Genetic Algorithms for Inductive Program Synthesis

M.R. Tromp
Supervisor: S. Dumančić
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

VanillaGP is an Inductive Program Synthesis algorithm that takes a Genetic Algorithm (GA) approach by using its 3 components: selection, mutation, and crossover. Many different alternatives exist for these components and although this is not the only application of a GAs on the Program Synthesis domain, it has not been extensively evaluated what the effects of using these different alternatives or combinations of them are. We explored this by evaluating the performance of multiple alternatives per component and comparing the results of combinations of these alternatives to VanillaGP. These evaluations were done on 3 IPS domains: robot, ASCII art, and strings. From these evaluations we conclude that Stochastic Universal Sampling combined with Queen Bee Crossover and an altered version of One Mutation Per Solution performs best on the string domain, and Down-Sampled Lexicase Selection combined with Three Parent Crossover and the same altered version of OMPS performs best on the other domains.

1 Introduction

The automatic generation of programs is called Program Synthesis (PS). PS as been seen as the holy grail of Computer science since the creation of Artificial Intelligence [Gulwani *et al.*, 2017]. It has for example been used to create functional recursive functions that process algebraic datatypes [Osera and Zdancewic, 2015], to assist developers by synthesising code snippets [Ferdowsifard *et al.*, 2021], and to deobfuscate malware [Jha *et al.*, 2010]. One of many types of PS is Inductive Program Synthesis (IPS), for which the synthesis algorithm is given a set of inputs and corresponding outputs, and creates a program based on that set [Solar-Lezama, 2018].

To create a program, IPS algorithms search a tree made up out of all possible programs. Because of the sheer size of these trees it is not possible to search the entire tree in a feasible time-frame. This means that searching efficiently is an integral part of IPS algorithms. There are many different ways to search efficiently, one of which is using Genetic Algorithms (GAs).

VanillaGP is an IPS algorithm that takes a Genetic Algorithm approach [Azimzade and Dumančić, 2022]. It consists of 3 components: selection, crossover, and mutation. As mentioned in [Azimzade and Dumančić, 2022], VanillaGP was created to try and overcome an algorithm named Brute's tendency to get stuck in local optima.

Brute [Cropper and Dumančić, 2020] is a best-first search program synthesis algorithm. However, as described in [Azimzade and Dumančić, 2022], VanillaGP only outperforms Brute on one out of 3 test domains, and for that 1 test domain the performance still leaves a lot to be desired. Therefore, it is important to explore if there are alternatives for the current implementations of VanillaGP's components that could improve its performance. This importance is strengthened by the fact that there has not been a lot of research to the effects of different alternatives and their combinations on the performance of GAs in a Program Synthesis domain.

Therefore, the following research question is used: Are there alternatives for the components of VanillaGP that will allow it to solve a higher percentage of tasks within the given domains during the same time frame?

The research question can be divided into the following sub-questions:

- Q1/Q2/Q3 - Are there alternatives for the selection/crossover/mutation component of VanillaGP that will allow it to solve a higher percentage of tasks within the given domains during the same time frame?
- Q4 - Is there a combination of found alternatives for the components of VanillaGP that will allow it to solve a higher percentage of tasks within the given domains during the same time frame?

Ultimately this project explores whether or not it is possible for an altered VanillaGP algorithm to solve a higher percentage of tasks than the unaltered VanillaGP algorithm and the Brute algorithm. This is done by evaluating alternatives to the components by themselves. However, it is also important to evaluate combinations of alternatives to different components, because a single alternative might be more or less effective when it is combined with other alternatives. A combination of alternatives can improve performance, even if the alternatives do not improve performance on their own.

The structure of this paper is as follows. Section 2 discusses the background to this research, with Section 3 discussing related work to this research. Then, in Section 4 the different alternatives that were chosen for each of the components are introduced and explained. The experiments and their results follow in Section 5. In Section 6 conclusions are drawn and possible future work is mentioned. Section 7 discusses the ethical aspects of this research.

2 Background

Inductive Program Synthesis

Inductive Program Synthesis (IPS) is a type of program synthesis. An IPS algorithm takes a set of inputs and corresponding outputs, and creates a program based on that set [Solar-Lezama, 2018]. The program that the IPS algorithm produces should produce the corresponding output for any given input.

At the heart of IPS is search. To create a program an IPS algorithm searches a tree made up out of all possible programs. If we were to search these trees entirely, we would not always get a program within a feasible timeframe. Therefore, we need a better way to search. There are many different better ways to search these trees, including best-first search, Monte Carlo Tree Search, Metropolis-Hastings algorithm, Large Neighbourhood Search, and Genetic Algorithms.

Although there are many different algorithms, they have one thing in common. The balancing of exploration vs exploitation. In essence this can be translated to the following question. Given that we have a program that is almost able to solve a task. Then, do we search somewhere else in the tree to find a program that can solve the task completely, or do we try and adapt that program to completely solve the task?

Balancing exploration and exploitation is crucial to finding a correct program within a feasible timeframe.

Genetic Algorithms

Genetic Algorithms (GAs) apply ideas from biology to programming. In [Mitchell, 1996] the author describes that GAs consist of a population of chromosomes that are evolved to create a new population. They state that the chromosomes are made up out of genes. Each of these genes is an instance of an allele, with the set of alleles being all possible values that a gene can take on, the author describes.

In practice this means that GAs work in iterations. Each iteration creates a generation. The first generation is randomly generated. After that, this generation is put through an algorithm that consists of 3 components: selection, crossover, and mutation. Crossover and mutation are not mandatory.

The selection component selects parents for the new generation from the previous generation based on their fitness, which is calculated by using the performance of the program on the input training task(s). Programs from the previous generation are chosen 0 or more times, resulting in a new generation of the same size that consists of parents that are more fit than the previous generation.

In the crossover component the genes of 2 or more parents are recombined to create 1 or more children. In some crossover algorithms parents may be used multiple times to create children, always resulting in a new generation of the same size as the previous generation.

The mutation component takes each child separately and mutates them by adding, deleting, or changing one or more genes of a program to a different allele. After the mutation is completed, the Genetic Algorithm moves on to the next iteration, with this generation as input.

3 Related Work

Brute

Brute takes a best-first search approach to Program Synthesis [Cropper and Dumančić, 2020]. Although Brute performs well on the given domains, it sometimes gets stuck in local optima, as described by the authors. The domains that it was evaluated on are the same as the domains that VanillaGP and this research is evaluated on. This means that we can easily compare the results and draw conclusions from that.

VanillaGP

This research builds upon VanillaGP [Azimzade and Dumančić, 2022]. VanillaGP takes a Genetic Algorithm approach to Inductive Program Synthesis (IPS), as described by the authors. They state that VanillaGP was created to overcome Brute's tendency to sometimes get stuck in local optima and mention that it only outperforms Brute on 1 domain.

The selection algorithm of VanillaGP is Stochastic Universal Sampling (SUS). SUS is a variant of Roulette Wheel Selection [Jebari and Madiafi, 2013]. The N different programs are put on a wheel with section size relative to their fitness over all training tasks. Then, a random number is used to pick N equidistant points on that wheel, selecting the programs at those points. A point of note is that in the VanillaGP implementation SUS was incorrect, causing us to not be able to draw any definitive conclusions about the performance of VanillaGP based on the results presented in the paper.

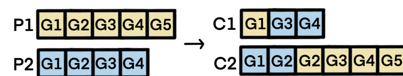

Figure 1: An example of One-Point crossover.

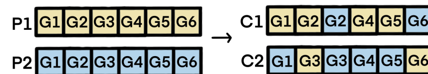

Figure 2: An example of N-Point crossover with N = 3.

There are 2 crossover algorithms implemented in VanillaGP. One-Point crossover, shown in Figure 1, randomly picks one point on each parent and swaps the parts after these points. N-Point crossover picks a random number between 1 and the length of the shortest parent divided by 2 and swaps every other section, as shown in Figure 2.

There are also 2 mutation algorithms implemented in VanillaGP. Classical Mutation is shown in Figure 3. It rolls a die for each gene of every program and uses a mutation chance to decide if a gene will be mutated. Uniform Mutation by Addition and Deletion (UMAD), shown in Figure 4, first iterates through the program while randomly adding genes. Then it iterates through the program again while randomly deleting genes.

Components

Genetic Algorithms (GAs) and the components of GAs in particular have been researched extensively. Many of these alternatives take a more basic alternative and build upon that alternative.

Selection Component

The most relevant benchmarking of alternatives to the selection component is summarized in [Helmuth and Abdelhady, 2020]. The authors describe the performance of 21 different Selection alternatives in the program synthesis domain, and are the first to do so extensively. Their findings show that Lexicase based selection methods outperform the other evaluated alternatives.

Both [Jebari and Madiafi, 2013] and [Kaya, 2011] describe existing alternatives and propose a new one. They benchmark these, but not in the Program Synthesis domain. The newly proposed methods are Combined Selection and Back Controlled Selection Operator respectively.

In [Ferguson *et al.*, 2020] Lexicase selection and 2 Lexicase selection variations are described and their performance is analysed when using random subsampling. The authors show that while these two subsampling variants reduce the computational load, it comes at the cost of some specialist individuals.

Crossover Component

[Kora and Yadlapalli, 2017], [Pavai and Geetha, 2016], and [Umbarkar and Sheth, 2015] all describe different crossover

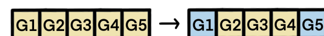

Figure 3: An example of Classical Mutation.

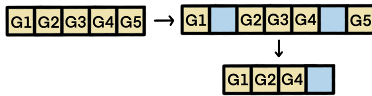

Figure 4: Uniform Mutation by Addition and Deletion.

alternatives, but do not benchmark them. Short descriptions of many different crossover operators are also given in [S. Mooi *et al.*, 2017]. They mention how the combination of crossover and mutation influences the predicament of exploration vs exploitation.

The only research that we mention that actually evaluates crossover methods is [Spears and Anand, 1992]. They evaluate 3 different crossover methods: uniform, one-point, and two-point crossover. The methods are benchmarked on a neural network problem. In this benchmarking uniform crossover outperforms all others, and not using crossover is outperformed by all 3 types of crossover methods.

Mutation Component

[Deb and Deb, 2014] describes 5 mutation methods for real-parameter genetic algorithms. The authors benchmark the methods on multiple different problems. They describe that all mutation methods outperform not using mutation at all.

[S. Mooi *et al.*, 2017] and [Soni and Kumar, 2014] both describe many different mutation methods, but do not benchmark them. The former also states how crossover and mutation influence the final outcome of the Genetic Algorithm.

Components in General

Although there are many reviews available for each component separately, to our knowledge there are not any resources that explore the combination of alternatives for different components. Our research evaluates component alternatives on their own, as well as in a combined setting.

Most of the research around the alternatives for GAs that does exist is not benchmarked within the field of Program Synthesis. This means that the results do not necessarily translate to our domain. To our knowledge there are not many other studies that evaluate the performance of alternatives to the components on the Program Synthesis domain.

The research on the performance of the alternatives in the Program Synthesis domains is also not directly applicable to our research, because the used domains are not the same as the domains we use in this research. While the reviews of the alternatives are useful as a single source of many different alternatives and to get a general idea of the performance of the alternatives, the results of alternatives to the components of GAs are domain specific and we therefore cannot draw any direct conclusions from the provided benchmarkings.

4 Methodology

To explore if there are alternatives to the components of VanillaGP that will allow it to solve a higher percentage of tasks within the given domains during the same time frame, we started with refactoring the provided codebase. This codebase includes implementations of both Brute and VanillaGP, as well as other algorithms.

VanillaGP does not perform as well as Brute on both the ASCII art domain and the robot domain. On the string domain it does solve a higher percentage of tasks than Brute on more complex tasks. This comes with the caveat that the implementation of Stochastic Universal Sampling (SUS) was incorrect. Therefore, we implemented a new version.

There are many different alternatives to each component of a Genetic Algorithm. Because Genetic Algorithms usually mostly perform well when fine-tuned to a specific problem, it is impossible to know which alternatives will perform well on our domains. Therefore, we implement many different alternatives that widely different approaches, favouring more widely known alternatives and ones of our own design.

The paper [Azimzade and Dumančić, 2022] describes that when N-Point crossover and Uniform Mutation by Addition and Deletion (UMAD) are not used, the programs that are produced by VanillaGP lack diversity. This could be caused by the incorrect implementation of the selection algorithm, which selects the same program 99% of the time.

The main goals we tried to achieve with the finding of the alternatives for the components are diversifying the programs that the algorithm produces without affecting the convergence to a correct program, and speeding up the runtime of the algorithm. A faster runtime allows for more generations, and thus possibly a better performing program.

Selection component

One of the alternatives we explored is Roulette Wheel Selection (RWS). In essence RWS puts N programs on a roulette wheel with their size on that wheel proportionate to their relative fitness and spins the wheel N times. RWS could cause an algorithm to converge to a local optima, which would normally be a disadvantage [Jebari and Madiafi, 2013]. However, because of possible lack of convergence in alternatives to the other components this could be an advantage.

Another alternative we explored is Lexicase selection (LS). Each time a parent program has to be chosen LS randomizes the order of the training tasks [Ferguson *et al.*, 2020]. Then, as the authors state, it iterates through the training tasks only keeping the program that have the highest fitness until it either has one program left or it has no more training tasks in which case it randomly picks a program. They also note that LS allows for specialized programs to be chosen, instead of only choosing programs that perform well on average over all training tasks. The string domain is a complicated domain that needs highly specialized programs to perform well. LS performs better than Tournament Selection and RWS [Helmuth and Abdelhady, 2020].

However, LS is computationally heavy [Ferguson *et al.*, 2020]. Therefore we implemented Down-Sampled Lexicase Selection (DSLS). We only consider the first 5 training tasks after randomization which differs from the implementation described in [Ferguson *et al.*, 2020], where they select a random subset of the training tasks by using a 'down-sample factor' and only randomize those tasks each iteration. We made this choice to allow for more specialized programs being chosen. [Helmuth and Abdelhady, 2020] states that DSLS has an average performance on easier problems, but also mentions that it performs better on the most complicated problems making it interesting in our evaluation.

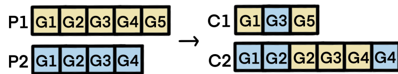

Figure 5: An example of Two-Point crossover.

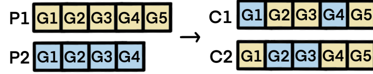

Figure 6: An example of Uniform Crossover.

We also implemented a LS variant of our own design, which we call Combined Lexicase Selection (CLS). If there are less than than 5 training tasks available to the Genetic Algorithm, SUS is used. Otherwise it uses LS. We implemented this, because for LS to work well it needs at least a hand full of training tasks. If we have 4 training tasks, we have at most 4! = 24 different orderings, which is a small amount compared to the generation size of 200. When we have at least 5 training tasks, this gives us at least 5! = 120 orderings.

Another alternative is Tournament Selection (TS), which is a variant of Rank-Based selection. For each program that has to be selected it randomly selects a set of k programs, ranks them based on their fitness, and selects the most fit program. [Jebari and Madiafi, 2013]. We implemented this with k = 5 and chose it as an alternative because it was shown in [Helmuth and Abdelhady, 2020] that TS performs better than RWS and therefore might perform better than SUS.

We also implemented Truncation Selection (TRS). For TRS we only select a percentage p of the fittest programs [Jebari and Madiafi, 2013], where in our implementation p equals 25%. To counteract the population now being smaller, we repeat this 4 times. We had not yet used any variants of this method, making it a good one to evaluate.

Crossover component

A crossover method that is in between One-Point crossover and N-Point crossover is Two-Point crossover. In Two-Point crossover, shown in Figure 5, both parents are cut in 3 parts swapping the middle parts. [Pavai and Geetha, 2016]. This could allow programs to find a shorter path to the actual goal or to find a correct path when the correct start and finish parts were already found.

To introduce a more blended mix of the parents we implemented Uniform Crossover (UC), shown in Figure 6. UC combines the parent's genes by taking each gene from one of the parents in a uniform way [Umbarkar and Sheth, 2015] [Kora and Yadlapalli, 2017]. To allow for offspring of variable length, we apply UC up until and including the length of the shortest parent and then copy the rest of the longest parent up until the length of the offspring.

A different approach to crossover is Queen Bee Crossover (QBC). In QBC one program is appointed as the Queen Bee and then breeds with all other programs using a another crossover method [Pavai and Geetha, 2016]. This could be a crossover method that promotes diversity, while also being able to converge to an optimal solution when the Queen Bee is the program with the highest fitness.

We also implemented Three Parent Crossover (TPC) [Kora and Yadlapalli, 2017], Figure 7, which selects 3 parent programs and compares each gene in the parent programs. If the gene in the first two parent programs is similar we choose that gene, otherwise we choose the gene on the third parent, the authors describe. Because it is difficult to decide if the genes are similar, we check for equality instead of similarity. If only 2 parents are long enough for a certain index we randomize which parent's gene the child inherits and we copy the final parent if only 1 is long enough. This method is differs greatly from the previously used methods, thus giving us a bigger variety of alternatives.

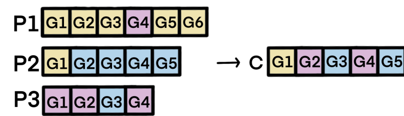

Figure 7: An example of Three Parent Crossover.

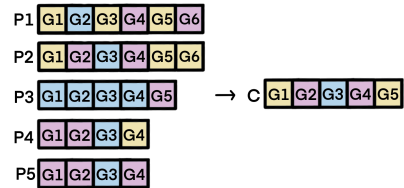

Figure 8: An example of Multiple Parent Crossover with N = 5.

Another alternative that we chose is Multiple Parent Crossover (MPX), shown in Figure 8. MPX chooses 5 parents, counts the occurrences of the different alleles for each gene, and then selects the allele that appears most frequent for each gene of the child [Pavai and Geetha, 2016]. We chose this alternative, because [Pavai and Geetha, 2016] states that multi-parent crossover operators can speed up the run-time of a Genetic Algorithm.

We also implemented Random Crossover [Pavai and Geetha, 2016], shown in Figure 9. The authors state that this method works well for variable length individuals which is the case with our programs. They describe the algorithm to allow for overhang on each side of the two parents, allowing for an entire program to not be considered.

Mutation component

One problem that [Azimzade and Dumančić, 2022] mentioned when using Classical Mutation was that there was not enough variation between programs. However, they also mentioned that UMAD might break working solutions in favour of exploration. Therefore, an alternative that is viable to explore is using a One Mutation Per Solution (OMPS) alternative. OMPS is shown in Figure 10 and does as it states: it always mutates one gene per program [Deb and Deb, 2014].

To allow for more variance in program length, we also implement a slightly altered version of OMPS, which is shown in Figure 11. In this Altered One Mutation Per Solution (AOMPS), it can also add a gene in front of the program or after the program.

We also added Interchanging Mutation (IM), shown in Figure 12. IM takes two random genes in a program and switches

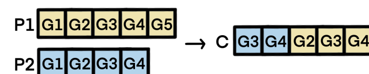

Figure 9: An example of Random Crossover.

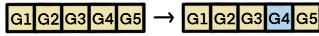

Figure 10: An example of One Mutation Per Solution.

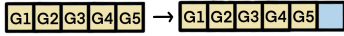

Figure 11: An example of Altered One Mutation Per Solution.

them [Soni and Kumar, 2014]. To allow for more different programs, and to allow this mutation method to work on any program, we allow for the two indices to be equal. This method could help a program that is very close to a solution get to the solution.

To create even more variety in the mutation methods, and therefore more variety in the programs we implemented Scramble Mutation (SM), shown in Figure 13, and Reversing Mutation (RM), shown in Figure 14. Both take 2 random indices in a program. SM randomly shuffles the genes at and between these indices [Soni and Kumar, 2014], and RM reverses their order [Soni and Kumar, 2014]. Just like for IM, we allow the two indices to be equal.

5 Experiments and Results

Experimental Setup

To evaluate the alternatives to the components accurately they are all benchmarked using the same procedure as Brute [Cropper and Dumančić, 2020], which is included in the provided codebase. The actual experiments were ran on the DelftBlue supercomputer [Delft High Performance Computing Centre (DHPC), 2022].

The 3 domains that the alternatives are benchmarked on are: robot, ASCII art, and strings. For each domain programs are made up of token functions of the types transition and Boolean and 2 shared functions which cannot be nested: an if-then-else and a while loop. The set of the alleles is equal to the set of all possible functions for that domain. The tasks are grouped by their complexity per domain, with the complexity being a measure of how difficult it is to solve the task.

The robot domain tasks have the challenge of moving a robot that has to pick up a ball and drop it off at the goal location over a grid, as shown in Figure 15. The transition functions are *MoveUp*, *MoveDown*, *MoveLeft*, *MoveRight*, *Grab*, and *Drop*. The boolean functions are *AtLeft*, *NotAtLeft*, *AtRight*, *NotAtRight*, *AtTop*, *NotAtTop*, *AtBottom*, and *NotAtBottom*.

The ASCII art domain tasks have the challenge of drawing a pixel representation of an ASCII string input on an empty grid by manipulating the cursor, as shown in Figure 16. The transition functions are *MoveLeft*, *MoveRight*, *MoveUp*, *MoveDown*, and *Draw*. The boolean functions are: *AtLeft*, *NotAtLeft*, *AtRight*, *NotAtRight*, *AtTop*, *NotAtTop*, *AtBottom*, and *NotAtBottom*.

The string domain tasks have the challenge of transforming an input string to a corresponding output string, as shown in Figure 17. This domain is the most difficult and is therefore the only domain that has training tasks. The amount of training tasks depends on the task complexity, with more complex tasks having more training tasks. The transition functions are: *MoveLeft*, *MoveRight*, *MakeUpperCase*, *MakeLowercase*, and *Drop*. The boolean functions are: *AtStart*, *AtEnd*, *IsLetter*, *IsNumber*, *IsSpace*, *IsUppercase*, *IsLowercase*, and their negations.

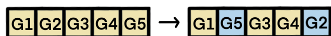

Figure 12: An example of Interchanging Mutation.

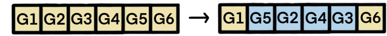

Figure 13: An example of Scramble Mutation.

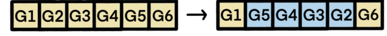

Figure 14: An example of Reversing Mutation.

For each experiment the maximum time per task is 60 seconds, the maximum token function depth is 5, and the maximum number of generations is 200, with a population size of 200. These, and any other settings that are not mentioned are the original settings for VanillaGP.

To be able to compare the results of the experiments, each experiment uses Stochastic Universal Sampling (SUS), One-point crossover, and Classical mutation as base components. The amount of experiments caused us to only be able to run each alternative and combination once. Because of stochastic nature of the Genetic Algorithm the received results will not be exactly the same every time an experiment is done.

Results and Discussion

The performance of the alternatives to the components and their combinations on the ASCII art domain was incredibly poor, with all alternatives solving less than 5% of tasks of complexity 2 and 0% of tasks of a higher complexity. For the combinations the solved percentages are not significantly higher. Therefore, we do not show the performances for the ASCII art domain.

Selection Component

Figure 18 shows the percentage of solved tasks for each selection alternative. Although previous research showed Lexicase (LS) based variants to outperform others, Tournament Selection (TS) outperforms the other alternatives on our domains. To see why, we look at the reasons for why the algorithms produced incorrect programs.

There are 4 possible reasons for a program produced by the algorithm being incorrect: a time-out; the maximum amount of iterations (generations) is reached; the train-cost is 0 but the test-cost is not, meaning that the produced program solves the training tasks but does not solve the test tasks; and the test-cost is infinite, meaning that the produced program is invalid when used for the test tasks. For some failed tasks multiple of these are true. Therefore, for the infinite test-cost we only consider cases for which the train-cost is 0. For the train-cost we only consider the cases when there no other possible reasons apply.

Table 1 shows what percentage of incorrect programs had what reason. From this Table, we can conclude that the LS

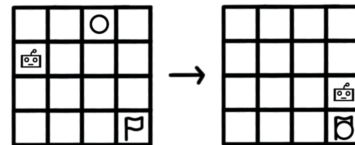

Figure 15: An input-output example for the robot domain.

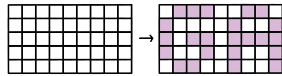

Figure 16: An example for the ASCII art domain.

"Genetic Algorithm" ⟶ "GA"

Figure 17: An input-output example for the string domain.

based methods are both too slow and have too little iterations, while the other selection methods mainly have too little iterations.

When looking at Table 1 it is clear that quite a high percentage of programs are incorrect because their test-cost is infinite. To see why, we look at test-task 1-3-9 from strings for TS. The input is a name in lower case, and the output is the first letter of that name capitalised. The training task is "laura", which results in "L". The test-tasks are 9 names of different lengths with the corresponding outputs. The problem arises when we encounter a test-task of length 4, "paul", as shown in Figure 19. Because the program only uses if-statements, it only works for strings with length 5. When we look at other programs that are incorrect because of infinite test-cost or their train-cost being 0 but their test-cost not we can usually see a similar problem. The fact that not using any loops produces many incorrect programs is consistent with our intuition for solving a problem like test-task 1-3-9, which would result in a program like the following: *[[MakeUppercase], LoopWhile(NotAtEnd [Drop]), If(NotAtStart [Drop] [MakeUppercase])]*. In fact, we cannot create a program that is able to transform a string of any length without using a loop. Combining these facts we hypothesize that loops are crucial for the performance of a program.

Take one of the worst performing alternatives, LS. Only 52% of the programs that it produces for the string domain contain loops. For the best performing alternative, TS, this is 67% and for the TRS this is 74%. So, again there seems to be a correlation between performance of the alternative and the percentage of produced programs that contain loops.

When comparing the average test-cost of incorrect non infinite tasks. There does not seem to be a correlation between between the percentage of solved tasks and the average test-cost. While Truncation (TRS) selection is slightly outperformed by TS, the average test-cost of TRS is lower than that of TS. This means that TRS could possibly outperform TS.

Crossover Component

In Figure 20 the percentages of solved tasks for all mutation alternatives are shown. From this Figure, we can conclude

Selection Alternative	Robot Time-out	Robot Iterations	String Test-cost	String Train-cost	String Time-out	String Iterations
Original SUS	60.3%	39.7%	6.7%	5.7%	85.6%	2.1%
SUS	0.2%	99.8%	8.3%	15.6%	0.1%	76.1%
RWS	0.2%	99.8%	7.7%	13.4%	0.1%	78.8%
LS	42.9%	57.3%	8.1%	10.0%	80.9%	1.0%
DSLS	50.4%	49.6%	8.2%	10.2%	80.7%	0.9%
CLS	0.0%	100.0%	8.6%	13.3%	48.7%	29.4%
TS	0.0%	100.0%	13.1%	21.9%	4.7%	60.4%
TRS	6.7%	94.3%	14.2%	21.9%	16.7%	47.8%

Table 1: Percentages of reason for failed task. The percentages are rounded to the nearest single decimal, or to 0.1 if between 0.0 and 0.1.

	laura	paul
If(IsNotSpace [[MakeLowercase, MakeUppercase]] [[MoveRight, MakeLowercase]]),	**L**aura	**P**aul
If(IsNumber [Drop] [[MoveRight, Drop]]),	L**u**ra	P**u**l
If(IsLetter [[MakeUppercase, MakeLowercase]] [[Drop, MoveLeft]]),	L**u**ra	P**u**l
If(IsLowercase [[MakeUppercase, Drop]] [[MoveRight, MakeUppercase]]),	L**r**a	P**l**
If(AtEnd [[Drop, MakeLowercase]] [[MakeLowercase, Drop]]),	L**a**	**p**
If(AtStart [[MoveRight, MoveLeft]] [Drop])	**L**	Invalid

Figure 19: The program produced by TS for test-task 1-3-9, based on train-tasks 1-3-9. The currently selected letter is in bold.

that Queen Bee Crossover (QBC) is the best performing alternative, while both the Three Parent (TPC) and Multiple Parent (MPX) crossover are the worst performing. Again, to see why we look at the reasons for why the algorithms got the programs incorrect.

The percentages of reasons for incorrect programs can be seen in Table 2, which shows that for all alternatives except for Queen Bee, their bottleneck is the amount of iterations. There is one reason for failed task that stands out. This is the infinite test-cost. For both TPC and MPX this is far lower than the other alternatives.

When we look at the final programs produced by the alternatives, we can see the same problem as for the selection component. However, the percentage of programs that contains loops shows the opposite. For QBC only 59% of programs contain a loop, while for MPX this is 73%. Looking more in depth at which programs contain these loops, it becomes clear why QBC outperforms MPX. For MPX, 68% of all programs that contain a loop is incorrect, with 66% of all programs with loops reaching a time-out or the max iterations. For QBC these numbers are 43% and 29% respectively. If we ignore programs that were incorrect because of a time-out or the max iterations being reached only 7% of programs produced by MPX contain loops, while for QBC this is 30%. We can do this because it is not certain that these programs would contain loops if the time-out or max iterations had not been reached. These results are in line with the results we saw before and if we adjust the discussed alternatives for the selection component the same way this also holds.

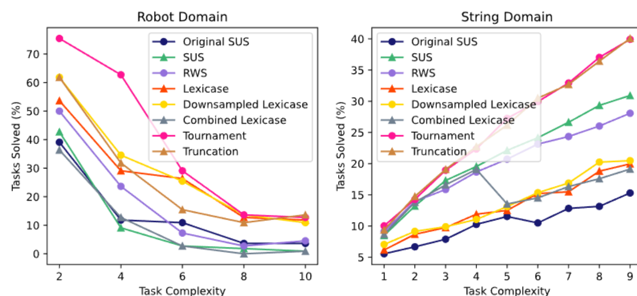

Figure 18: Percentages of solved tasks for all selection alternatives.

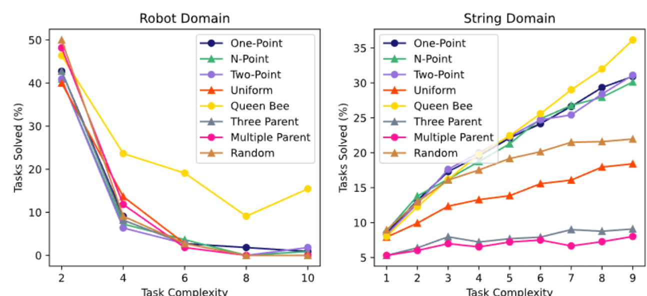

Figure 20: Percentages of solved tasks for all crossover alternatives.

Crossover Alternative	Robot Time-out	Robot Iterations	String Test-cost	String Train-cost	String Time-out	String Iterations
One-Point	0.2%	99.8%	8.3%	15.6%	0.1%	76.1%
N-Point	0.0%	100.0%	8.0%	16.2%	0.5%	75.3%
Two-Point	0.2%	99.8%	7.7%	15.1%	0.5%	76.7%
Uniform	0.0%	100.0%	3.5%	7.9%	0.0%	88.6%
QBC	28.2%	72.0%	15.6%	18.4%	41.0%	25.6%
TPC	0.0%	100.0%	1.6%	3.5%	0.0%	94.9%
MPX	0.4%	99.6%	1.6%	3.2%	1.4%	94.0%
Random	0.0%	100.0%	3.5%	9.7%	0.0%	86.7%

Table 2: Percentages of reason for failed task. The percentages are rounded to the nearest single decimal, or to 0.1 if between 0.0 and 0.1.

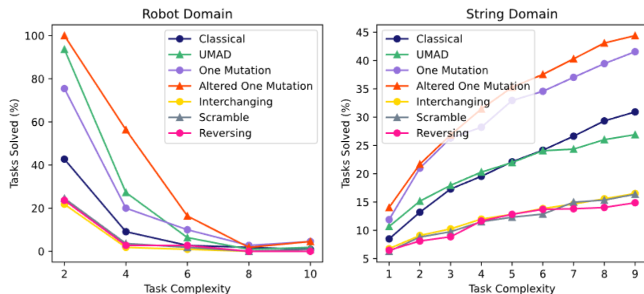

Figure 21: Percentages of solved tasks for all mutation alternatives.

Mutation Component

The percentage of solved tasks for each mutation alternative is shown in Figure 21. There are 3 groups of alternatives that perform similarly on the string domain. Out of all alternatives, Altered One Mutation Per Solution (AOMPS) performs the best, and Interchanging (IM), Scramble (SM), and Reversing (RM) mutation perform the worst.

Looking at Table 3 we can see that while for all alternatives the number of iterations are the bottleneck, the percentage of time-outs is significantly larger for the alternatives that perform worse. We can also see that like for the selection and crossover components, the best performing alternatives have the highest percentage of infinite cost test-tasks.

When we look at the final programs that the alternatives produce we can see why. Like for the selection alternatives, the higher performing alternatives have a higher percentage of programs with loops. For AOMPS this is 83%, while for RM this is only 42%. If we adjust these percentages to not include programs that reach the time-out or the max iterations, these percentages become 45% and 10%, thus following the same trend as the other components.

Combinations

Because of the sheer amount of possible combinations, we were not able to evaluate every single combination of alternatives. Based on the results from the components separately

Mutation Alternative	Robot Time-out	Robot Iterations	String Test-cost	String Train-cost	String Time-out	String Iterations
Classical	0.2%	99.8%	8.3%	15.6%	0.1%	76.1%
UMAD	0.0%	100.0%	5.9%	11.0%	0.0%	83.1%
OMPS	0.5%	99.5%	10.3%	19.0%	6.7%	64.2%
AOMPS	0.0%	100.0%	11.2%	21.3%	6.1%	61.5%
IM	2.7%	97.3%	5.2%	9.2%	19.5%	66.1%
SM	3.3%	96.7%	5.0%	8.9%	18.2%	68.0%
RM	1.2%	98.8%	4.8%	8.4%	16.9%	70.1%

Table 3: Percentages of reason for failed task. The percentages are rounded to the nearest single decimal, or to 0.1 if between 0.0 and 0.1.

	Selection	Crossover	Mutation	Reason
1	SUS	One-Point	Classical	Intended implementation for VanillaGP.
2	SUS	N-Point	UMAD	Intended alternate implementation for VanillaGP.
3	TS	QBC	AOMPS	Best performing alternatives for components.
4	DSLS	TPC	UMAD	DSLS performs slightly better than LS, but time-out is a bottleneck. Therefore, we combined it with crossover and mutation methods that have iterations as bottleneck.
5	TRS	QBC	AOMPS	Lowest average test-cost for components.
6	RWS	TPC	RM	Lowest percentage of infinite test-cost for components.
7	SUS	QBC	UMAD	SUS and UMAD have iterations as bottleneck, while QBC has time as bottleneck.
8	CLS	QBC	IM	Smallest difference between percentage of time-outs and percentage of iterations on string domain. Possibly a good balance between the two.
9	TS	QBC	OMPS	Best performing alternatives for ASCII art domain.
10	TS	Random	AOMPS	Initial best performing alternatives for robot domain and string domain.
11	TRS	TPC	AOMPS	Lowest average test-cost, lowest percentage of infinite test-cost, and highest solved percentage of tasks respectively. Combining these might result in a program that performs well on all of these.
12	DSLS	QBC	AOMPS	DSLS performs better than LS, and the other two are the best performing alternatives to their components.

Table 4: Tested combinations of alternatives, their number and their reasoning.

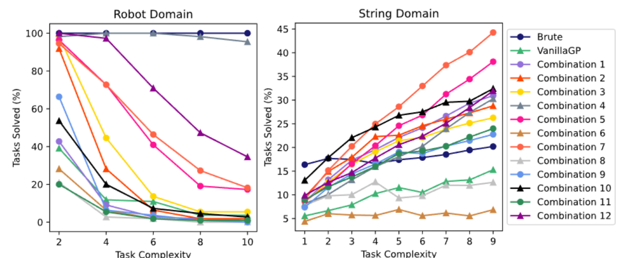

Figure 22: Percentages of solved tasks for the combinations of alternatives.

we decided on evaluating the combinations shown in Table 4.

From the results seen in Figure 22 we can see that when a program performs relatively well in the robot domain, it does not necessarily perform well in the string domain. The Figure shows that all combinations except for Combination 8 outperform VanillaGP, and that most outperform Brute from task complexity 4 and up.

One point of note is that when we compare all combinations to the alternatives separately, we can see that AOMPS combined with SUS and One-Point crossover outperforms all combinations that were compared in this subsection. To see why, we look at the reasons for why the programs were incorrect.

Again, in Table 5 we see that the better performing combinations have a higher percentage of infinite test-cost programs. Although for most combinations the bottleneck is still either the time-out or the iteration limit, the actual percentages are much more divided when compared to the components separately.

When we look at the final programs themselves, they have the same problem as the alternatives separately that causes the bad performance. For Combination 6 the percentage of programs that contains loops is 44%, and for Combination 7

	Robot Time-out	Robot Iterations	String Test-cost	String Train-cost	String Time-out	String Iterations
Brute	N.A.	N.A.	2.9%	4.9%	92.2%	N.A.
VanillaGP	60.3%	39.7%	6.7%	5.7%	85.6%	2.1%
Combination 1	0.2%	99.8%	8.3%	15.6%	0.1%	76.1%
Combination 2	0.5%	99.5%	6.1%	12.1%	0.2%	81.6%
Combination 3	0.5%	99.5%	7.8%	11.9%	14.2%	66.4%
Combination 4	77.8%	22.2%	10.5%	15.2%	73.5%	0.9%
Combination 5	95.7%	4.3%	15.5%	19.3%	54.3%	11.1%
Combination 6	0.3%	99.7%	1.0%	2.9%	0.0%	96.1%
Combination 7	0.4%	99.6%	15.9%	20.9%	13.6%	49.9%
Combination 8	47.3%	52.7%	7.3%	8.1%	78.9%	5.7%
Combination 9	0.2%	99.8%	7.5%	9.9%	13.0%	69.9%
Combination 10	0.0%	100.0%	7.4%	14.3%	0.1%	78.3%
Combination 11	88.6%	17.8%	8.4%	10.3%	2.7%	78.9%
Combination 12	0.0%	100.0%	10.9%	15.2%	73.9%	0.1%

Table 5: Percentages of reason for failed task. The percentages are rounded to the nearest single decimal, or to 0.1 if between 0.0 and 0.1.

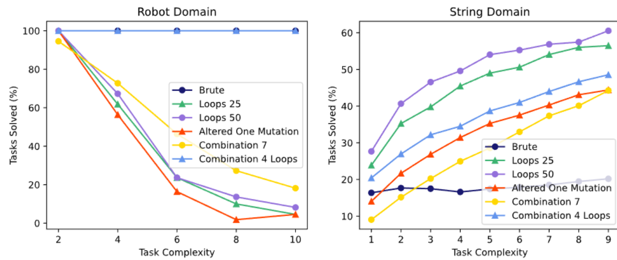

Figure 23: Percentages of solved tasks for the more loops implementation and benchmarkings.

this is 72%. When we adjust them like the components, these percentages become as little as 5% and 39% respectively.

To test our hypothesis of the importance of loops, we implemented one more mutation alternative. This is an altered version of AOMPS, which guarantees p% of the mutations to be a loop. Two versions were evaluated, 1 with p equals 25% and 1 with p equals 50%, because for the best performing alternatives 75% of the final programs already have loops in them and we do not want to force every single program to have a loop. In Figure 23 we can see that while both percentages perform significantly better than the other found combinations, the 50% performs the best by solving 49.8% of all string tasks. Its biggest limitations are the max iterations with 45% and train-cost in second place with 26%.

This is reflected in the percentage of programs that contains a loop. For the 50% this is 96% and for the 25% this is 94%. Compared to the 83% for AOMPS and the 72% for Combination 7 this is significantly higher. Therefore, we can conclude that the percentage of programs that contains loops significantly influences the performance of the combinations.

To create an algorithm that performs better on all domains, we took the combination that performs the best on the ASCII art and robot domain, Combination 4, and replace its mutation algorithm with the version of AOMPS that guarantees 50% of the mutations to be a loop. We successfully created a combination that performs relatively well on all domains as visible in Figure 23. For the ASCII art domain this combination solves 98% of the tasks for complexity 1, 22% of the tasks for complexity 2, and 0% for complexity 3 and up, with a total of 24.0% of all tasks solved. This is significantly higher than for any other combination for the components. When we look at the reasons for incorrect programs, we see that 100% of the incorrect programs are because of a time-out.

6 Conclusions and Future Work

Conclusions

This paper answered the question *Are there alternatives for the components of VanillaGP, a Genetic Algorithm, that separately or combined allow it to solve a higher percentage of tasks within the given Inductive Program Synthesis domains during the same time frame?* To answer this question, many different alternatives and some of their combinations were evaluated on their performance within 3 given domains: robot, ASCII art, and strings.

For all of the components we found alternatives that solve a significantly higher percentage of tasks within two of the domains, robot and strings. The ASCII art domain allowed us to only find 1 combination of alternatives that performs significantly better. The evaluations showed that algorithms that performed better created a higher percentage of programs that contained loops. Therefore, we conclude that loops are a key part of the performance of a program.

On the string domain Stochastic Universal Sampling combined with Queen Bee Crossover and an altered version of One Mutation Per Solution (OMPS) performs the best. This altered OMPS allows for a gene to be added to the front or back of the program and ensures that at least 50% of the mutations the chosen allele is a loop. On the other domains Down-Sampled Lexicase Selection combined with Three Parent Crossover and the same altered version of OMPS, solving 100% of the robot tasks.

While we can answer the question with a resounding yes, the found combinations are not perfect. The best performing combination on the ASCII art domain solves 24.0% of all tasks, reaching a time-out in 100% of the cases in which the produced program is incorrect. The best performing combination on the string domain solves 49.8% of all tasks, with the generation limit being the most common reason for an incorrect program. Therefore, we can conclude that the biggest limitations for the evaluated algorithms are the time-out, the generation limit, and the lack of loops in produced programs.

Future Work

Based on the conclusions from this research the following possible future work is presented. The current fitness function that is used only considers the error of the produced program when given the input. To create a better fitness function, different metrics could be used. Some that can be explored are as follows: favouring loops over many if-statements, the length of the programs, and the execution time of the programs.

There are many different settings of the Genetic Algorithm but also the token generation that were not changed during this research. These include the population size, the generation limit, the time-out, the max length of the programs in the initial generation, and the max token depth. Settings like these influence the performance of the Genetic Algorithm, so exploring these further could result in an algorithm with better overall performance.

7 Responsible Research

Responsible research was an integral part of this research. There are multiple aspects that we discuss to this point.

The first aspect is the reproducibility of the results. To attain reproducibility, the experimental setup was explained in detail and all experiments use the same set-up. The set-up is the same as for Brute [Cropper and Dumančić, 2020], therefore making sure that the data is most likely valid and representative for these domains. It is also mentioned that the experiments were run on the the Delft University of Technology DelftBlue supercomputer [Delft High Performance Computing Centre (DHPC), 2022]. When running these experiments on a different machine the results can vary because of the large computational load. The fact that information on the entire set-up including the machine that experiments were run on is available, combined with the fact that the repository containing the code for this research and the training and test data is available, makes sure that similar results can be reproduced. However, as mentioned before, the results that were obtained are only from one run of each experiment and thus can vary when running an experiment again. To get more general and more valid results, the results should be averaged over many runs of the same experiment.

The second aspect is impartiality toward the obtained results. To achieve this, this paper mentions all alternatives and combinations of alternatives that were experimented with, even if the results were not positive. There were many alternatives and even some combinations that did not outperform VanillaGP, yet they were still mentioned. Although the goal of improving on VanillaGP was fulfilled, there were no combinations of alternatives found that come even close to Brute's performance on the ASCII art domain. Despite this, the results are still published.

The third aspect is transparency. This paper is written to be as clear as possible as to why alternatives and combinations of alternatives were chosen and implemented. To do this, each choice that was made has its reasoning mentioned. These reasons were mostly citations from other literature or findings during the experiments. It is also made clear in this paper that the performance of Genetic Algorithms depend on the domains they are tested on and that therefore we cannot make any hard conclusions of the performance of Genetic Algorithms as a whole on the entire domain of Program Synthesis. To attain this, more experiments should be done of more combinations and these experiments should be done on more domains.

References

[Azimzade and Dumančić, 2022] F. Azimzade and S. Dumančić. *VanillaGP: Genetic Algorithm for Inductive Program Synthesis*, 2022.

[Cropper and Dumančić, 2020] A. Cropper and S. Dumančić. Learning large logic programs by going beyond entailment. In C. Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 2073–2079. International Joint Conferences on Artificial Intelligence Organization, 7 2020. Main track.

[Deb and Deb, 2014] K. Deb and D. Deb. Analysing mutation schemes for real-parameter genetic algorithms. *International Journal of Artificial Intelligence and Soft Computing*, 4:1–28, 02 2014.

[Delft High Performance Computing Centre (DHPC), 2022] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 1). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1, 2022.

[Ferdowsifard *et al.*, 2021] K. Ferdowsifard, S. Barke, H. Peleg, S. Lerner, and N. Polikarpova. Loopy: Interactive program synthesis with control structures. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[Ferguson *et al.*, 2020] A. Ferguson, J. Hernandez, D. Junghans, Alexander Lalejini, E. Dolson, and C. Ofria. *Characterizing the Effects of Random Subsampling on Lexicase Selection*, pages 1–23. 05 2020.

[Gulwani *et al.*, 2017] S. Gulwani, A. Polozov, and R. Singh. *Program Synthesis*, volume 4. NOW, August 2017.

[Helmuth and Abdelhady, 2020] T. Helmuth and A. Abdelhady. *Benchmarking Parent Selection for Program Synthesis by Genetic Programming*, page 237–238. Association for Computing Machinery, New York, NY, USA, 2020.

[Jebari and Madiafi, 2013] K. Jebari and M. Madiafi. Selection methods for genetic algorithms. *International Journal of Emerging Sciences*, 3:333–344, 12 2013.

[Jha *et al.*, 2010] S. Jha, S. Gulwani, Sanjit A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224, 2010.

[Kaya, 2011] M. Kaya. The effects of a new selection operator on the performance of a genetic algorithm. *Applied Mathematics and Computation*, 217:7669–7678, 06 2011.

[Kora and Yadlapalli, 2017] P. Kora and P. Yadlapalli. Crossover operators in genetic algorithms: A review. *International Journal of Computer Applications*, 162:34–36, 03 2017.

[Mitchell, 1996] M. Mitchell. *An introduction to genetic algorithms*. Cambridge, Mass. : MIT Press, 1996.

[Osera and Zdancewic, 2015] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 619–630, New York, NY, USA, 2015. Association for Computing Machinery.

[Pavai and Geetha, 2016] G. Pavai and T. V. Geetha. A survey on crossover operators. 49(4), 2016.

[S. Mooi *et al.*, 2017] S. Lim S. Mooi, A. Bakar Md Sultan, Md Sulaiman, A. Mustapha, and K.Y. Leong. Crossover and mutation operators of genetic algorithms. *International Journal of Machine Learning and Computing*, 7:9–12, 02 2017.

[Solar-Lezama, 2018] A. Solar-Lezama. *Lecture 2: Introduction to inductive synthesis*, 2018.

[Soni and Kumar, 2014] N. Soni and T. Kumar. Study of various mutation operators in genetic algorithms. 2014.

[Spears and Anand, 1992] W. Spears and V. Anand. A study of crossover operators in genetic programming. *Proceeding of the Sixth International Symposium on Methodologies for Intelligent Systems*, 542, 07 1992.

[Umbarkar and Sheth, 2015] A.J. Umbarkar and P.D. Sheth. Crossover operators in genetic algorithms: A review. *ICTACT JOURNAL ON SOFT COMPUTING*, 06:1083–1092, 10 2015.