Weights	Factor	Total Rank
NormalizedScore	InverseFrequency	0.2	**11**
RegularScore	AllImportant	0.2	19
NormalizedScore	InverseFrequency	0.1	17
RegularScore	AllImportant	0.5	22
RegularScore	AllImportant	0.1	**9**
Baseline	Baseline	0.0	**11**
NormalizedScore	InverseFrequency	0.5	30
RegularScore	AllImportant	1.0	27
NormalizedScore	InverseFrequency	1.0	34

Table 4.5: Results of Regularizing for similarity to Gold Standard Solution, Rankings. Full raw results found in A.3 and rankings per problem found in A.4.

Table 4.6: Significance Testing: Regularizing to Gold Standard program. Here we use the Mann Whitney method for hypothesis testing between the Baseline and each of the experimental settings, for each of the problems. The null hypothesis is that the scores are drawn from the same distribution. The **p-value** is given for each problem, along with whether the average of the experiment was above or below the baseline.

Experiment Name	Median	CountOdds	Digits	VectorAverage	SmallOrLarge
NormalizedScore, InverseFrequency, HugeFactor	1.0000	1.0000	1.0000	1.0000	0.9997
NormalizedScore, InverseFrequency, LargeFactor	0.9913	**0.0004**	1.0000	1.0000	0.6751
NormalizedScore, InverseFrequency, MedFactor	0.9471	**3.581e-05**	0.9201	0.9903	0.4105
NormalizedScore, InverseFrequency, SmallFactor	0.9896	**3.581e-05**	0.9518	0.9817	0.5569
RegularScore, AllImportant, HugeFactor	1.0000	0.9937	1.0000	1.0000	0.3071
RegularScore, AllImportant, LargeFactor	0.9987	**3.581e-05**	1.0000	0.9993	0.2314
RegularScore, AllImportant, MedFactor	0.9173	**3.581e-05**	0.9892	0.9423	0.8947
RegularScore, AllImportant, SmallFactor	0.7387	**3.581e-05**	0.9518	0.9583	0.3192

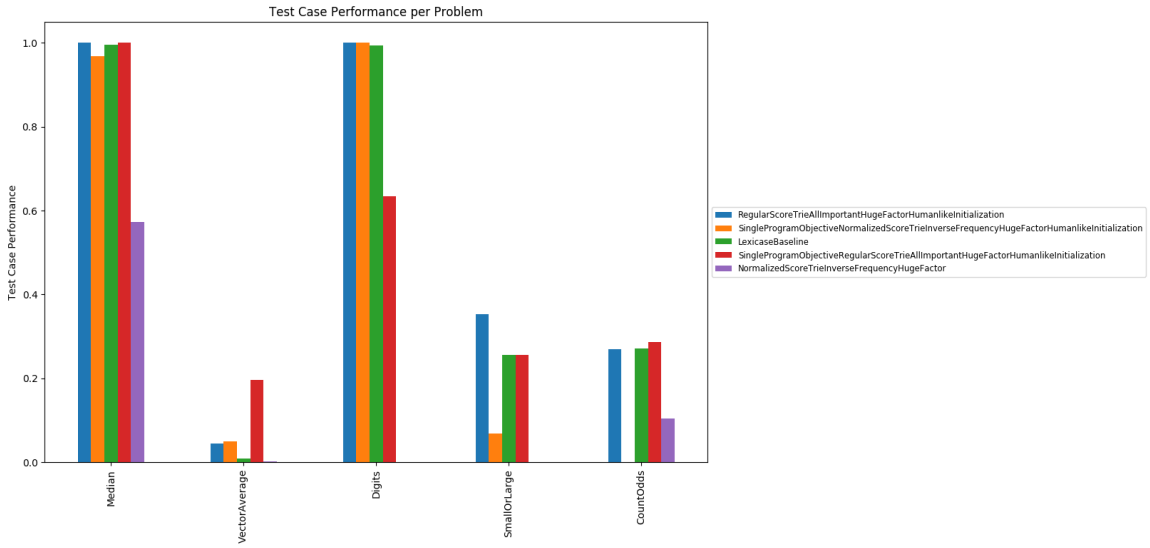

Figure 4-4: Results of Initializing with a population optimized only for the Humanlike score. Measured in average best percentage of test cases solved per run.

Score Type	Node Weights	Factor	Total Rank
RegularScore	AllImportant	1.0	**8**
NormalizedScore	InverseFrequency	1.0	16
Baseline	Baseline	0.0	**10**
RegularScore	AllImportant	1.0	**6**
NormalizedScore	InverseFrequency	1.0	18

Table 4.7: Results Table: Humanlike Initialization, Rankings. Rankings were computed for each individual problem, and then summed across all solutions.

population towards the humanlike factor alone provides a good starting point for future evolution, in order to help answer **RQ2.4**.

In the results, none of the settings seemed to have results consistently above the baseline. However, one setting seemed to have reasonable results, which used the Regular Score scheme, with all nodes equally important. In this case, the rankings seemed to be at the same level or better than the baseline of simply lexicase selection alone. Table 4.8 shows that the results aren't significantly better across all runs, but they are significantly better for the SmallOrLarge problem. With this in mind, we chose to run experiments with a reduced set of experiments and more runs, to get a more accurate significance measure.

Experiment Name	Digits	SmallOrLarge	CountOdds	VectorAverage	Median
NormalizedScore, InverseFrequency	1.0000	1.0000	1.0000	1.0000	0.9984
RegularScore, AllImportant	0.5872	**3.581e-05**	0.0809	0.2205	0.7188
GoldStandard, NormalizedScore, InverseFrequency	0.7799	**3.581e-05**	0.7290	0.9953	0.1326
GoldStandard, RegularScore, AllImportant	0.1891	0.5245	0.0809	0.4593	**0.0218**
TournamentBaseline	0.6294	0.5051	1.0000	0.9671	0.5235

Table 4.8: Significance Testing: Humanlike Initialization. Here we use the Mann Whitney method for hypothesis testing between the Baseline and each of the experimental settings, for each of the problems. The null hypothesis is that the scores are drawn from the same distribution. The **p-value** is given for each problem, along with whether the average of the experiment was above or below the baseline.

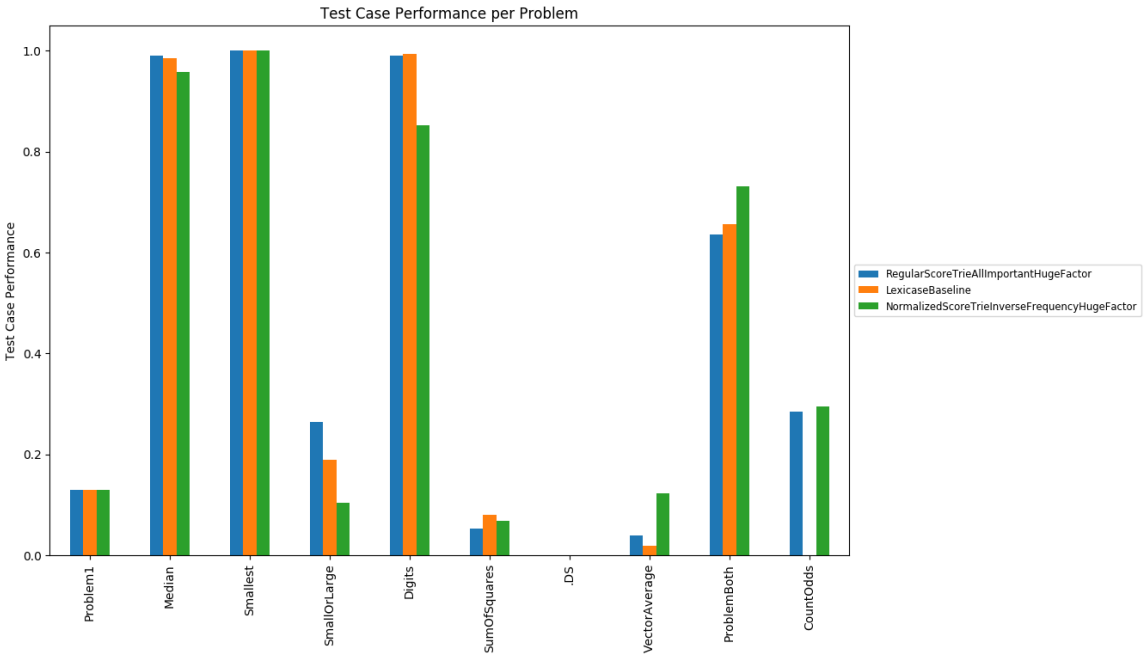

Figure 4-5: Second set of Results of Initializing with a population optimized only for the Humanlike score. Measured in average best percentage of test cases solved per run. Here 100 runs are used.

4.3.4 Humanlike Initialization, Constrained Set of Experiments

When looking at Figure 4-5, we can see that there were some runs that had improvements over the baseline, especially in the VectorAverage problem. And looking at the rankings alone in Table 4.9, we can see that this method ends up outperforming the baseline for the VectorAverage problem. While these regimes aren't significantly better across the board, Table 4.10 shows that the *Regular Score, All Important* regime is significantly better on the Smallest and Vector Average problems, which are both some of the harder problems for GE to solve.

Score Type	Node Weights	Factor	Total Rank
Baseline	Baseline	0.0	16
NormalizedScore	InverseFrequency	1.0	14
RegularScore	AllImportant	1.0	**13**

Table 4.9: Results Table: Humanlike Initialization Constrained, Rankings (100 runs)

Experiment Name	SmallOrLarge	Smallest	Digits	VectorAverage	CountOdds	SumOfSquares
NormalizedScore, InverseFrequency	1.0000	**0.0005**	0.9989	0.9981	0.3797	0.4537
RegularScore, AllImportant	1.0000	**8.3869e-12**	0.0612	**0.0064**	0.5225	0.7633

Table 4.10: Significance Testing: Humanlike Initialization, Constrained set of experiments (100 runs). Here we use the Mann Whitney method for hypothesis testing between the Baseline and each of the experimental settings, for each of the problems. The null hypothesis is that the scores are drawn from the same distribution. The **p-value** is given for each problem, along with whether the average of the experiment was above or below the baseline.

4.3.5 Humanlike Score vs. Random Search

In this experiment, GE was run with two different modes:

- Random Search: perform GE while ignoring test case fitness and Humanlike score entirely.

- Humanlike Optimization: Perform GE while ignoring test case fitness, and only using Humanlike score to evaluate the fitness of programs.

This was done to answer **RQ2.1, RQ2.2**, and **RQ2.3**: to see if this score improves program performance and readability, along with seeing if directly optimizing for this score changes the distribution of Humanlike score in the population.

Figure 4-6 shows the performance of Random search as compared to Humanlike optimization, and Table 4.11 shows the tests for significant differences between Random Search and optimizing for Humanlike Score only.

As can be seen from the chart, random search appears to perform much better than just optimizing for Humanlike score, indicating that optimizing for our Humanlike score could actually hinder performance. This is confirmed by the significance test in Table 4.11 - In every problem and setting, the null hypothesis that the experiment median is lower than the random search median is accepted with high probability.

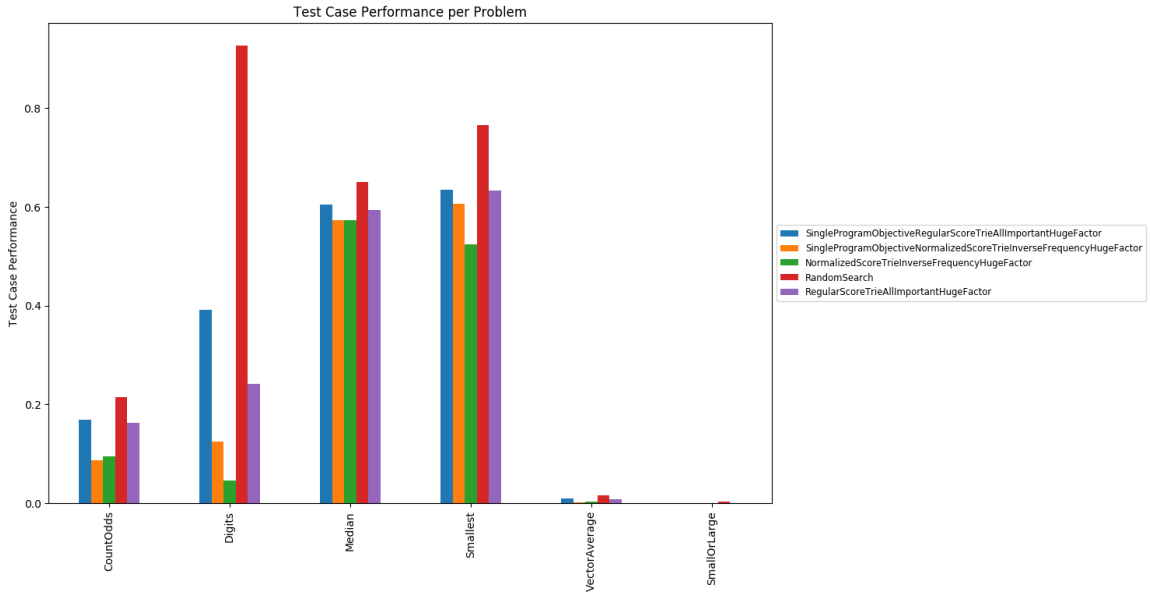

Figure 4-6: Results of comparing our Humanlike score to Random Search. Measured in average best percentage of test cases solved across all runs.

Experiment Name	SmallOrLarge	Digits	Smallest	CountOdds	Median	VectorAverage
NormalizedScore, InverseFrequency	1.0000	1.0000	1.0000	0.8516	1.0000	1.0000
RegularScore, AllImportant	1.0000	1.0000	0.9982	0.8516	1.0000	0.9762
GoldStandard, NormalizedScore, InverseFrequency	1.0000	1.0000	1.0000	0.8516	1.0000	1.0000
GoldStandard, RegularScore, AllImportant	1.0000	0.9999	0.9608	0.8516	0.9999	0.9380

Table 4.11: Significance Testing: Humanlike Score vs. Random Search. Here we use the Mann Whitney method for hypothesis testing between the Baseline and each of the experimental settings, for each of the problems. The null hypothesis is that the scores are drawn from the same distribution. The **p-value** is given for each problem, along with whether the average of the experiment was above or below the baseline.

However, what do the solutions look like? Below are three sample solutions: the first is drawn from the Random Search run, and the next two are drawn from the Humanlike Optimization runs with two different settings.

- **Random Search**:

```
b0 = bool(); b1 = bool(); b2 = bool()
i0 = int(); i1 = int(); i2 = int()
res0 = int()
while b0:
  while divInt(abs(( ( mod(in2,i1) * mod(( int(0.0) * int(685.0) ),i2) ) *
  divInt(( int(758.0) - in1 ),int(3.0)) )),divInt(abs(in0),int(4.0))) >= int(906.0):
    b1 = False
    if loopBreak > loopBreakConst or stop:
      break
    loopBreak += 1
  b1 = max(min(min(int(2.0), int(8232.0)), max(in2, min(in0, divInt(in2,abs(int(4.0))))))),
  int(2.0)) != mod(res0,min(i1, in1))
  b2 = mod(( in0 - int(42.0) ),mod(i1,i2)) >= int(330.0)
  if loopBreak > loopBreakConst or stop:
    break
  loopBreak += 1
```

  While still simple, this program features two nested while loops, one of which is completely useless.

- **Humanlike Optimization, Normalized Score, Inverse Frequency Node Importance**:

```
b0 = bool(); b1 = bool(); b2 = bool()
i0 = int(); i1 = int(); i2 = int()
res0 = int()
if ( not not ( ( not not not not not not not not not not not not not not not not
 not not not not not not not not not not not not not not not not True and not not not not
 not not not not not not not not not not not not abs(int(6.0)) >= i2 ) or ( True or not
 not not not not not not not not False ) ) and not not not not not not not not
 not not not not not not not not not not not not False ):
  b2 = not not not not not not not not not not not not not not not not not not not not
  not not not b0
```

  This code features many repeated 'not' statements - this is likely a degenerate behavior of the Normalized Score.

- **Humanlike Optimization, Regular Score, Equal Node Importance**:

```
b0 = bool(); b1 = bool(); b2 = bool()
i0 = int(); i1 = int(); i2 = int()
res0 = int()
b0 = not ( b0 or abs(( ( abs(max(max(mod(int(8.0),abs(max(in2, in0))), i0),
int(28.0))) + abs(( int(5.0) * ( divInt(in2,min(min(i1, min(int(9.0),
max(i2, in2))), in2)) - ( in0 + in0 ) ) )) ) + int(4.0) )) != i1 )
```

While mildly complex, this doesn't feature multiple nested loops.

The first thing that sticks out is the normalized score - it provides many repeated "not" statement, which is undesirable. While the normalized score was motivated by not making our score just be a regularization on program length, it also creates degenerate solutions which cause very common nodes to appear, as this gives a bonus to the overall score. This goes to show the fragility of these metrics, and the care in which they must be designed.

While the solution generated by the random solution and the second Humanlike Optimization experiment are relatively similar in complexity, the Random Search solution contains two nested while loops (one of which is completely useless, simply setting a variable to False every iteration), the program generated by the Humanlike Optimization was simpler, and didn't have the problem of generating nested loops. In addition to the above, it shows that the context sensitivity potentially needs to be tuned, as in this case, loops were avoided.

4.3.6 Changes in Humanlike Score throughout the experiments

In this section, we look at the ways in which the Humanlike Score changed throughout the course of the experiments. We look only at the Humanlike score that uses the settings with Regular Score, Trie Data Structure, and All Nodes equally important.

Figure 4-7 shows the Humanlike scores computed across the programs at the beginning and end of a GE run in which only the Humanlike score was optimized for, along with the scores of the programs from the Github dataset. One thing that can be seen here is the fact that it shifts significantly in the positive direction over time - however, it does this to the point where it overshoots the average across all programs in the Github corpus. Another thing that can be seen is that many

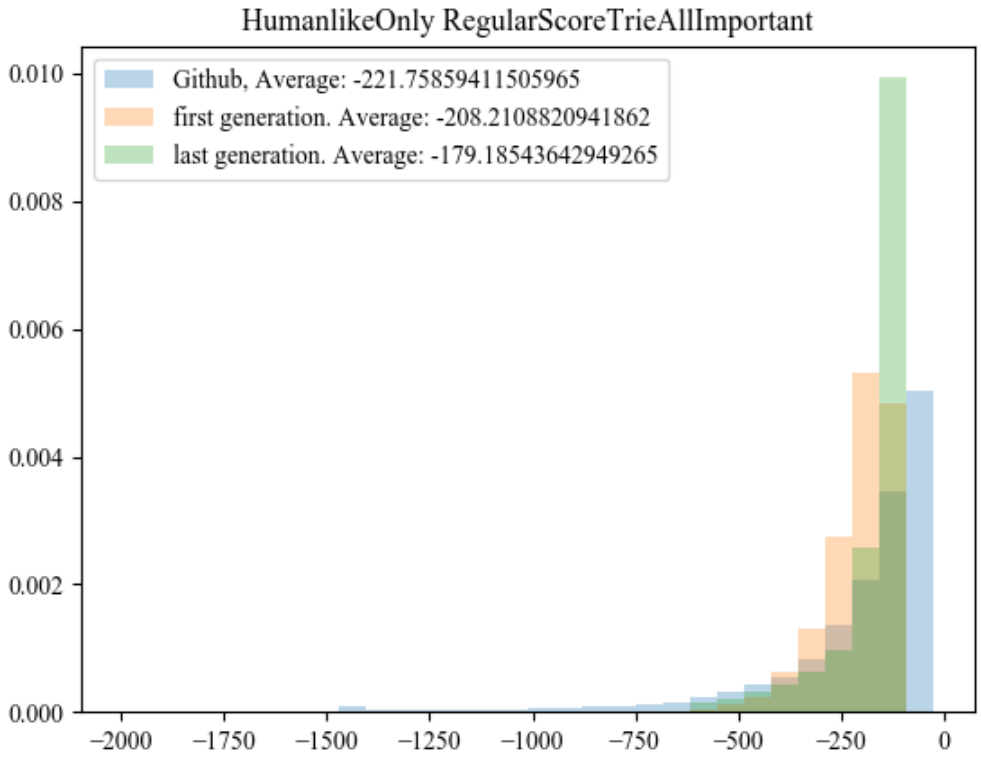

Figure 4-7: Distribution of Humanlike Scores across the initial and final populations of the run in which only the Humanlike score was optimized for. Scores for the github corpus shown as well.

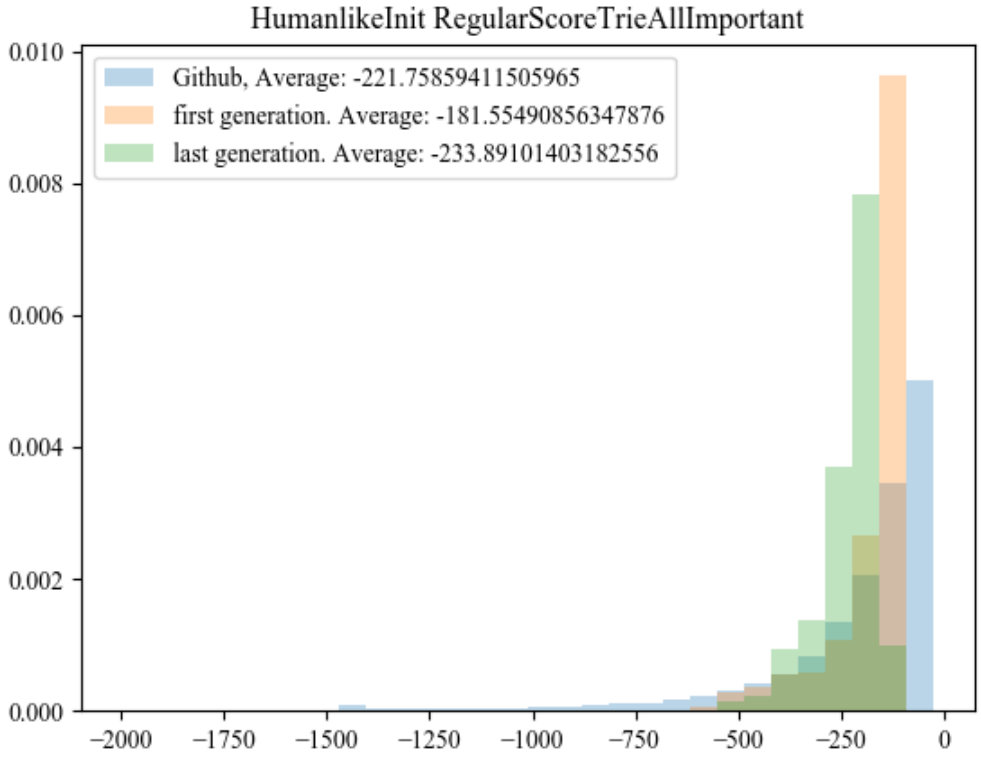

Figure 4-8: Distribution of Humanlike Scores across the initial and final populations of the Humanlike Initialization run. Scores for the github corpus shown as well.

programs are within the same increasingly narrow band of values as the Humanlike score is optimized for - this has the potential to reduce diversity. One thing we can learn from this is that when you optimize for this score, it might constrain the values to a more narrow range.

While Figure 4-7 showed the results of only optimizing for the Humanlike score, Figure 4-8 shows the results of initializing from a population that has been optimized for the Humanlike score only, and then running standard GE on the given population. What can be seen here is that the average score value drops back to a much lower value than before, which is the same as what it started at in the previous point. This is interesting - it shows that the population is naturally resisting a shift towards higher scores.

Chapter 5

Conclusions and Future Work

5.1 Transferability Conclusions

In Chapter 3, attempts at transferring information between populations solving separate but related problems were explored, utilizing the first two problems of the first 6.00.1x problem set. Specifically, we explored the effects of 1.) giving a population intermediate goals to solve before placing a complex task in front of it, 2.) pre-optimizing a population to solve one task before optimizing it to solve another task, and 3.) initializing populations with solutions to related problems as opposed to random individuals. The main conclusions are listed below:

1. When solving a complex task with Genetic Programming, using a more difficult to attain intermediate goal as opposed a simpler one was found to be beneficial in terms of allowing the population to solve a complex problem with multiple parts. Additionally, more time spent optimizing for the intermediate goal was shown to be beneficial.

2. While pre-optimizing a population to one task before solving another related task can be harmful or neutral, it also has the potential to be beneficial under the right circumstances. Both the problem to pre-optimize for and the amount of pre-optimization done factor into how it will affect performance of the population.

3. The structure of the problem and the grammar are important for knowledge transfer - if the grammar isn't set up to transfer the structure of one problem to another, then initializing from solutions to one problem might not necessarily help it solve a related problem.

5.2 Humanlike Optimization Conclusions

In Chapter 4, we defined a context-sensitive metric for scoring programs relative to their similarity to a given target population. We used this score as an optimization objective along with fitness, looked at the qualitative effects of optimizing for this objective, and investigated using this objective in the initialization step. The main conclusions from these experiments are enumerated below:

1. Optimizing for the Humanlike score has the potential to create degenerate solutions if not designed carefully, and thus need to be carefully designed to produce reasonable programs.

2. Optimizing for a Humanlike score leads the results to be in a relatively constrained range of Humanlike scores - broadening this could potentially be a way to add more diversity to the population.

3. While the Humanlike scores don't necessarily help test case performance, qualitative analysis shows that they may be helpful in making programs more humanlike. Whether or not this just occurs because the score applies parsimony pressure or not still needs to be determined.

4. These scores are something that can be optimized for, and by optimizing for them, you can change their average value within a population. However, when performing a normal GE run, they usually end up drifting back to lower values - even if you try to maximize this score across a population, the natural tendency during GE is to drift to lower values.

5. Optimizing for the Humanlike Score alone was shown to be significantly less effective than Random Search at solving the test cases.

6. Initializing from a population of programs that had been pre-optimized for the Humanlike score only didn't have a significant effect in many problems, but was statistically significant in the Smallest and VectorAverage problems, which are some of the hardest problems we looked at.

5.3 Future Work

While much work was put into this thesis, there are still many directions we could potentially take this. A few are listed below:

1. While the grammars used in these experiment were each specific to a given problem or pair of problems, there are grammars that can express solutions to many problems. Such grammars would likely allow for much easier transfer of structure between problems, so it would be interesting to repeat the task switching experiment with a larger number of tasks with such a grammar.

2. One big problem the concontext-sensitive score was supposed to solve was the fact that nested loops happen a lot in GE and can be harmful. However, the grammars we currently use are very restricted, making this less of an issue. It would be much more interesting to test these measures out on a much less restricted grammar, as this would be more likely to show the value of the Humanlike Score.

3. Some of the Humanlike Score variants performed better than others, which shows it can have an effect on the search. One future direction is to further explore different forms for this Humanlike Score, and try to tune it until it can speed up GE search.

4. While the effects were unclear of initializing a population with solutions to a related problem, this is likely due to the fact that the grammar and operators

weren't optimized for transferring the information between problems. One future direction of work would be to try to design grammars that are effective at transferring structure between problems.

Appendix A

Tables

A.0.1 Experiments on Regularizing to Github Corpus

A.0.2 Experiments on Regularizing to Gold Standard Solution

A.0.3 Humanlike Initialization

A.0.4 Humanlike Initialization, Constrained

A.0.5 Humanlike Score vs. Random Search

Table A.1: Results Table: Regularize for similarity to Github Corpus, raw scores. Values shown are measured in terms of the average percentage of test cases solved by the best individual per run.

Problem	Score Type	Node Weights	Factor	best_percent_test_cases_solved_mean
CountOdds	N/A	N/A	N/A	0.271
CountOdds	NormalizedScore	AllImportant	0.1	0.250
CountOdds	NormalizedScore	AllImportant	0.2	0.280
CountOdds	NormalizedScore	AllImportant	0.5	0.221
CountOdds	NormalizedScore	InverseFrequency	0.1	0.247
CountOdds	NormalizedScore	InverseFrequency	0.2	0.245
CountOdds	NormalizedScore	InverseFrequency	0.5	0.206
CountOdds	RegularScore	AllImportant	0.1	0.256
CountOdds	RegularScore	AllImportant	0.2	0.262
CountOdds	RegularScore	AllImportant	0.5	0.206
CountOdds	RegularScore	InverseFrequency	0.1	0.254
CountOdds	RegularScore	InverseFrequency	0.2	0.253
CountOdds	RegularScore	InverseFrequency	0.5	0.217
Digits	N/A	N/A	N/A	0.994
Digits	NormalizedScore	AllImportant	0.1	0.995
Digits	NormalizedScore	AllImportant	0.2	0.994
Digits	NormalizedScore	AllImportant	0.5	0.992
Digits	NormalizedScore	InverseFrequency	0.1	0.992
Digits	NormalizedScore	InverseFrequency	0.2	0.994
Digits	NormalizedScore	InverseFrequency	0.5	0.994
Digits	RegularScore	AllImportant	0.2	0.994
Digits	RegularScore	AllImportant	0.5	0.994
Digits	RegularScore	InverseFrequency	0.1	0.956
Digits	RegularScore	InverseFrequency	0.2	0.993
Digits	RegularScore	InverseFrequency	0.5	0.991
Median	N/A	N/A	N/A	0.995
Median	NormalizedScore	AllImportant	0.1	0.977
Median	NormalizedScore	AllImportant	0.2	0.911
Median	NormalizedScore	InverseFrequency	0.2	0.969
Median	RegularScore	AllImportant	0.2	0.892
Median	RegularScore	AllImportant	0.5	0.783
Median	RegularScore	InverseFrequency	0.1	0.984
Median	RegularScore	InverseFrequency	0.2	0.935
SmallOrLarge	N/A	N/A	N/A	0.256
SmallOrLarge	NormalizedScore	AllImportant	0.1	0.041
SmallOrLarge	NormalizedScore	AllImportant	0.2	0.069
SmallOrLarge	NormalizedScore	AllImportant	0.5	0.036
SmallOrLarge	NormalizedScore	InverseFrequency	0.1	0.127
SmallOrLarge	NormalizedScore	InverseFrequency	0.2	0.161
SmallOrLarge	NormalizedScore	InverseFrequency	0.5	0.000
SmallOrLarge	RegularScore	AllImportant	0.1	0.039
SmallOrLarge	RegularScore	AllImportant	0.2	0.062
SmallOrLarge	RegularScore	AllImportant	0.5	0.021
SmallOrLarge	RegularScore	InverseFrequency	0.1	0.088
SmallOrLarge	RegularScore	InverseFrequency	0.2	0.124
SmallOrLarge	RegularScore	InverseFrequency	0.5	0.000
Smallest	NormalizedScore	AllImportant	0.1	1.000
Smallest	NormalizedScore	AllImportant	0.2	1.000
Smallest	NormalizedScore	AllImportant	0.5	1.000
Smallest	NormalizedScore	InverseFrequency	0.1	1.000
Smallest	NormalizedScore	InverseFrequency	0.2	1.000
Smallest	NormalizedScore	InverseFrequency	0.5	1.000
Smallest	RegularScore	AllImportant	0.1	1.000
Smallest	RegularScore	AllImportant	0.2	1.000
Smallest	RegularScore	AllImportant	0.5	1.000
Smallest	RegularScore	InverseFrequency	0.1	1.000
Smallest	RegularScore	InverseFrequency	0.2	1.000
Smallest	RegularScore	InverseFrequency	0.5	1.000
VectorAverage	N/A	N/A	N/A	0.009
VectorAverage	NormalizedScore	AllImportant	0.1	0.007
VectorAverage	NormalizedScore	AllImportant	0.2	0.011
VectorAverage	NormalizedScore	AllImportant	0.5	0.011
VectorAverage	NormalizedScore	InverseFrequency	0.1	0.055
VectorAverage	NormalizedScore	InverseFrequency	0.2	0.049
VectorAverage	RegularScore	AllImportant	0.1	0.061
VectorAverage	RegularScore	AllImportant	0.2	0.009
VectorAverage	RegularScore	AllImportant	0.5	0.014
VectorAverage	RegularScore	InverseFrequency	0.1	0.005
VectorAverage	RegularScore	InverseFrequency	0.2	0.011
VectorAverage	RegularScore	InverseFrequency	0.5	0.010

Table A.2: Results Table: Regularizing for similarity to Github Corpus, Rankings per problem.

Score Type	Node Weights	Factor	Total Rank	Median	CountOdds	Digits	VectorAverage	SmallOrLarge	Smallest
Baseline	Baseline	0.0	24	1	2	2	9	1	10
RegularScore	InverseFrequency	0.1	34	2	5	11	12	5	1
NormalizedScore	AllImportant	0.1	28	3	7	1	11	8	1
NormalizedScore	InverseFrequency	0.2	21	4	9	6	3	2	1
RegularScore	InverseFrequency	0.2	24	5	6	7	6	4	1
NormalizedScore	AllImportant	0.2	18	6	1	3	7	6	1
RegularScore	AllImportant	0.2	26	7	3	5	10	7	1
RegularScore	AllImportant	0.5	32	8	12	4	4	11	1
RegularScore	AllImportant	0.1	25	10	4	10	1	9	1
NormalizedScore	InverseFrequency	0.1	23	10	8	9	2	3	1
NormalizedScore	AllImportant	0.5	34	10	10	8	5	10	1
RegularScore	InverseFrequency	0.5	42	10	11	10	8	12	1
NormalizedScore	InverseFrequency	0.5	41	10	13	5	10	12	1

Table A.3: Results Table: Regularizing for similarity to Gold Standard Solutions, Raw Scores

Problem	Score Type	Node Weights	Factor	best_percent_test_cases_solved_mean
CountOdds	N/A	N/A	N/A	0.271
CountOdds	NormalizedScore	InverseFrequency	0.1	0.255
CountOdds	NormalizedScore	InverseFrequency	0.2	0.279
CountOdds	NormalizedScore	InverseFrequency	0.5	0.250
CountOdds	NormalizedScore	InverseFrequency	1.0	0.081
CountOdds	RegularScore	AllImportant	0.1	0.277
CountOdds	RegularScore	AllImportant	0.2	0.260
CountOdds	RegularScore	AllImportant	0.5	0.245
CountOdds	RegularScore	AllImportant	1.0	0.163
Digits	N/A	N/A	N/A	0.994
Digits	NormalizedScore	InverseFrequency	0.1	1.000
Digits	NormalizedScore	InverseFrequency	0.2	1.000
Digits	NormalizedScore	InverseFrequency	0.5	0.927
Digits	NormalizedScore	InverseFrequency	1.0	0.045
Digits	RegularScore	AllImportant	0.1	1.000
Digits	RegularScore	AllImportant	0.2	1.000
Digits	RegularScore	AllImportant	0.5	1.000
Digits	RegularScore	AllImportant	1.0	0.487
Median	N/A	N/A	N/A	0.995
Median	NormalizedScore	InverseFrequency	0.1	0.966
Median	NormalizedScore	InverseFrequency	0.2	0.965
Median	NormalizedScore	InverseFrequency	0.5	0.706
Median	NormalizedScore	InverseFrequency	1.0	0.582
Median	RegularScore	AllImportant	0.1	0.973
Median	RegularScore	AllImportant	0.2	0.959
Median	RegularScore	AllImportant	0.5	0.873
Median	RegularScore	AllImportant	1.0	0.590
SmallOrLarge	N/A	N/A	N/A	0.256
SmallOrLarge	NormalizedScore	InverseFrequency	0.1	0.114
SmallOrLarge	NormalizedScore	InverseFrequency	0.2	0.111
SmallOrLarge	NormalizedScore	InverseFrequency	1.0	0.000
SmallOrLarge	RegularScore	AllImportant	0.1	0.150
SmallOrLarge	RegularScore	AllImportant	0.2	0.172
SmallOrLarge	RegularScore	AllImportant	0.5	0.044
SmallOrLarge	RegularScore	AllImportant	1.0	0.000
VectorAverage	N/A	N/A	N/A	0.009
VectorAverage	NormalizedScore	InverseFrequency	0.1	0.010
VectorAverage	NormalizedScore	InverseFrequency	0.2	0.049
VectorAverage	NormalizedScore	InverseFrequency	0.5	0.008
VectorAverage	NormalizedScore	InverseFrequency	1.0	0.001
VectorAverage	RegularScore	AllImportant	0.1	0.047
VectorAverage	RegularScore	AllImportant	0.2	0.007
VectorAverage	RegularScore	AllImportant	0.5	0.013
VectorAverage	RegularScore	AllImportant	1.0	0.011

Table A.4: Results Table: Single Program Objective, Rankings

Score Type	Node Weights	Factor	Total Rank	Digits	CountOdds	Median	SmallOrLarge	VectorAverage
NormalizedScore	InverseFrequency	0.2	11	1	1	4	5	1
RegularScore	AllImportant	0.2	19	1	4	5	2	8
NormalizedScore	InverseFrequency	0.1	17	1	5	3	4	5
RegularScore	AllImportant	0.5	22	1	7	6	6	3
RegularScore	AllImportant	0.1	9	1	2	2	3	2
Baseline	Baseline	0.0	11	2	3	1	1	6
NormalizedScore	InverseFrequency	0.5	30	3	6	7	10	7
RegularScore	AllImportant	1.0	27	4	8	8	7	4
NormalizedScore	InverseFrequency	1.0	34	5	9	9	7	9

Table A.5: Results Table: Humanlike Initialization, Raw Scores (23 Runs)

Problem	Score Type	Node Weights	Factor	best_percent_test_cases_solved_mean
CountOdds	N/A	N/A	N/A	0.271
CountOdds	NormalizedScore	InverseFrequency	1.0	0.104
CountOdds	RegularScore	AllImportant	1.0	0.269
CountOdds	RegularScore	AllImportant	1.0	0.287
Digits	N/A	N/A	N/A	0.994
Digits	NormalizedScore	InverseFrequency	1.0	1.000
Digits	RegularScore	AllImportant	1.0	0.633
Digits	RegularScore	AllImportant	1.0	1.000
Median	N/A	N/A	N/A	0.995
Median	NormalizedScore	InverseFrequency	1.0	0.572
Median	NormalizedScore	InverseFrequency	1.0	0.968
Median	RegularScore	AllImportant	1.0	1.000
Median	RegularScore	AllImportant	1.0	1.000
SmallOrLarge	N/A	N/A	N/A	0.256
SmallOrLarge	NormalizedScore	InverseFrequency	1.0	0.000
SmallOrLarge	NormalizedScore	InverseFrequency	1.0	0.068
SmallOrLarge	RegularScore	AllImportant	1.0	0.256
SmallOrLarge	RegularScore	AllImportant	1.0	0.352
VectorAverage	N/A	N/A	N/A	0.009
VectorAverage	NormalizedScore	InverseFrequency	1.0	0.002
VectorAverage	NormalizedScore	InverseFrequency	1.0	0.049
VectorAverage	RegularScore	AllImportant	1.0	0.044
VectorAverage	RegularScore	AllImportant	1.0	0.195

Table A.6: Results Table: Humanlike Initialization, Rankings

Score Type	Node Weights	Factor	Total Rank	Digits	SmallOrLarge	VectorAverage	CountOdds	Median
RegularScore	AllImportant	1.0	8	1	1	3	3	1
NormalizedScore	InverseFrequency	1.0	16	1	4	2	7	3
Baseline	Baseline	0.0	10	2	2	4	2	2
RegularScore	AllImportant	1.0	6	3	3	1	1	1
NormalizedScore	InverseFrequency	1.0	18	7	5	5	4	4

Table A.7: Results Table: Humanlike Initialization Constrained, Raw Scores (100 Runs)

Problem	Score Type	Node Weights	Factor	best_percent_test_cases_solved_mean
CountOdds	NormalizedScore	InverseFrequency	1.0	0.295
CountOdds	RegularScore	AllImportant	1.0	0.284
Digits	N/A	N/A	N/A	0.994
Digits	NormalizedScore	InverseFrequency	1.0	0.852
Digits	RegularScore	AllImportant	1.0	0.989
Median	N/A	N/A	N/A	0.985
Median	NormalizedScore	InverseFrequency	1.0	0.958
Median	RegularScore	AllImportant	1.0	0.990
Problem1	N/A	N/A	N/A	0.130
Problem1	NormalizedScore	InverseFrequency	1.0	0.130
Problem1	RegularScore	AllImportant	1.0	0.130
ProblemBoth	N/A	N/A	N/A	0.656
ProblemBoth	NormalizedScore	InverseFrequency	1.0	0.731
ProblemBoth	RegularScore	AllImportant	1.0	0.636
SmallOrLarge	N/A	N/A	N/A	0.190
SmallOrLarge	NormalizedScore	InverseFrequency	1.0	0.104
SmallOrLarge	RegularScore	AllImportant	1.0	0.264
Smallest	N/A	N/A	N/A	1.000
Smallest	NormalizedScore	InverseFrequency	1.0	1.000
Smallest	RegularScore	AllImportant	1.0	1.000
SumOfSquares	N/A	N/A	N/A	0.080
SumOfSquares	NormalizedScore	InverseFrequency	1.0	0.068
SumOfSquares	RegularScore	AllImportant	1.0	0.053
VectorAverage	N/A	N/A	N/A	0.019
VectorAverage	NormalizedScore	InverseFrequency	1.0	0.123
VectorAverage	RegularScore	AllImportant	1.0	0.039

Table A.8: Results Table: Humanlike Initialization Constrained, Rankings (100 runs)

Score Type	Node Weights	Factor	Total Rank	SumOfSquares	VectorAverage	ProblemBoth	Median	CountOdds	Problem1	SmallOrLarge	Smallest	Digits
Baseline	Baseline	0.0	16	1	3	2	2	4	1	2	1	1
NormalizedScore	InverseFrequency	1.0	14	2	1	1	3	1	1	3	1	3
RegularScore	AllImportant	1.0	13	3	2	3	1	2	1	1	1	2

Table A.9: Results Table: Humanlike Score vs. Random Search, Raw Scores

Problem	Score Type	Node Weights	Factor	best_percent_test_cases_solved_mean
CountOdds	N/A	N/A	N/A	0.215
CountOdds	NormalizedScore	InverseFrequency	1.0	0.087
CountOdds	NormalizedScore	InverseFrequency	1.0	0.094
CountOdds	RegularScore	AllImportant	1.0	0.162
CountOdds	RegularScore	AllImportant	1.0	0.168
Digits	N/A	N/A	N/A	0.926
Digits	NormalizedScore	InverseFrequency	1.0	0.046
Digits	NormalizedScore	InverseFrequency	1.0	0.124
Digits	RegularScore	AllImportant	1.0	0.242
Digits	RegularScore	AllImportant	1.0	0.392
Median	N/A	N/A	N/A	0.650
Median	NormalizedScore	InverseFrequency	1.0	0.573
Median	NormalizedScore	InverseFrequency	1.0	0.574
Median	RegularScore	AllImportant	1.0	0.594
Median	RegularScore	AllImportant	1.0	0.604
SmallOrLarge	N/A	N/A	N/A	0.003
SmallOrLarge	NormalizedScore	InverseFrequency	1.0	0.000
SmallOrLarge	NormalizedScore	InverseFrequency	1.0	0.000
SmallOrLarge	RegularScore	AllImportant	1.0	0.000
SmallOrLarge	RegularScore	AllImportant	1.0	0.000
Smallest	N/A	N/A	N/A	0.765
Smallest	NormalizedScore	InverseFrequency	1.0	0.525
Smallest	NormalizedScore	InverseFrequency	1.0	0.606
Smallest	RegularScore	AllImportant	1.0	0.634
Smallest	RegularScore	AllImportant	1.0	0.634
VectorAverage	N/A	N/A	N/A	0.016
VectorAverage	NormalizedScore	InverseFrequency	1.0	0.001
VectorAverage	NormalizedScore	InverseFrequency	1.0	0.003
VectorAverage	RegularScore	AllImportant	1.0	0.008
VectorAverage	RegularScore	AllImportant	1.0	0.010

Table A.10: Results Table: Humanlike Score vs. Random Search, Rankings.

Score Type	Node Weights	Factor	Total Rank	SmallOrLarge	Digits	VectorAverage	Median	Smallest	CountOdds
Baseline	Baseline	0.0	5	1	1	1	1	1	1
RegularScore	AllImportant	1.0	10	2	2	2	2	2	2
RegularScore	AllImportant	1.0	15	2	3	3	3	3	3
NormalizedScore	InverseFrequency	1.0	23	2	5	4	5	5	4
NormalizedScore	InverseFrequency	1.0	22	2	4	5	4	4	5

Bibliography

[1] Lark parser.

[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] Ayesha Bajwa, Ana Bell, Erik Hemberg, and Una-May OâĂŹReilly. Analyzing student code trajectories in an introductory programming mooc. In *2019 IEEE Learning With MOOCS (LWMOOCS)*, pages 53–58. IEEE, 2019.

[4] Iwo Bladek and Krzysztof Krawiec. Simultaneous synthesis of multiple functions using genetic programming with scaffolding. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 97–98, 2016.

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

[6] Ian Dempsey, Michael O'Neill, and Anthony Brabazon. *Foundations in grammatical evolution for dynamic environments*, volume 194. Springer, 2009.

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

[8] George Gerules and Cezary Janikow. A survey of modularity in genetic programming. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 5034–5043. IEEE, 2016.

[9] D.R. Whitney H.B. Mann. On a test of whether one of two random variables is stochastically larger than the other. volume 18, pages 50–60. 1947.

[10] 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.

[11] Thomas Helmuth and Lee Spector. Detailed problem descriptions for general program synthesis benchmark suite. *School of Computer Science, University of Massachusetts Amherst, Tech. Rep.*, 2015.

[12] 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.

[13] 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*, pages 1039–1046, 2019.

[14] Wojciech Jaśkowski, Krzysztof Krawiec, and Bartosz Wieloch. Multitask visual learning using genetic programming. *Evolutionary computation*, 16(4):439–459, 2008.

[15] 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.

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

[17] Krzysztof Krawiec and Bartosz Wieloch. Functional modularity for genetic programming. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 995–1002, 2009.

[18] Krzysztof Krawiec and Bartosz Wieloch. Automatic generation and exploitation of related problems in genetic programming. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[19] 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.

[20] Miguel Nicolau. Understanding grammatical evolution: initialisation. *Genetic Programming and Evolvable Machines*, 18(4):467–507, 2017.

[21] 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.

[22] 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.

[23] Michael OâĂŹNeill and Lee Spector. Automatic programming: The open issue? *Genetic Programming and Evolvable Machines*, pages 1–12, 2019.

[24] Anuradha Purohit, Narendra S. Choudhari, and Aruna Tiwari. Code bloat problem in genetic programming. volume 3, pages 1–21. April 2013.

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

[26] Conor Ryan, Maarten Keijzer, and Mike Cattolico. Favourable biasing of function sets using run transferable libraries. In *Genetic Programming Theory and Practice II*, pages 103–120. Springer, 2005.

[27] Ruchira Sasanka and Konstantinos Krommydas. An evolutionary framework for automatic and guided discovery of algorithms. *arXiv preprint arXiv:1904.02830*, 2019.

[28] Eric O Scott and Kenneth A De Jong. Automating knowledge transfer with multi-task optimization. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 2252–2259. IEEE, 2019.

[29] Dominik Sobania and Franz Rothlauf. Teaching gp to program like a human software developer: using perplexity pressure to guide program synthesis approaches. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2019)*, pages 1065–1074. July 2019.

[30] Jacob Soderlund, Darwin Vickers, and Alan Blair. Parallel hierarchical evolution of string library functions. In *International Conference on Parallel Problem Solving from Nature*, pages 281–291. Springer, 2016.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] Xiaolong Zheng, AK Qin, Maoguo Gong, and Deyun Zhou. Self-regulated evolutionary multi-task optimization. *IEEE Transactions on Evolutionary Computation*, 2019.