Getting a Head Start on Program Synthesis with Genetic Programming

Jordan Wick[1], Erik Hemberg[1], and Una-May O'Reilly[1]

[1] MIT, USA
[2] wickj@alumn.mit.edu, hembergerik@csail.mit.edu, unamay@csail.mit.edu

Abstract. We explore how to give Genetic Programming (GP) a head start to synthesize a programming problem. Our method uses a related problem and introduces a schedule that directs GP to solve the related problem first either fully or to some extent first, or at the same time. In addition, if the related problem's solutions are written by students or evolved by GP, we explore the extent to which initializing the GP population with some of these solutions provides a head start. We find that having a population solve one programming problem before working to solve a related programming problem helps to a greater extent as the targeted problems and the intermediate problems themselves are selected to be more challenging.

Keywords: Genetic Programming, grammar, program synthesis, multi task

1 Introduction

It is possible to learn programming on one's own but following in a course with a teacher can make the learning easier. That is, teachers often give their students a head start on new problems and, prior to a problem set, they introduce examples of problems related to it. They may also provide a solution that a student only needs to extend or re-factor, with modest effort, to solve the problem, all of which provides support for learning. This notion of moving from one problem to a related one is conceptually similar to what, in Machine Learning(ML), is called multi-task learning [30]. A teacher often introduces small, modular solutions that can be easily combined in different ways with others. This head start allows them to show how to compose them in different combinations to solve larger problems(tasks) that share some subsolutions in common. This is similar to *curriculum learning* [4] in ML which starts with a small task and then introduces of a multi-stage curriculum.

We are interested in giving *Genetic Programming Used for Program Synthesis* (GP) a head start. Our desire is to improve the capacity of GP to apply existing problem solving knowledge to solve a set of similar problems. Specifically, we study how GP can, "within a run", solve multiple similar problems, whether in sequence or concurrently. This would involve only changing input-output requirements and priming the population with some solutions similar to

the target problem. It would reduce reliance on manual intervention, external libraries or restarts as required by executing multiple runs. While the benefits of a head start seem obvious, it is not immediately clear how to best provide it to GP. We can draw options from a number of observations. First, instructor programs ready for modification could seed the GP's initial population. These could be starting points for GP closer to a solution to the problem or more suited to easily reach a solution. Second, we could provide elements of a human curriculum for programming. We could take two consecutive problems A and B from a programming course, and try and learn one then the other (B after A and A after B), learn them simultaneously (A and B at same time because they share common subproblems and can share subsolutions) or a new problem that is a combination of problems A and B. This set of problems would extend the current GP program synthesis benchmark suite. Currently its problems have been selected for different criteria such as solely having input/output examples, multiple solutions, no synthesis method bias and that are representative of student programming problems [10] (see Section 2 for related work).

To proceed with these options, we selected two similar Python programming problems from an actual programming course, specifically *MITx 6.00.1x Introduction to computer science and programming in Python* (*6.00.1x*), a MOOC offered on the EdX platform. We modify a grammatical GP system [15], to allow GP to solve multiple programming problems in one run (see Section 3). The modifications are: **a)** a schedule to change one programming problem to another (by changing the input-output examples), **b)** initialization of the population with existing solutions to a programming problem. We seed the population with random (normal GP initialization) programs, human coded solutions, and programs generated from previous runs of GP. Our intent is to forgo inventing specialized operators or fitness measures.

We pose the following research questions (see Section 4 and 5) for GP: **1)** How does the quality of a solution or time to identifying a solution improve when GP switches to the problem "mid-stream", i.e. in the process of solving a similar problem? By how much does the timing of the switch impact performance? **2)** How does the quality of a solution or time to identifying a solution improve when GP 's population is initialized with solutions to similar problems? What proportion of similar problem solutions helps? In the context of head starts, we will use human judgment to select problems that are similar in terms of requiring the same program conceptual knowledge.

The key contributions (see Section Section 6) of this paper are: **1)** Introduction of a new program synthesis problem presented as a pair that is similar according to a programming curriculum. **2)** Development of schedule for multi programming problem synthesis. **3)** Analysis of head start concept based on multiple similar programming problems, programming problem schedules and initialization.

2 Related work

A head start for GP is intrinsically related to other attempts in GP that repurpose solutions. In GP the seeding of initial population as a means of headstart has been studied for e.g. software improvement, controllers, symbolic regression and AI planning [2, 28, 20, 32] Another thread of work is multi-task learning. It aims to solve multiple programming problems (tasks) simultaneously to improve on the performance of solving each programming problem independently. The assumption is that there exists some subsolutions (information) in common for solving related tasks [34]. This introduces a relation between modularity and multi-task learning, since the reuse of sub-tasks is promoted by modularity. In GP there have been multiple studies regarding modularity, as reported in this survey [8], though without explicit focus on multiple tasks.

While not all this work is on program synthesis, connections exist, e.g transfer learning of genetic programming instruction sets and libraries [9, 24]. Some recent work on program synthesis with GP also considers multiple functions, string library functions and vector problems [5, 27, 25]. Previous multi-task work in GP has focused on other domains for example, multi task visual learning, robot controllers, symbolic regression and Boolean problems [14, 16, 19, 18, 26, 13, 21]. Furthermore, a head start for GP is also related to non-stationary problems, due to the repurposing of the population. In non-stationary problems GP is presented with a continuum of different requirements [6] whereas when looking for a head start GP pivots explicitly between explicitly different problems.

GP program synthesis has used various techniques, see [17], and also considered specific programming approaches, such as recursion, lambda abstractions and reflection [22, 1, 33, 31]. An important milestone for GP is the program synthesis benchmark suite of 29 problems, selected from sources teaching introductory computer science programming [11]. We propose a new paired problem setup that could be introduced to the suite. We differentiate from others by investigating similar-problem solution initialization of the population, the switching from solving one problem to another, and a combination of the problem pair.

3 Method

In this section we first present a formalization of program synthesis and synthesis of multiple problems. Then we show how we proceed in a minimalist fashion by introducing two simple modifications that give GP a head start. We outline a variety of design options this allows. We end by reviewing Grammatical Evolution [23] which our GP framework uses.

3.1 Formalization

A formulation of program synthesis is as an optimization problem: find a program (solution) q from a domain Q that minimizes combined error on a set of input-output cases $d = \{(x_0, y_0), \ldots, (x_n, y_n)\}, x \in X, y \in Y$, with $q : X \rightarrow Y$.

Algorithm 1 $GP(\mathbf{D}, S, D_K, \Theta)$ *Multi-task GP with domain knowledge*
Parameters: $\mathbf{D}$ test cases, S programming problem schedule, D_K existing solutions as knowledge base, Θ hyper parameters
Return: Population

```
1: P ← ∅                                          ▷ Population
2: P ← P∪ initialize(Θ, D_K)                      ▷ Initialize population
3: P ← evaluate(P, Θ, F)                          ▷ Evaluate pop fitness
4: for t ∈ [1, ..., Θ_T] do                       ▷ Iterate over generations
5:     P′ ← selection(P, Θ)                       ▷ Select new population
6:     P′ ← variation(P′, Θ)                      ▷ Subtree mutation and crossover
7:     F ← getProgrammingProblem(S, t, D)         ▷ Get the fitness function
8:     P′ ← evaluate(P′, Θ, F)                    ▷ Evaluate population on programming problem
9:     P ← replacement(P′, Θ)                     ▷ Update population
10: return P                                      ▷ Return final population
```

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. There exists a set of programs $\mathbf{q}^* = \{q^*_0, \ldots, q^*_n\}$ that can solve all input-output cases. We can formulate the program synthesis problem as

$$\arg\min_{q \in Q}(q(x) - y)$$

Here we define learning of multiple programs as

$$\arg\min_{q \in Q} \sum_{x,y \in \mathbf{D}} (q(x) - y)$$

The data set $\mathbf{D}$ is $\{d_0, \ldots, d_n\}$, where d_i has an optimal solutions $\mathbf{q}^*_i$. Domain knowledge is expressed as similar solutions or sub-solutions, $D = \{q_i, \ldots, q_j\}$. For example, initial population head start in this paper is a population of solutions $P = [q_0, \ldots, q_n]$.

3.2 GP

We use a standard GP progtram synthesis algorithm for initializing with a head start, evaluating, selecting, varying and replacing multi-programming problem program synthesis, Algorithm 1. The modifications for a head start are: **1)** Injection of previous program synthesis solutions for initialization, see Alg. 1 line 2. **2)** A schedule to change the programming problem that is evaluated.

Scheduling multiple programming problems We design for both serial and parallel scheduling of multiple program synthesis (Alg. 1 line 7). With a serial schedule one programming problem is first evaluated and then another. With a parallel schedule multiple programming problems are evaluated at the same time, this is done in Alg. 1 line 7. More formally:

Serial One programming problem, $q^t = q_0$ is evaluated at generation t (intermediate), then another programming problem, $q^{t+1} = q_1$ is evaluated at generation $t + 1$. Fitness score is based on the current programming problem that is being evaluated, $F(q^t)$.

Parallel Multiple programming problems are evaluated at the same time, $\mathbf{q} = [q_0, \ldots, q_n]$. Fitness score is the sum of each programming problem fitness score, $\sum_{q \in \mathbf{q}} F(q)$.

Serial and Parallel Programming problems can be evaluated both in serial and parallel, $\mathbf{q}^t = [q_0]$ and $\mathbf{q}^{t+1} = [q_0, \ldots, q_n]$. Fitness score is based on the current programs that are being evaluated, $\sum_{q \in \mathbf{q}^t} F(q)$.

Existing solution initialization Different solutions to programming problems are seeded into the initial population **a)** Existing source code(solutions) for other human student problems, D^H **b)** Existing evolved solutions in the GP representation for other program synthesis programming problem, D^S **c)** Randomly generated programs using the standard GP initialization procedures, R.

Seeding the GP search with existing similar programs (options **a** and **b**) is one way of providing domain knowledge for the program synthesis. Note, this seeding can be seen as a variant of serial multi programming problem switching, i.e. the first programming problems have been synthesized to completion and provides a starting point for the next programming problem.

To initialize with existing programs, we need to parse the codes into the representation that we use for search. We do this using a grammar. First, we preprocesses the code and refactors variables and function names to a consistent naming scheme. Then, we recursively parse a tree representation of the code depth-first left-to-right and returns a list of integers indicating production choices (GE genome). The first matching production will be returned. Note, future work will investigate if there is any search bias in GE from the code parsing.

3.3 Grammatical Evolution

Grammatical Evolution (GE) is a genetic programming algorithm where a Backus Naur Form (BNF) context free grammar is used in the genotype to phenotype mapping process [23]. A production rule is defined as a non-terminal left-hand side, a separator and a list of productions on the right hand side, each production contains terminals and/or non-terminals. In GE the probability of selecting a production from a rule is depends on the number of productions. The grammar is the starting point for a two step sequence to decode a genotype to a program(phenotype): **1) Genotype to derivation tree**: The genotype, an integer sequence, rewrites non-terminals to terminal via the production rules. This rule production sequence can be represented as a derivation tree. At each step in the rewriting the integer "gene" determines which production to expand the current production rule. 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 sentence (executable code) that GE can evaluate.

Synthesized candidate programs in GE are evaluated in the same way as in GP, we provide a grammar as input to Alg. 1, and when initializing the existing solutions are parsed to a genotype representation. GE's genotype-phenotype mapping step raises locality issues [29]. One way to address the lack locality is

to use variation operators that manipulate subtrees. We chose GE since it can incorporate domain information and be used with Python.

4 Experiments

First, we present the program synthesis problems and data used for the initialization experiments. Then we present the experimental setup. Finally, we show the results and discuss them.

4.1 Solutions of Similar Problems

We identify programming problems that are similar according to human experts, composable into a combined program, and for which we have a corpus of correct and incorrect human coded solutions. We draw upon *MITx Introduction to computer science and programming in Python* (*6.00.1x*), a MOOC offered on the EdX platform [3]. Because they are also from an introductory programming course, they be similar in complexity to the problems in the GP program synthesis benchmark suite.

The learning design for *6.00.1x* assigns problems of progressiv complexity, e.g. Boolean operations, iterators and then combinations of iterators and Boolean operations. Because their similarity we focus on the first two problems *P1* (`count_vowels`) and *P2* (`count_bob`), which check the students' understanding of control flow. The two problems are similar, they initialize a count variable, iterate through a string, and add to a total count if some condition was met. We scraped solution history data from 2016 Term 2 and 2017 Term 1. Here we consider the correct solutions from the 3,485 who earned a certificate. We compared each solution to a gold standard solution on the basis of keyword frequencies using Pearson correlation. Most of the correct ones were correlated to the gold standard [3]. Note that privacy prevents a public release of this data set. We create a combination of *P1* and *P2* called *Combo*, all problems are in Figure 1.

We used the grammar in Figure 2 to both parse and generate solutions. We expect only a few distinct solutions, since we standardize them for parsing, filter for correctness, the course provides instructions for solving the problems, and solutions are available online. There are $\approx 2,000$ solutions for *P1* but 2 distinct solutions after parsing. For *P2* there are $\approx 1,000$ solutions and after parsing a single distinct solution. There are no existing student solutions for *Combo*, since we create it for GP. Note, GP copies the solutions in the initial population.

4.2 Experimental Setup

We report program synthesis performance in the same way as [11], 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

```
def count_vowels(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

def count_bob(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

def 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
```

Fig. 1. Problems *P1* (`count_vowels`), *P2* (`count_bob`) and *Combo* (`combo`) used for GP. In Python execution of Boolean operators is short-circuited.

```
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 ":}")
num : "0"|"1"|"2"
int_var : "i0"|"i1"|"res0"|"res1"
int_assign :  int_var "=" int
int :  int_var | ("int(" num ".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'"
```

Fig. 2. EBNF grammar for problems *P1*, *P2* and *Combo*

using Intel(R) Xeon(R) CPU E5-2450 v2 @ 2.50GHz. Our GP implementation is based on the `PonyGE2` [7, 12].

The set of parameters we use throughout all our experiments is listed in Table 1. We use subtree crossover on the GE derivation trees [7]. We use novelty selection since it was shown to be useful in program synthesis with GE [12]. Fitness is the number of correct test cases solved during training [11].

4.3 Experimental Design

For each approach, multiple variants were tested across a range of parameters. We specify both the programming problem schedule (programming problems to be solved, and in what order) and the percent of generations to spend on each programming problem. For initialization schemes, 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 2. The names in the figures are concatenations of these. Baseline is solving only one programming problem. All variants use the same number of fitness evaluations (16,000), regardless if they solve one or multiple programming problems with GP head start during the run.

Table 1. Experimental settings for GP

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

Table 2. Experimental variants, columns show the name and a description.

Variant Name	Description
Multi programming problem learning	
`EarlySingleSwitch`	Change programming problem after 25% of generations have passed
`MedSingleSwitch`	Change programming problem after 50% of generations have passed
`LateSingleSwitch`	Change programming problem after 75% of generations have passed
`OneThenTwo`	Initially optimize for *P1* and switch to *P2*
`OneThenBoth`	Initially optimize for *P1* and switch to *Combo*
`TwoThenOne`	Initially optimize for *P2* and switch to *P1*
`TwoThenBoth`	Initially optimize for *P2* and switch to *Combo*
Initialization Schemes	
`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
`UserSolutions`	Infuse population with student-submitted solutions
`NonDiverseRandom`	Infuse population with multiple copies of a single program that doesn't solve either programming problem

5 Results

In summary we observe that GP variants that solve multiple programming problems can improve the quality of the solutions for the more difficult problems. However, for the simple problem the GP head start gives no benefit. Table 3 outlines the experiments and results. Although the three problems are similar in structure, they have varying difficulties given our grammar. Figure 3 shows this difference, the *Combo* alone is the most difficult, and *P1* is more easily solvable than *P2* because there are no requirements on the loop variable.

Sequential programming problem schedules When solving a complex problem, it can be helpful to search for a problem solution to an intermediate programming problem (a sequential schedule) before solving the complete programming problem. For solving *Combo*, useful intermediate programming problems are to solve each of the *P1* and *P2* problems individually before moving to solve the combined problem. With this in mind, we solved multiple programming problem learning by starting to solve from either *P1* or *P2* and changing the programming problem to solve *Combo* at some point throughout the evolution. The

Table 3. Experimental results. Variants and problem are explained in Table 2. Percent of 100 runs solving all test cases is in the **Solved%** column for each variant.

Variant Name	Problem	Solved%
	Baseline	
Baseline	*P1*	(Best for *P1*) **97**
Baseline	*P2*	48
Baseline	*Combo*	3
	Switching	
`EarlySingleSwitch`	`TwoThenOne`	87
`MedSingleSwitch`	`TwoThenOne`	83
`LateSingleSwitch`	`TwoThenOne`	48
`EarlySingleSwitch`	`OneThenTwo`	(Best for *P2*) **49**
`MedSingleSwitch`	`OneThenTwo`	16
`LateSingleSwitch`	`OneThenTwo`	4
`EarlySingleSwitch`	`OneThenBoth`	3
`EarlySingleSwitch`	`TwoThenBoth`	11
`MedSingleSwitch`	`OneThenBoth`	1
`MedSingleSwitch`	`TwoThenBoth`	10
`LateSingleSwitch`	`OneThenBoth`	2
`LateSingleSwitch`	`TwoThenBoth`	(Best for *Combo*) **15**
	Initialization Schemes	
`UserSolutions, AllFromDir`	*P1*,	84
`UserSolutions, HalfFromDir`	*P1*	(Best init. *P1*) **94**
`NonDiverseRandom, AllFromDir`	*P1*	93
`NonDiverseRandom, HalfFromDir`	*P1*	93
`UserSolution AllFromDir`	*P2*	37
`UserSolution HalfFromDir`	*P2*	42
`NonDiverseRandom, AllFromDir`	*P2*	(Best init. *P2*) **51**
`NonDiverseRandom, HalfFromDir`	*P2*	45

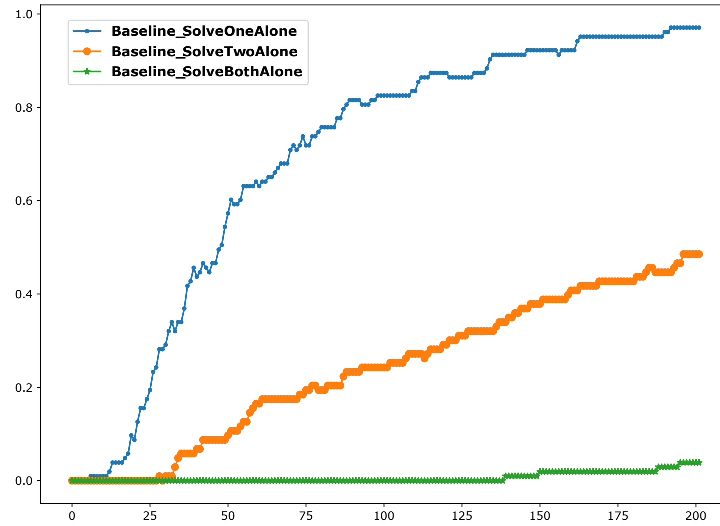

Fig. 3. Y-axis is the fraction of runs which contained a program that solved all of the test cases and x-axis is generation. Lines are a single problem (*P1*-`Baseline_SolveOneAlone`, *P2*-`Baseline_SolveTwoAlone`, or *Combo*-`Baseline_SolveOneAlone`)

percentage of runs in which at least a single program solved all given test cases is given in Figure 4. 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 *P1* and then switched to spending 150 generations on *P2*, then it would be shifted back 50 generations relative to the other lines, to get a common start point for the second problem.

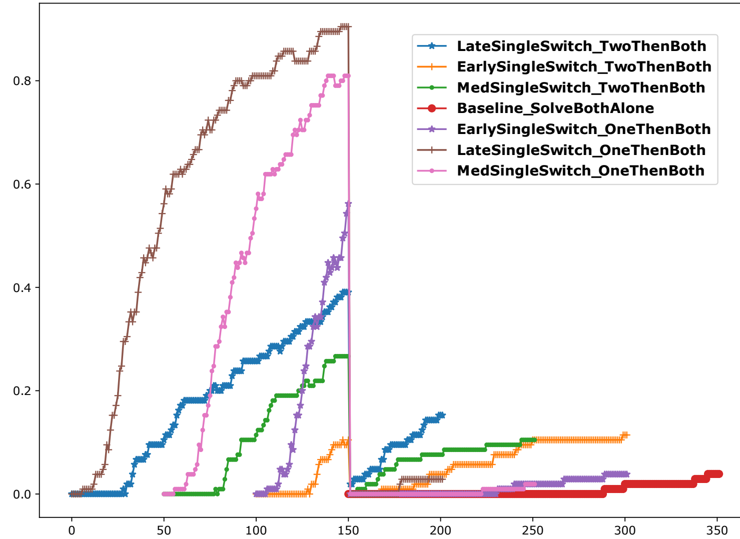

Fig. 4. Y-axis is the fraction of runs which contained a program that solved all of the test cases and x-axis is generation. The start of the lines is shifted in order to line up the point at which they started working on the final problem. The lines are the multi programming problem learning experiment, when switching from solving `OneThenBoth`, `TwoThenBoth` and not switching `Baseline_SolveBothAlone`

Populations that spent more time searching for the intermediate problem (`LateSingleSwitch`) generally had a higher upward trajectory than those that spent less time searching for it (`EarlySingleSwitch`). This effect can be seen in Figure 5, which shows the number of test cases solved by the best individual in a run, averaged across all runs for the same problem. A longer time spent searching for the intermediate programming problem allows a population to solve a higher number of test cases from *Combo* earlier, in comparison to those that spent less time searching on the intermediate programming problem.

Additionally, this experiment revealed that harder problems can sometimes be a better intermediate signal than easier ones. *P1* is an easier problem to solve for the population than *P2*, and Figure 4 shows better performance when searching for *P2* before *Combo*, as opposed to searching for *P1* before *Combo*.

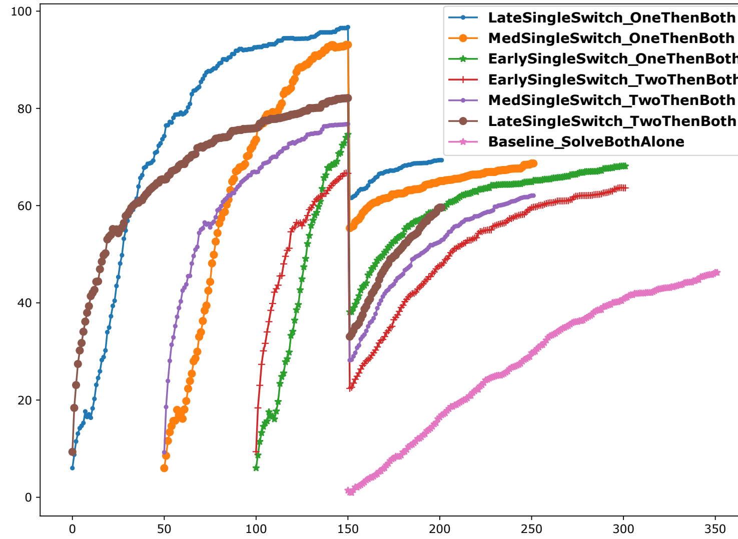

Fig. 5. Y-axis is the average number of test cases solved by the best individual and x-axis is the generation. The start of the lines is shifted in order to line up the point at which they started working on the final problem. Lines are for when switching from solving `OneThenBoth`, `TwoThenBoth` and not switching `Baseline_SolveBothAlone`

Transferring Information from One Programming Problem to Another The two problems are similar, in that they initialize a count variable, iterate through a string, and add to a total count if some condition was met. A population that had previously been optimized for solving *P1* could be better equipped to solve *P2* than a population solely focused on *P2* from the beginning.

We can see that the effect of searching for a previous programming problem was different in each case. When going from *P2* to *P1* in Figure 6, this intermediate programming problem 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 6, we can see that while too many generations on a related problem can be harmful, the right amount can be beneficial. E.g. the `MedSwitch` and `LateSingleSwitch` schemes perform worse than the baseline, the `EarlySingleSwitch` performed better than the baseline, even though 25% of the time was spent searching for a different problem.

Initializing with Preexisting Knowledge Because of the similarity in the solutions to each of the given problems, our hypothesis was that initializing populations with individuals that solved *P1* would be better equipped at solving *P2*, and vice versa, by capturing common statements. None of the runs initialized with solutions to one problem did any better than the runs in which random initialization methods were used. Surprisingly, when a single program which solved neither problem was chosen as a starting point, *P2*, `NonDiverseRandom, AllFromDir`,

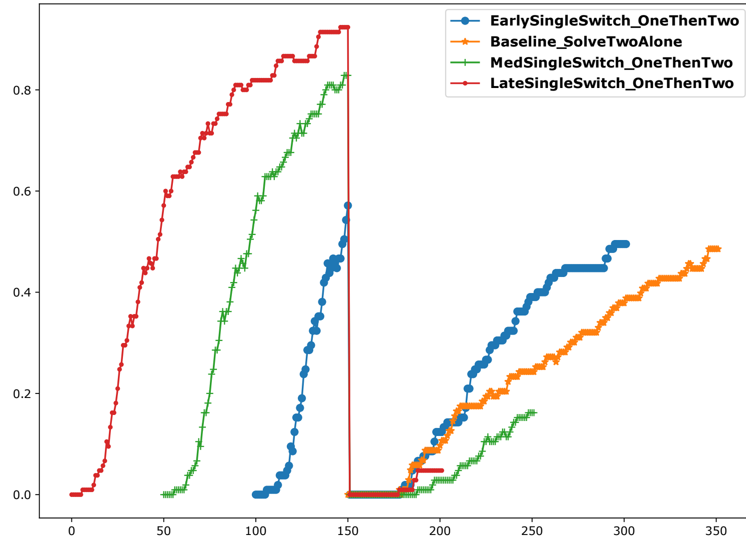

Fig. 6. Y-axis is the fraction of runs which contained a program that solved all of the test cases and x-axis is generations. The start of the lines is shifted in order to line up the point at which they started working on the final problem. Lines are for when the programming problem was switching from solving `OneThenTwo` and not switching `Baseline_SolveTwoAlone`

performance was not hindered. In fact, it even solved the problem in a higher number of runs than the baseline.

5.1 Discussion

This study attempts to clarify one simple approach to improve GP performance and investigate the experimental design options for solving multiple programming problems for GP. This leads to a number of limitations in our proof-of-concept study. First, the human solution data is limited and biased by amount and diversity of human solutions that are available for programming problems. Second, we have only investigated a few programming problems, and can thus not draw any strong conclusions. Another limit is that we use only informal human judgment of similarity. There exists a body of work in education and software engineering regarding programming problem similarity that may offer some help. Finally, another limit is that there is no consensus on curriculum learning design for humans, so GP learning designers might struggle to specify a curriculum as well.

It is our aim is to expand the boundaries of expectations regarding what is provided to GP. Others have questioned why GP starts "from scratch" when helpful knowledge is available. For example [12] investigated providing domain knowledge to the programming synthesis task that GP tried to solve. On the basis of our experiments, it is arguable that giving GP a head start could improve

GP performance on the more difficult problems in the GP program synthesis benchmark suite assuming that similar problems could offer a head start.

6 Conclusions and Future Work

We explored transferring information within a single population by solving separate but related programming problems, the first two problems of the first MITx 6.00.1x course problem set. Specifically, we explored **1)** solution quality from a transfer between programming problem using a schedule for GP program synthesis to solve a similar programming problem, **2)** solution quality when providing information from similar programming problems during initialization. When solving the more difficult *Combo* programming problem with GP, using the P2 problem as an intermediate goal as opposed to the simpler P1 was found to be beneficial. Additionally, more time spent searching for the intermediate goal proved to be beneficial. The effects of initializing a population with solutions to a related problem were unclear.

There are multiple directions for future work. An evaluation of the difficulty of finding additional problem pairs that set up the head start could be conducted. This could also reveal whether the initial observations around ordering and problem difficulty hold more generally. Head start also could be expanded to integrate existing and new work on modularity and reuse in GP. Other metrics for comparing code similarity will also be investigated.

References

1. Agapitos, A., Lucas, S.M.: Learning recursive functions with object oriented genetic programming. In: Collet, P., Tomassini, M., Ebner, M., Gustafson, S., Ekárt, A. (eds.) Proceedings of the 9th European Conference on Genetic Programming. Lecture Notes in Computer Science, vol. 3905, pp. 166–177. Springer, Budapest, Hungary (10 - 12 Apr 2006)
2. Arcuri, A., White, D.R., Clark, J., Yao, X.: Multi-objective improvement of software using co-evolution and smart seeding. In: Asia-Pacific Conference on Simulated Evolution and Learning. pp. 61–70. Springer (2008)
3. Bajwa, A., Bell, A., Hemberg, E., O'Reilly, U.M.: Analyzing student code trajectories in an introductory programming mooc. In: 2019 IEEE Learning With MOOCS (LWMOOCS). pp. 53–58. IEEE (2019)
4. Bengio, Y., Louradour, J., Collobert, R., Weston, J.: Curriculum learning. In: Proceedings of the 26th annual international conference on machine learning. pp. 41–48 (2009)
5. Bladek, I., Krawiec, K.: Simultaneous synthesis of multiple functions using genetic programming with scaffolding. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion. pp. 97–98 (2016)
6. Dempsey, I., O'Neill, M., Brabazon, A.: Foundations in grammatical evolution for dynamic environments, vol. 194. Springer (2009)
7. Fenton, M., McDermott, J., Fagan, D., Forstenlechner, S., Hemberg, E., O'Neill, M.: Ponyge2: Grammatical evolution in python. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. pp. 1194–1201 (2017)

8. Gerules, G., Janikow, C.: A survey of modularity in genetic programming. In: 2016 IEEE Congress on Evolutionary Computation (CEC). pp. 5034–5043. IEEE (2016)
9. Helmuth, T., Pantridge, E., Woolson, G., Spector, L.: Transfer learning of genetic programming instruction sets. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion. pp. 241–242 (2020)
10. Helmuth, T., Spector, L.: Detailed problem descriptions for general program synthesis benchmark suite. School of Computer Science, University of Massachusetts Amherst, Tech. Rep. (2015)
11. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. pp. 1039–1046. ACM (2015)
12. Hemberg, E., Kelly, J., O'Reilly, U.M.: On domain knowledge and novelty to improve program synthesis performance with grammatical evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 1039–1046 (2019)
13. Hoang, T.H., Essam, D., McKay, R.I.B., Hoai, N.X.: Developmental evaluation in genetic programming: The TAG-based frame work. International Journal of Knowledge-Based and Intelligent Engineering Systems **12**(1), 69–82 (2008). https://doi.org/doi:10.3233/KES-2008-12106, http://content.iospress.com/articles/international-journal-of-knowledge-based-and-intelligent-engineering-systems/kes00142
14. Jaśkowski, W., Krawiec, K., Wieloch, B.: Multitask visual learning using genetic programming. Evolutionary computation **16**(4), 439–459 (2008)
15. Kelly, J., Hemberg, E., O'Reilly, U.M.: Improving genetic programming with novel exploration - exploitation control. In: European Conference on Genetic Programming. Springer (2019)
16. Koza, J.R.: Evolution of subsumption using genetic programming. In: Proceedings of the First European Conference on Artificial Life. pp. 110–119 (1992)
17. Krawiec, K.: Behavioral program synthesis with genetic programming, vol. 618. Springer (2016)
18. Krawiec, K., Wieloch, B.: Functional modularity for genetic programming. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation. pp. 995–1002 (2009)
19. Krawiec, K., Wieloch, B.: Automatic generation and exploitation of related problems in genetic programming. In: IEEE Congress on Evolutionary Computation. pp. 1–8. IEEE (2010)
20. Langdon, W.B., Nordin, J.: Seeding genetic programming populations. In: European Conference on Genetic Programming. pp. 304–315. Springer (2000)
21. Lopez, U., Trujillo, L., Silva, S., Vanneschi, L., Legrand, P.: Unlabeled multi-target regression with genetic programming. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference. pp. 976–984 (2020)
22. Lucas, S.: Exploiting reflection in object oriented genetic programming. In: Keijzer, M., O'Reilly, U.M., Lucas, S.M., Costa, E., Soule, T. (eds.) Genetic Programming 7th European Conference, EuroGP 2004, Proceedings. LNCS, vol. 3003, pp. 369–378. Springer-Verlag, Coimbra, Portugal (5-7 Apr 2004)
23. Ryan, C., Collins, J.J., O'Neill, M.: Grammatical evolution: Evolving programs for an arbitrary language. In: European Conference on Genetic Programming. pp. 83–96. Springer (1998)
24. Ryan, C., Keijzer, M., Cattolico, M.: Favourable biasing of function sets using run transferable libraries. In: Genetic Programming Theory and Practice II, pp. 103–120. Springer (2005)

25. Sasanka, R., Krommydas, K.: An evolutionary framework for automatic and guided discovery of algorithms. arXiv preprint arXiv:1904.02830 (2019)
26. Scott, E.O., De Jong, K.A.: Automating knowledge transfer with multi-task optimization. In: 2019 IEEE Congress on Evolutionary Computation (CEC). pp. 2252–2259. IEEE (2019)
27. Soderlund, J., Vickers, D., Blair, A.: Parallel hierarchical evolution of string library functions. In: International Conference on Parallel Problem Solving from Nature. pp. 281–291. Springer (2016)
28. Tanev, I., Kuyucu, T., Shimohara, K.: Gp-induced and explicit bloating of the seeds in incremental gp improves evolutionary success. Genetic Programming and Evolvable Machines **15**(1), 37–60 (2014)
29. Thorhauer, A., Rothlauf, F.: On the locality of standard search operators in grammatical evolution. In: International Conference on Parallel Problem Solving from Nature. pp. 465–475. Springer (2014)
30. Thrun, S.: Explanation-based neural network learning: A lifelong learning approach, 1996
31. Wan, M., Weise, T., Tang, K.: Novel loop structures and the evolution of mathematical algorithms. In: Silva, S., Foster, J.A., Nicolau, M., Machado, P., Giacobini, M. (eds.) Genetic Programming - 14th European Conference, EuroGP 2011, Torino, Italy, April 27-29, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6621, pp. 49–60. Springer (2011). https://doi.org/10.1007/978-3-642-20407-4_5, https://doi.org/10.1007/978-3-642-20407-4_5
32. Westerberg, C.H., Levine, J.: Investigation of different seeding strategies in a genetic planner. In: Workshops on Applications of Evolutionary Computation. pp. 505–514. Springer (2001)
33. Yu, T., Clack, C.: Recursion, lambda-abstractions and genetic programming. In: Poli, R., Langdon, W.B., Schoenauer, M., Fogarty, T., Banzhaf, W. (eds.) Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming. pp. 26–30. CSRP-98-10, The University of Birmingham, UK, Paris, France (14-15 Apr 1998)
34. Zheng, X., Qin, A., Gong, M., Zhou, D.: Self-regulated evolutionary multi-task optimization. IEEE Transactions on Evolutionary Computation (2019)