VanillaGP: Genetic Algorithm for Inductive Program Synthesis

Bachelor Thesis for CSE3000

by Farhad Azimzade

VanillaGP: Genetic Algorithm for Inductive Program Synthesis

Farhad Azimzade*
Supervisor: Sebastijan Dumančić †
EEMCS, Delft University of Technology, The Netherlands

January 26, 2022

Abstract

Inductive Program Synthesis is the problem of generating programs from a set of input-output examples. Since it can be reduced to the search problem in the space of programs, many search algorithms have been successfully applied to it over the years. This paper proposes, develops, and analyses a novel algorithm in the family of Genetic Algorithms, called VanillaGP. While generally not showing superior performance compared to a recent best-first Brute method on the subset of program synthesis tasks used in the paper, VanillaGP does appear to reach a comparable relative improvement of the errors in the training data.

1 Introduction

Program synthesis is the problem of generating general computer programs from a starting set of conditions, requirements, or samples. It allows for automatic generation of algorithms given a problem domain, namely its Domain Specific Language (DSL), and some description of the intended behavior. Program synthesis has been used to (re)discover complex tree-based algorithms [5], as well as database queries from a handful of examples [6], therefore making a strong case for its usefulness and effectiveness.

The general approach is to search the program space in an attempt to stumble upon a suitable program that covers the specified preconditions. Since program synthesis can be reduced to a search problem in the space of programs defined in a particular DSL, a number of search techniques and heuristics can be applied to solve program synthesis.

Recently, a search algorithm, dubbed Brute [2], has been proposed to tackle program synthesis. The method used in Brute is a best-first approach, which means that as programs are being explored, only decisions and steps that immediately improve the loss function are taken. Brute first looks for a function that, when applied to the input, results in the closest output to the one desired. Then, it iteratively searches the set of functions to append to

*f.azimzade@student.tudelft.nl
†S.Dumancic@tudelft.nl

the current best program. This strategy, while producing decent results, is susceptible to converging to a sub-optimal solution too quickly. That is, the Brute algorithm will find the locally optimal solution (i.e. the program that produces the lowest error of all the programs explored up to that point). However, such a solution may not be the globally optimal solution, and the algorithm will inevitably miss this fact, as it never backtracks.

One way of going beyond locally optimal solutions is the family of search algorithms called Genetic Algorithms (GA). GAs attempt to emulate the process of reproduction and gene-passing in biological systems. These algorithms essentially sample random points in the search space, and then continue to transform them, by crossing them together or mutating them, in an attempt to increase the resulting fitness of each found solution.

Classic GAs work with fixed-length encoding of the search space. In this case, the space of all the programs generateable from the set grammar of the DSL does not adhere to that property, since programs can be of varying lengths. This means that gene cross-overs that take place within an iteration of a classic GA are not quite as suitable. Instead, an approach popularized by John R. Koza, called Genetic Programming (GP) [4], may seem more compelling in this respect, as it tackles programs in their tree form directly. The algorithm this paper proposes draws some inspiration from Koza's work, but effectively attempts to adapt the classic GA (i.e. working on strings rather than trees) to the problem of program synthesis.

The core idea of the algorithm, VanillaGP, is to apply the genetic algorithm method to the search for programs in the enriched space of programs. What is meant by "enriched" is the preliminary step of generating auxiliary functions, such that the pool of programs the genetic algorithm can sample from as its *function alphabet* is larger than the set of the basic functions defined by the DSL. This step ensures that a sufficient complexity can be maintained in the resulting programs. Additionally, it allows for the simplification of the search algorithm, as the need for constructing those sub-programs (or functions) is largely omitted and delegated to this so-called *invent* stage. This step of preliminary invention of functions of a certain complexity actually comes directly from the Brute algorithm [2]. There it assits the best-first search algorithm, while in VanillaGP, a GA uses those functions alongside the basic DSL-defined functions as genes in individual program genotypes.

This research paper attempts to investigate how well and under what conditions VanillaGP performs on the problem of program synthesis using three types of experiments as an evaluation proxy. These experiments, as described in [2], are experiments with pixel images, robot agent navigation, and string transformation. One of the issues this paper explores is the impact of using a genetic algorithm, such as VanillaGP, in program synthesis on the ability to escape local extrema. The stochasticity of VanillaGP has the potential to explore the problem space more boldly and not be tempted by local extrema as much as a best-first algorithm, such as Brute, would. Another point of interest is the generalizability of the found problems. This is primarily relevant to the string transformation domain, whereby the program found by looking at training examples has to solve or almost solve the test samples as well.

Further, section 2 of the paper lays out the general approach taken within this research project to investigate the research questions. Section 3 introduces the basic notion of a

program that is used with respect to inductive program synthesis, as well as a general introduction to Genetic Algorithms. The full description of the VanillaGP algorithm itself is in section 4. Section 5 describes the experiments conducted in the process and the corresponding results obtained. In section 6, some of the ethical considerations of the research project are discussed. Section 7 concludes the paper and exposes some of the limitations of this current work and potential remedies.

2 Methodology

This research paper extends the recent work done in [2] on applying a best-first search algorithm with a *distance function* to the problem of Inductive Program Synthesis (IPS). The key takeaway from that work is that a fairly straightforward deterministic algorithm can be useful in program synthesis if a distance function is used that allows to differentiate between two candidate solutions when neither is the desired solution for the problem. That is, the usage of a distance function can provide the search algorithm with the additional information as to how far the produced solution is from the desired result. This in turn leads the algorithm down the potentially more beneficial search paths.

Furthermore, this research work is part of a larger collaborative peer-group effort that explores various alternative search algorithm paradigms in their applicability to IPS. The peer group involves five algorithm designs, one of which is the Genetic Algorithm, that were researched, constructed, and tested on the data that was used in the Brute paper.

The original paper on Brute implements the algorithm in a logic programming language, Prolog. While having its advantages, it could be argued that a more common imperative language implementation would be more simple to work with going forward. As such, the peer group settled on the idea of re-implementing both the original Brute algorithm and each of the novel algorithms in Python.

Alongside the algorithms, the testing environment had to be implemented as well. This involves the definitions of programs that algorithms would be working on, the domain-unique token functions that make up the programs, the interpreter for running the programs, and a variety of parsers for the training and test data.

Finally, experiments examining the performance of the search algorithms were conducted. The three experiment domains are the ones described in the Brute paper [2], and some of the similar properties of interest were analysed, such as solution percentage per task complexity.

3 Background

3.1 What Are Programs?

Since the particular flavour of program synthesis tackled in this paper is inductive program synthesis, the algorithm for the generation of programs is supplied with input-output pairs that constitute the positive examples for the problem. That is, the desired program P_c needs to meet the requirement that for each pair (in, out), $P_c(in) = out$ holds.

The positive examples are themselves pairs of input states and output states. The nature of the states is experiment-dependent, which means each of the three experiments imposes its own interpretation of the state. However, the general abstraction of the state is the type Env (environment). So, an example supplied to the algorithm would be a pair (in, out) of type (Env, Env).

Finally, the program itself represents a pipeline of functions that transforms a given input state into the output state of the same type. A program then is a sequence:

$$P_i = f_n \circ f_{n-1} \circ ... \circ f_1$$

where each function f_i is of type $Env \to Env$.

One aspect relevant to genetic algorithms particularly is that such a definition of a program is more natural in the context of genetic algorithms than the more traditional representation of programs in a tree form. This is because in GA, genomes are represented as strings, with each character coming from some (usually binary) alphabet [7]. As such, a program can be thought of as a *genome* where each *gene* comes from a set of the pre-defined functions (*function alphabet* F):

$$F = \{f_0, f_1, ..., f_n\}$$
$$P_i = [f_j \ : \ f_j \in F]$$

3.2 Genetic Algorithm Blueprint

Genetic algorithms are crude approximations of the basic processes involved in the evolution of biological systems. They commonly involve two fundamental operations: gene crossover and gene mutation. Crossovers emulate the reproduction of two members of a population, while the mutation models the potential random gene mutations in the DNA of biological organisms. The proposition of genetic algorithms is that such an iterative process of breeding and mutation can lead to better performing genotypes (i.e. stronger evolved organisms).

One of the core operations in genetic algorithms is that of genetic crossover. A crossover is the process that involves two members of a population exchanging parts of their genetic material and producing new members of the same "species", broadly speaking. As per [7], genetic algorithms commonly work on binary strings. In this case, the genotype of each member is a binary string. This means that during crossover, parts of one parent genotype of a population are combined with some other part of another parent's genotype to produce a child genotype.

Crossovers come in different sizes. The simplest is a one-point crossover. This is a situation whereby the two parent strings are cut in one place each. A single point is selected in one parent, and a potentially different point is selected in the second parent. Then, the part up until the selected point of the first parent and the part from the selected point of the second parent onward are combined, resulting a single string that represents one child. The same is done with the second pair of substrings, which produces the second child. The more complex crossovers are n-point crossovers. The essentially execute the same operation as the one-point crossover, except, as the name suggests, they cut each string n times. This

results in a more complex recombination of children programs.

The other fundamental operator is genetic mutation. This is a process by which genomes within a population may be altered with a certain probability. Each gene can be replaced with one of the other available *alleles*. In the traditional case of binary strings, the alleles that a gene can take are the two entries of the binary alphabet {0, 1}. So, in this case, a random mutation is akin to a random bit flip in a binary string.

4 Genetic Algorithm: VanillaGP

As mentioned in Section 3, a program comprises functions with a single argument of type *Env* and a single output of type *Env* as well. Koza's classic approach to genetic programming [4], however, operates on abstract syntax trees (ASTs), whereby each function in a program can have any number of arguments but only one output. This difference brings VanillaGP closer to ordinary genetic algorithms than Koza's vision of GP, although technically, it still is a program generation search algorithm.

VanillaGP is an iterative genetic algorithm that performs a set number of genetic operations repeatedly to search for and find an appropriate solution. Figure 1 shows a diagram of a single iteration of the algorithm.

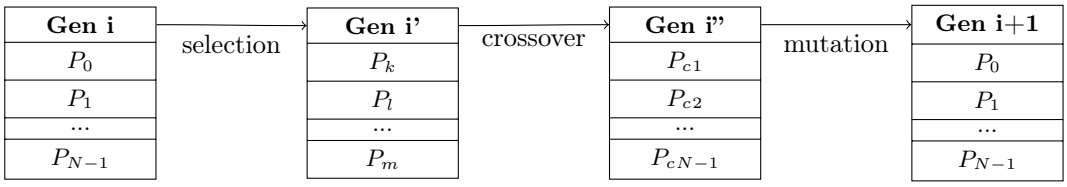


Figure 1: A single iteration of the VanillaGP algorithm

The algorithm starts with a randomly generated population of programs, such that each program in the initial population is at most of length l_{max} and the total number of programs in the population $\{P_0, P_1, ..., P_{N-1}\}$ is N. Usually, the initial population performs fairly poorly on the tasks, although given a high enough N, these random programs may happen to stumble upon solutions to the simplest of problems within a given domain.

Next, programs from the initial population *Gen* 0 need to be considered for inclusion in the next generation. A sensible approach is to take into the account the corresponding fitness values of the programs. That is, the more fit individual programs ought to get a better chance of propagating their genes onward during crossover. This filtering of programs is the purpose of the *selection algorithm*.

The selection algorithm is SUS (Stochastic Universal Sampling). SUS [1] is a method of stochastically selecting elements from a set based on some weighting. In this case, it is used to select N programs as candidates for reproduction from one generation based on their fitness values. The idea behind it is to represent the fitness of each program as a proportionally

scaled section of a metaphorical circular wheel. Then, N randomly-generated equidistant points are selected on the wheel. The "bins" or sectors of the wheel that these points land on then determine which N programs get selected.

In order for SUS to determine the proportions on the wheel, each program in the generation needs to be assigned a fitness value. This is the purpose of the $fitness\ function$. In the case of VanillaGP, the fitness function is the reciprocal of the cumulative error obtained from running every candidate program on each of the sample inputs and comparing the resulting state with the desired output state. The discrepancy in the two is the resulting $error$. The relationship between the fitness and error of a program P_i is as follows:

$$fitness\ function(P_i) = \frac{1}{error_i}$$

where the value of the $error_i$ itself is computed as:

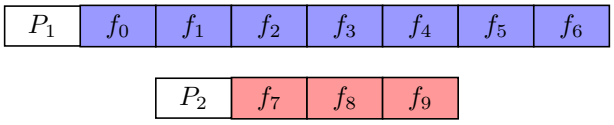
$$error_i = \sum_{j \in examples} error_i^j$$

This means that for a given training example (in^j, out^j), the distance between the output state resulting from $P_i(in^j) = out_i^j$ and the true/desired output state out^j is computed by the $distance\ function$ and stored as the $error_i^j$. Having done that for each training example, the function sums the distances up to get the cumulative error.

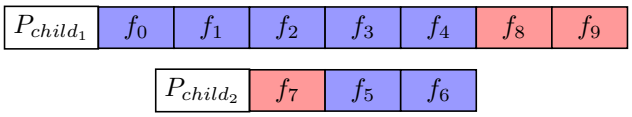
One thing to note here is that the cumulative error function can also return an infinite error. This occurs when the interpreter encounters a run-time error while a candidate program P_i is being evaluated on one of the inputs in^j. This indicates that the particular program attempted to perform illegal operations, such as, for instance in the case of the robot domain, escaping the bounds of the grid. Alternatively, the error function can return 0 value in the case that the program P_i successfully solves all the specified examples. These two special cases are handled by assigning zero fitness to invalid programs and infinite fitness to the programs that fully solve the examples.

If after the selection is complete, a program ends up chosen, it is paired up with another selected program to produce two child programs in the process of $crossover$.

The type of crossover used here is a straightforward one-point crossover, which means given two sequences of functions, each one will be cut at a single point and then recombined with the other part of the second sequence. An example of such a procedure is as follows:
Given are two parent programs:

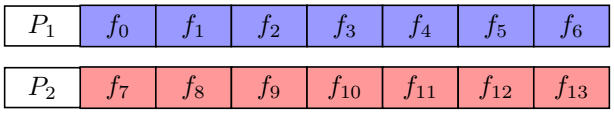


After the crossover, the resulting child programs are:

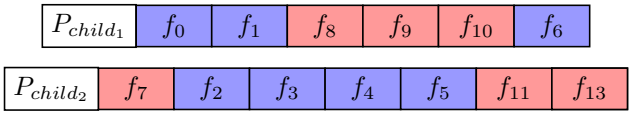


Here, the one-point crossover uses index 4 in the first program and index 0 in the second program. Crossover produces two child programs with a mixture of the functions from the parent programs.

A more intricate crossover is the *n-point crossover*. This operator cuts the program genome into n random sublists and then recombines them into two child programs. In theory this provides more program diversity, as the children bear less similarity to the parent programs. An example of a 2-point crossover starts with two parent programs:

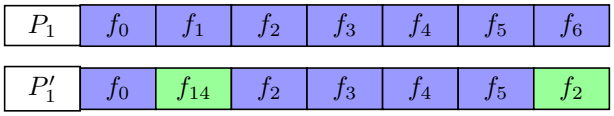


before each gets cut into 3 randomly-sized pieces and recombined:

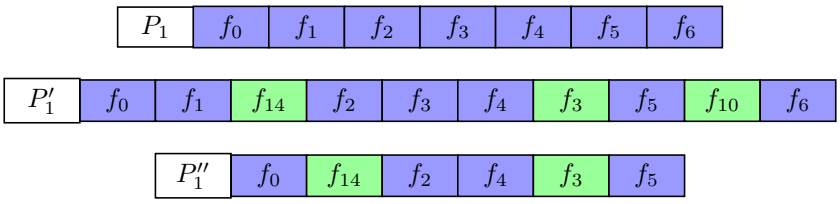


As a result, the child programs have more diversity in their genome, since, parts of the parent genomes more freedom for ordering themselves in the child genomes.

The final operation performed on the generation is *mutation*. Traditionally, mutation involves replacing a gene in the genome of an individual by a different *allele* – a possible value of the gene. In this case, replacement gene comes from the pool of functions. Here, the sequence of functions in each program is altered by substituting another function from the pool of possible functions in place of an existing function in the sequence. So, given a program such as $P_i = [f_0, f_1, f_2, f_3, f_4, f_5, f_6]$, a random mutation operation may choose to mutate the function f_1 into a function f_{14}, where $f_{14} \in function\ alphabet$. This results in the program $P_i' = [f_0, f_{14}, f_2, f_3, f_4, f_5, f_6]$. Such an example (but with one additional mutation) is shown here:



However, in the hope of attaining more program diversity, VanillaGP also implements a different mutation operation, called UMAD (Uniform Mutation by Addition and Deletion) [3]. It is a variation of the mutation operator that allows for, as the name suggests, adding and deleting genes from a genome. UMAD works by first iterating through the genome of an individual and adding genes with a certain probability either right before or right after an existing gene. Then it iterates again to perform deletions with some probability. Such a process has the potential of increasing or decreasing the lengths of genomes stochastically, resulting in a potentially more diverse population. The following is an example of a single UMAD operation:



Here, after the addition step, the program P_1 has been enriched with three randomly chosen functions f_{14} f_3, and f_{10} to produce an intermediate program P_1'. Then, in the deletion step, functions that happen to have been deleted are f_1, f_3, f_{10}, and f_6. As can be seen, the starting program is longer than the resulting program, whereas the ordinary mutation operator preserves the size of the program.

At the end, the initial generation is transformed into *Gen* 1. This process is repeated for a set number of iterations, in the hope of producing a final generation that contains one of the plausible solutions to the original problem. Furthermore, the best program from each generation is saved, such that if the evolutionary process ends up with a worse-performing program, the algorithm manages to return the best-performing program it has encountered over the course of the run.

5 Experiments and Results

5.1 Experimentation Environment

In order to analyse the performance of VanillaGP, it has been deployed in the experimentation environment. The experimentation environment follows the one outlined in [2] and involves three testing domains: robot, pixel, and string domains. Each domain comes with a set of possible atomic functions and specified testing and, in the case of the string domain, training samples. These samples themselves are grouped together by their *task complexity*, which is a property defined for each domain. Task complexity, as the name implies, is a measure of the difficulty that the algorithm is expected to have when generating the appropriate program. It also means a larger potential distance between input and output states, which makes the search for the solution tougher. Each domain has five complexity groups.

As for the atomic functions, each domain has a specific set of transition functions (ones that return a transformed state) and boolean functions (ones that return a boolean truth value). In addition, programs in all domains can use two common functions: an if-then-else statement and a while loop, the conditions for which come from the boolean function set, and the bodies are programs in their own right and can be sequences of any functions valid in the domain.

The **robot domain** represents a challenge of directing a robot agent across a grid, one cell of which contains a ball that can be picked up and moved. The goal for the algorithm is to find a program that, given the starting positions of the robot and the ball, can instruct the robot to pick up the ball and drop it off at the desired position – goal cell.

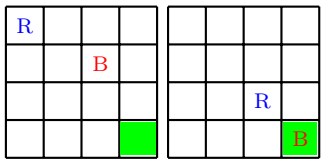


Figure 2: An example of the input and output states of the robot experiment

The list of possible base functions here is:

[*MoveLeft, MoveRight, MoveUp, MoveDown, Grab, Drop*]

and eight boolean functions:

[*AtLeft, AtRight, AtTop, AtBottom, NotAtLeft, NotAtRight, NotAtTop, NotAtBottom*]

Task complexity in the robot domain refers to the size of the grid that is used in an example. The example in Figure 2, for example, is a 4×4 grid, which places it in the second complexity group. In total there are examples of complexity 2, 4, 6, 8, and 10, based on the grid sizes.

The **pixel domain** poses a challenge of drawing binary pixel images of characters/symbols given the desired ASCI character. The input states are empty pixel canvases, and the task is to produce a program that can walk around the grid and draw in the appropriate pixel values.
The list of possible base functions in this domain is:

[*MoveLeft, MoveRight, MoveUp, MoveDown, Draw*]

and eight boolean functions:

[*AtLeft, AtRight, AtTop, AtBottom, NotAtLeft, NotAtRight, NotAtTop, NotAtBottom*]

The **string domain** is the most complex domain of the three and is, essentially, the problem of discovering an appropriate program that can transform the set input strings into the corresponding output strings. This is also the only domain that requires a training dataset as well as the test set. This is because the algorithm is expected to discover a potentially complex rule/mapping that the training set adheres to and then apply it to the test set to demonstrate its competence and *generality*.

The training examples fed into the algorithm are of the form:

"Kurt Gödel" $\implies$ "KG"

Here, the rule that the algorithm is expected to discover is that the string containing the first and last names of a person should be converted to the string of just the initials.

First, the algorithm attempts to minimize the error for the training data. This produces a program that, in theory, correctly transforms the training inputs into outputs. Then, to check the generality of the solution program found by the algorithm, the program is run on the unseen, test examples. These two evaluations result in the *training* and *test* errors of the best-found program respectively.

Furthermore, specific to the string domain, samples are run in a fashion resembling a traditional *k-fold cross validation*. Given n input-output examples, each sample case contains $n_{train} < n$ training examples that are fed into the algorithm to discover the underlying rule/mapping of strings. The sample case also contains $n_{test} = n - n_{train}$ test examples that are unseen examples that are used to test the final program returned by the algorithm after the training is complete. In total, there are multiple such cases for the same set of input-output examples, such that each sample case uses a different training and testing subsets, hence the resemblance to k-fold validation.

The list of possible base functions here is:

[*MoveLeft, MoveRight, MakeUppercase, MakeLowercase, Drop*]

boolean location functions:

[*AtStart, AtEnd, NotAtStart, NotAtEnd*]

and boolean character functions:

[*IsLetter, IsNumber, IsSpace, IsUppercase, IsLowercase, IsNotLetter, IsNotNumber, IsNotSpace, IsNotUppercase, IsNotLowercase*]

5.2 Results and Discussion

With the experimentation environment defined, VanillaGP could be run to check its performance. One of the main properties of interest when examining the results of the experiments is the percentage of tasks solved for each complexity class. This measure is aimed at comparing the competence of an algorithm as the tasks become tougher. In most cases, one would expect the performance to go down as the complexity increases.

To compare Brute and VanillaGP, both algorithms were given the same amount of time per sample task. In this case the time limit was 1 minute, after which, if the algorithms did not find a solution to the task, they had to terminate and return the best program they had found. Specific to VanillaGP, all experiments were run with the population size $N = 200$, maximum initial program length $l_{max} = 10$, and allowed to run for 200 generations at most. The caveat here is that due to the sheer number of tasks and complexity of the string domain, the time limit was set to only 10 seconds per task. Figure 3 shows the results of the runs on the three domains.

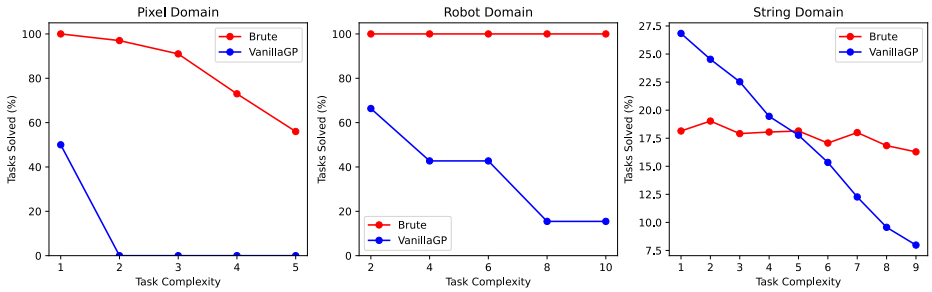


Figure 3: Solved percentage against complexity of Brute and VanillaGP.

As can be seen, the initial results of running VanillaGP on the three domain experiments left quite a lot to be desired. It was evident that it could solve only some of the simpler examples, and struggled more still to attain the true solutions for the somewhat more complex ones. Furthermore, it was outperformed by Brute search in all three domains on all complexities. The only domain where VanillaGP had a spell of superiority is the string domain. Here, it solves more cases than Brute for the first 4 complexity classes, but drops off rather quickly afterwards.

However dismal those results may have been, it was curious to see at least how close VanillaGP got to the desired targets without solving the tasks. This was not obvious in the plots of the solved tasks, since that measure only takes into account the tasks that terminated with an error of exactly 0. While this *is* the desired behavior of the algorithms, it could still be insightful to look at the improvement that VanillaGP could obtain from the input states it was given in the tasks.

Hence, a new measure of *relative improvement* was conjured up. Relative improvement is the percentage measure of how much closer the output state of the final program returned by the algorithm is to the desired output state. In other words, how does the final program compare to an empty program? It is defined as:

$$I_{rel} = \frac{initial\ error - final\ error}{initial\ error} = \frac{error(P_{empty}) - error(P_{final})}{error(P_{empty})}$$

Relative improvement is a "consolation" measure of sorts. Since with respect to the fully solved cases, the performance of VanillaGP is not on par with the benchmark set by Brute, it becomes more useful to look a broader statistic to check its performance. Such a statistic is provided by I_{rel}, as it gives an idea of whether or not any reasonable learning has occurred for a given example.

Since I_{rel} is calculated for each task individually, for visualization, an average value was taken within each complexity class of tasks. The resulting plots are shown here:

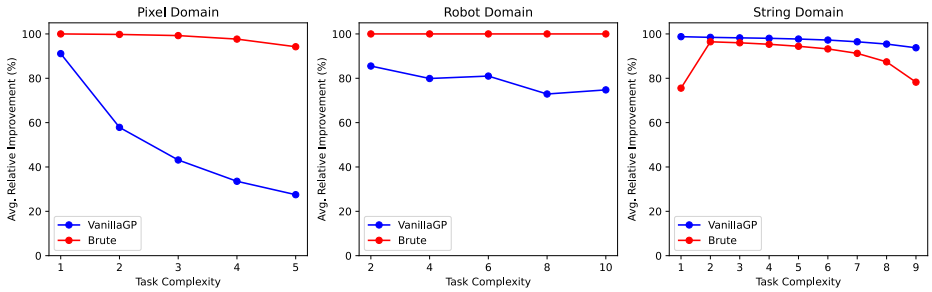


Figure 4: Average relative improvement grouped by complexity for VanillaGP.

The results of I_{rel} are somewhat more promising. Although the relative improvement wanes with increasing complexity in the pixel domain, the other two indicate better results. The plots show that for the robot domain, VanillaGP achieves around 80% improvement for all complexities. What is also evident is that Brute outperforms VanillaGP in this regard as well, with its I_{rel} at 100% for the robot and almost 100% for the pixel domain. String domain is the only one where VanillaGP does consistently better, but the discrepancy in the performance is largely inconsequential. That being said, it appears that the downward trend for Brute is somewhat steeper than that of VanillaGP, perhaps indicating a higher robusteness of VanillaGP at more complex tasks.

While it may not be trivial to explain the positive performance in the string domain, the reasonably high relative improvement of VanillaGP may stem from its tendency, or lack thereof, of becoming stuck in local extrema that Brute may find itself in. The following is a plot of the error progression throughout the iterations of the two algorithms on one individual task:

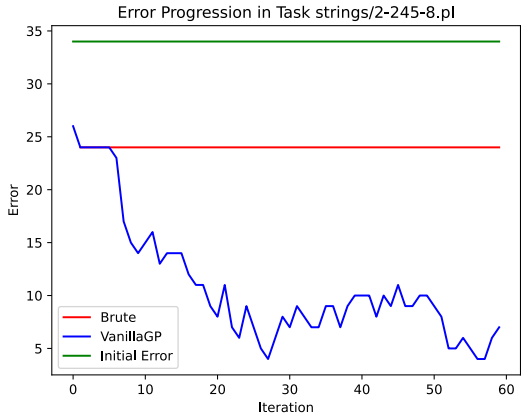


Figure 5: Change in the error of the best-found program across generations.

From here, it may be observed that while Brute seems to start showing almost monotonic behavior rather early on, VanillaGP does wiggle its way out of such states of stagnation

more successfully. Such is the design of the algorithm and its parental paradigm. It's clear that the solution VanillaGP proposes in one iteration is occasionally worse than that in the previous iteration. This jagged stochastic behavior helps it prevent lengthy stagnation. However, despite its potential to escape monotonic behavior, it is evident that the actual error with respect to the desired solution is still relatively high, with VanillaGP's final figure plateauing at about 6 error units and the overall minimal error of 3 in the entire run, which was obtained half way through the run.

Another point of interest is the generalizability of the resulting programs in the string domain. Having been trained on the training set, the algorithm produces a singular best program that it has encountered. This program is expected to be able to tackle all of the test cases as well, since the underlying rule of string transformations is the same for both sets of input-output examples.

A very simple observation that was made during the inspection of the results was the surprising number of tasks whereby the programs discovered by VanillaGP produced an infinite error when evaluated on the test set. A further analysis reveals the following figures:

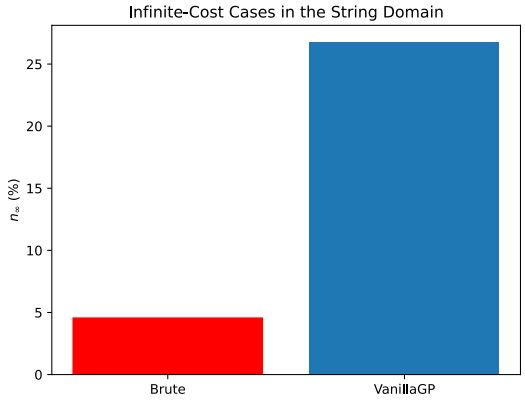


Figure 6: The portion of the tasks where test error becomes infinite.

Figure 6 provides a comparison of Brute and VanillaGP in terms of the percentage of tasks that had infinite test errors. As it turns out, VanillaGP is about 6 times as likely to produce programs that prove to be invalid when applied to test cases. This is a clear sign of the lack of generalizability of the programs found by the genetic algorithm.

The percentage of infinite-cost programs is a proxy used for the generalizability potential of a search algorithm. As mentioned in Section 3, an infinite cost implies that the program attempts invalid operations when run on a given input. However, the training error is always finite, since all inputs are valid states, and the simplest best program that the algorithm starts with is the empty program, meaning that the resultant program will be at least as good as the empty program, which is valid. So, a reasonably generalizable algorithm will attempt to minimize the number of instances where the train cost is finite, while the test cost is infinite (indicating a run-time error in the found program on valid input). Despite its

naivety, this measure n_∞ already indicates that VanillaGP fails to return robust programs in over a quarter of the cases. This is contrasted with Brute's 5% n_∞.

The first version of VanillaGP was a basic implementation of a genetic algorithm with an ordinary mutation operator and one-point crossover. However, in the course of development, it appeared that there may be a certain lack of program diversity.

In order to check this, a short statistic was observed about each generation. The standard deviation of the program length within each generation was used as a proxy for program diversity. What was noticeable was an almost nonexistent change in the lengths of the programs across generations. This was thought to be a limitation and a sign of potential stagnation early on.

Both n-point crossover and UMAD are attempts at diversifying the population. These versions of the genetic operators have the effect of disturbing the structure of individual programs and so exploring more varied regions of the search space of programs. However, this property may not always be of value. In the case where a program in the generation is particularly close to the solution, it has a higher likelihood of being destroyed in the name of diversity, since neither operation (in their current implementations, at least) take the fitness of programs into account. As such, on the occasions where a program needs to be honed, these operators may be doing more harm than good.

6 Responsible Research

An integral part of academic research are the ethical considerations and implications of research. As such, this section of the paper is dedicated to the discussion of the aspects of this research project relevant to maintaining academic integrity.

Firstly, an important aspect of research is reproducibility and the availability of data. To this end, a joint decision amongst the peer group has been made to make the repository containing both the novel code and the data fed into the algorithms publicly available. In the same way as the previous development in Inductive Program Synthesis conducted by the peer group's supervisor, as well as the data used for experiments, was kindly provided to the members of the peer group for use and reference, this paper and all the information associated with it is up for public viewing.

Another significant academic value is the impartiality toward the obtained results. That is, it is crucial for individual researchers to provide and for institutions to at least not disincentivize negative results in research work. Negative results, while not immediately impressive, allow for a crucial step of inquiry – shrinking the search space. Elimination of certain types of methods, techniques, causal factors, etc. can indeed prove to reduce the space of parameters that upcoming researchers in the field would have to look into.

As was shown above, the performance of the algorithm developed in this paper has fallen short of expectations and did not appear to explicitly support the initial hypotheses. However, in order to maintain integrity, and in spite of the minor positive aspects of the algorithm's performance, all of the encountered flaws or potential shortcomings have been

hereby addressed.

7 Conclusions and Future Work

7.1 Conclusion

This research paper provides the design and analysis of a novel Genetic Algorithm on the problem of Inductive Program Synthesis (IPS). VanillaGP is a classical genetic algorithm applied to the domain of IPS as a potential improvement on or alternative to the deterministic Brute algorithm that was recently shown to be particularly effective at generating programs, albeit in a restricted Domain Specific Language (DSL).

VanillaGP, as any Genetic Algorithm, works by creating and breeding generations of programs in search of a program meeting the pre-defined condition(s). The programs themselves are defined as varied-length linear sequences of token functions. This means that the algorithm essentially finds itself somewhere between the traditional Genetic Algorithms and the specific branch of them, called Genetic Programming.

The experiments that were conducted as part of this research project have demonstrated the comparatively poor performance of VanillaGP with regards to generating complete solutions to tasks. It is noticeably outperformed by Brute in all domains but one. VanillaGP's performance on the string domain seems to start off higher, but wanes as the complexity of the string-based tasks it faces increases. However, VanillaGP does appear to provide reasonable relative improvement in the robot and string domains, which is close to that of Brute.

All in all, while VanillaGP cannot be definitively proposed as a remedy to the world's program-synthesis-related woes, certain properties of it, as a Genetic Algorithm, can be exploited under specific circumstances. That is, as any other piece of technology, it should be used sparingly and in the suitable contexts, the full nature of which is yet to be determined.

7.2 Limitations and Future Research

One limitation of the current approach is the lack of dynamic generation of control tokens. That is, the only if- and loop- statements that the search algorithm uses are the ones generated beforehand during the invention stage. While it is acceptable for most small/simple problems, problems that may require nested if-statements, if-checks within loops, or even arbitrarily long bodies of these control tokens, remain beyond the reach of the search algorithm.

In traditional GP, this is less of an issue, since the initial population of programs is generated as a collection of trees node-by-node. Furthermore, the crossovers that are performed can interchange any two subtrees of two programs, meaning that arbitrary control tokens are potentially possible.

While incorporating similar flexibility may be challenging, Koza provides another way of diversifying resulting programs. Automatically Defined Functions (ADFs) are, as the name suggests, sub-programs (or functions) that the search algorithm generates automatically

during the search [4]. This requires programs to have several ADF branches and one result-producing branch. The result-producing branch is the sub-program that may use (call) the ADFs in order to solve the problem at hand more efficiently.

This extension to the current algorithm is more attainable, since one could simply emulate different ADFs by storing a number of program lists alongside the final program. Those program lists will then be treated the same way as ordinary programs, but the result-producing program would be granted the ability to invoke those auxiliary programs. However, the process of crossovers and the decision of crossover points between programs may become more complex.

A more maverick approach would be a hybrid search algorithm that combines the strengths of a deterministic Brute and a stochastic Genetic Algorithm. It is possible to envisage such a hybrid system as a GA doing the exploration and sampling of the search space and then delegating the honing job to Brute. That is, in each generation, the GA could transfer a subset of the population that is particularly performant to Brute. Brute will, in turn, iteratively try to improve those programs by appending new functions to them. Its high speed means that Brute can potentially be invoked multiple times throughout the run of a GA, without significant overhead. Whilst having its flaws (such as the assumption that performant programs ought to be better when enriched with functions), a hybrid approach may, if implemented, prove to be a reasonable reconciliation of stochastic and deterministic search algorithms.

References

[1] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, page 14â21, USA, 1987. L. Erlbaum Associates Inc.

[2] Andrew Cropper and Sebastijan Dumancic. Learning large logic programs by going beyond entailment. *CoRR*, abs/2004.09855, 2020.

[3] 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*, GECCO '18, page 1127â1134, New York, NY, USA, 2018. Association for Computing Machinery.

[4] John R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, Jun 1994.

[5] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *SIGPLAN Not.*, 51(6):522–538, June 2016.

[6] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 452–466, New York, NY, USA, 2017. ACM.

[7] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, Jun 1994.