Multi-Donor Neural Transfer Learning for Genetic Programming

ALEXANDER WILD and BARRY PORTER, Lancaster University, UK

Genetic programming (GP), for the synthesis of brand new programs, continues to demonstrate increasingly capable results towards increasingly complex problems. A key challenge in GP is how to learn from the past, so that the successful synthesis of simple programs can feed in to more challenging unsolved problems. Transfer Learning in the literature has yet to demonstrate an automated mechanism to identify existing donor programs with high-utility genetic material for new problems, instead relying on human guidance. In this paper we present a transfer learning mechanism for GP which fills this gap: we use a Turing-complete language for synthesis, and demonstrate how a neural network (NN) can be used to guide automated code fragment extraction from previously solved problems for injection into future problems. Using a framework which synthesises code from just 10 input-output examples, we first study NN ability to recognise the presence of code fragments in a larger program, then present an end-to-end system which takes only input-output examples and generates code fragments as it solves easier problems, then deploys selected high-utility fragments to solve harder ones. The use of NN-guided genetic material selection shows significant performance increases, on average doubling the percentage of programs that can be successfully synthesised when tested on two separate problem corpora, in comparison with a non-transfer-learning GP baseline.

CCS Concepts: • **Computing methodologies** → **Supervised learning**; **Neural networks**; **Genetic programming**.

Additional Key Words and Phrases: genetic programming, neural networks, transfer learning, genetic algorithms

ACM Reference Format:
Alexander Wild and Barry Porter. 2022. Multi-Donor Neural Transfer Learning for Genetic Programming. *ACM Trans. Evol. Learn.* 1, 1 (August 2022), 43 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The ability to synthesise source code from input-output examples, in which a source code implementation of a function is created based on one or more demonstrations of input-output mapping, is a fundamental question in machine learning. We specifically study this question in the form of scenarios where large corpora of existing code are not available (e.g., human-written programs in open-source repositories). Providing this capability would allow non-programmers to generate programs, or experts to abstract away trivial coding tasks. In addition, from a machine learning perspective, it allows complex functions to be generated from training examples in a symbolic and human-readable form, in contrast to the current trend of function-approximation training which stores results opaquely in NNs. Having a symbolic, inspectable representation allows us to subject generated functions to a wide range of static analysis tools to model generality or correctness.

To date, this challenge has been studied by researchers with approaches including NN-driven synthesis, GP, and SMT solvers. At present, both neural-synthesis and SMT approaches are significantly constrained in the complexity of the target language in which code is synthesised, particularly in control-flow operators such as loops and branch statements, which limits the kinds of program that can be generated [2, 32–34, 40]. GP, meanwhile, is limited by our ability to specify a fitness function which can successfully navigate to a solution for a particular problem – in the high-dimensional,

Authors' address: Alexander Wild, a.wild3@lancaster.ac.uk; Barry Porter, b.f.porter@lancaster.ac.uk, Lancaster University, Lancaster, UK, LA1 4YW.

2022. 2688-3007/2022/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

highly irregular, and often neutral fitness landscape of program space [21, 30]. The use of transfer learning (TL) in GP has shown initial success in highly specialised problems to help narrow the search area, but to date has used significant human input in selecting which base material to transfer [25, 28].

In this paper we examine fully automated transfer learning for GP, in which a NN is trained on already-solved problems and determines which code fragments may be useful for new problems based on inferred similarities between their input/output examples. This allows us to start from simple problems which can be solved by a GP process alone (where no already-solved problems are available), then use these solutions to facilitate the solving of more complex problems through NN-suggested donor material. We examine the capabilities of this approach in a Turing-complete programming language which can be cross-compiled into C/Java code and includes loop and branch operators, demonstrating the performance of our approach in the very large search spaces that such a language presents.

The key challenges in using a NN in this capacity are (i) finding an approach which avoids the NN needing to model all of program space – which is intractably large – and (ii) finding a way to estimate utility of genetic material from existing solutions for new problems. For the first challenge we employ a solution in which the NN models highly abstracted features of program space, and demonstrate that these abstract features are still sufficiently useful that they allow a GP process to solve problems, with suggested donor genetics material, that it is otherwise unable to. For the second challenge we use NN-inferred similarity of input/output example data between existing problems and new unsolved ones; solved problems with a higher level of similarity to new problems, in their input/output examples, are judged to have more useful donor code to those new problems. Each time we solve an additional problem, the solution to that problem is used in a further round of NN training to yield further predictions on potentially useful material for other problems.

We study our approach when applied to two different problem sets: a set of array-based problems which are similar to those used in other code synthesis research, including sorting an array, where a user provides example input/output such as an unsorted and sorted array (full description of problems in Appendix B); and a set of canvas drawing problems in which we need to synthesise the algorithm which draws a user-provided shape (which has also been used in synthesis research elsewhere [32]). Our results demonstrate that our NN-based transfer learning approach significantly enhances the performance of a baseline GP system, doubling the percentage of programs that can be successfully synthesised.

2 RELATED WORK

Code synthesis from input/output (IO) examples has been studied using three major approaches to date: deductive solvers; neural networks (NNs) with search; and genetic programming (GP). For code synthesis in a Turing-complete language, no deductive solvers have yet been shown to operate well with loop-based flow control operators (although frameworks which manually define any non-linear program flow can yield good performance [32]). Neural synthesis, by comparison, is limited by how much of program space for a more general-purpose target language can be captured in a NN model, while GP is limited by the difficulty of deriving a fitness function to navigate to a solution [30]. In the remainder of this section we focus on NN and GP approaches in more detail. We consider only works concerning programming by example, in which input/output examines are used to specify the desired behaviour of a program; other code synthesis work (e.g. based on natural language prompts [5]) is outside the scope of this research.

Neural synthesis. Neural synthesis works by training a NN on a sub-sample of the entirety of program space for the target language (e.g., sampling at a uniform interval or at random). When

presented with a new problem as an IO example, the NN will then be asked to predict which lines of code (or particular operators) are likely to be present in the solution based on similar IO transforms observed in the training set from the above sub-sample. The system will then perform an exhaustive search of program space to fill in the remaining (non-predicted) features. Notable examples here include DeepCoder and RobustFill, among others (e.g., [2, 7, 8, 31, 40])

The key limitation to this approach is that it must be able to train on a detailed enough sub-sample of program space, and store this sample inside a NN, to make meaningful predictions on program features for unseen problems. While this works for highly simplified languages (DeepCoder, for example, has no loop operators [2]), the search space size of a Turing-complete language is astronomical by comparison. If we consider that the capacity of a feed-forward ReLu NN to differentiate between classes (in our case programs), termed its Vapnik–Chervonenkis (VC) dimension, at best grows as a linear function of $w * L * log(w)$ where w is the total number of weights and L the number of layers [3], it is unlikely that a NN on current hardware would be able to represent a useful sub-sample of possible program permutations yielded by the search space of a more general-purpose language.

NNs have also been deployed used as a way to directly navigate program space and synthesise code. Bunel et al [4] examined training a NN with a set of samples of different programs against how close the output of those programs was to the desired output of a target problem. They then had the NN generate its best estimate of the program that perfectly matched the desired output; if this program was not correct they it to re-train the NN again and repeat the process of estimation. This approach depends on a NN being able to predict a program's output without executing it, and also requires that the NN can internally represent the majority of total program search space. While this shows good results in some cases, the extent to which these two assumptions are generally true is unclear.

Genetic programming. GP relies on iterative travel through program space from a starting point to the solution, guided by a fitness function [35, 38]. The field has a long history [12] and shows results [24] that are competitive with NN-synthesis [1], and an ability to tackle complex problems mixing diverse datatypes [29].

Unlike neural synthesis, a GP approach does not need to encode the entirety of program space in a model, and so can in theory work in a scalable fashion on high-dimensional search spaces as long as a fitness function is provided which can guide the search incrementally towards a solution. The key problem with GP for code synthesis is that large areas of program space are difficult to navigate, exhibiting large plateaus of neutrality (no fitness change despite significant code change) and highly irregular responses to code change (jagged fitness landscape) [21, 30]. A fitness function may potentially also fail to capture higher level properties, for example “all outputs are even" or “all values from the input are repeated in the output" if the designer does not consider these features, which may otherwise be identified by a NN which is able to learn in an online fashion based on problems encountered.

Despite these difficulties, the literature of GP has, to our knowledge, exceeded that of direct neural code synthesis, with recent strong results [29] on complex multiple-data-type functions. As such, we should seek to leverage the power of GP for search, while also employing the capability of NNs to recommend high-utility genetic material from existing solutions to narrow the overall search space.

Augmenting GP search with additional input has been explored by Hemberg et al [16], but using domain-specific knowledge rather than NN-suggested genetic material. Hemberg's work analyses the program specification, which in this case is provided in terms of both the IO examples and an accompanying natural language description, and specialises the GP process towards a

particular problem – for example by augmenting the literals available or by biasing the mutations. Our approach by comparison does not require natural language descriptions, and instead learns which genetic material from solved problems is likely to be useful to inject into new problems. Both approaches are potentially complementary, however, and could be used in combination.

In wider GP research, a General Program Synthesis Benchmark Suite [15] has been proposed as a common point of evaluation, consisting of 29 different problems for which synthesise a solution (and having been used in existing research, e.g. [10, 11]). While we considered the use of this benchmark in our own work, we noted that the problems included are selected from a wide range of different domains, in terms of input/output data types, and so present a relatively small set of problems in each domain against which NN-guided transfer learning could be attempted. We therefore use two different problem sets: one array-based set, in which an array of input integers must be transformed in some way including sorting the array; and one canvas-drawing problem set which has been used in existing research [32].

Existing work in transfer learning for GP. Finally, transfer learning (TL) has been studied to some extent in the field of GP, in particular with a large body of work into the effects of transferring sub-trees between symbolic regression problems and how to select trees for best results [6, 25, 28]. Our work differs from this in two major areas. The first is that we present a fully automated approach to selecting which prior solved problem, and associated code fragments, to use as a basis for new unsolved problems – rather than relying on human-aided selection. As such, we require no designer knowledge of the problem space in selecting useful genetic material for transfer in to new problems.

The second difference is the target environment. The main body of work in TL for GP focuses on symbolic regression, with work on numeric regression and boolean tasks. While good performance has been shown on this target domain, our work focuses on a Turing-complete programming language with a broader set of problems – representing both a larger search space in the target language, and more generalised problems.

Relating to more general programming, existing approaches to transfer learning in GP tend to use existing solutions as callable functions for new solutions (e.g. [18–20]). This allows sub-tasks to be learned first, then exploited to solve more complex tasks which involve those sub-tasks. This differs significantly from our approach, in that we are able to extract selected code from solved problems in order to narrow the search space for new problems, based on NN-guided inference on similarity between each problem. This allows our approach to solve problems that don't necessarily re-use existing solutions in their entirety, but rather have some similar features. The ability to re-use whole functions is entirely complementary to our work and could be used in combination.

Finally, a range of research has examined more granular transfer of learned features across problems. Jaskowski et al [17] which employ GP to solve visual character recognition tasks, and allow transferral of geometric splines to allow shared properties of the glyphs to be exploited for faster learning; this shows compelling results, though is a relatively problem-specific approach to knowledge transfer. Munoz et al [26] examine how donor-recipient pairings of programs may benefit from TL across different problems, and studies the properties of these donations, while Helmuth et al [14] conisder adjusting the mutations used in a GP process based on transfer learning, among other sources of guidance. As far as we are aware, our research is the first to examine fully automated recommendation of selected source code fragments from already-solved problems to increase the probability of solving new problems.

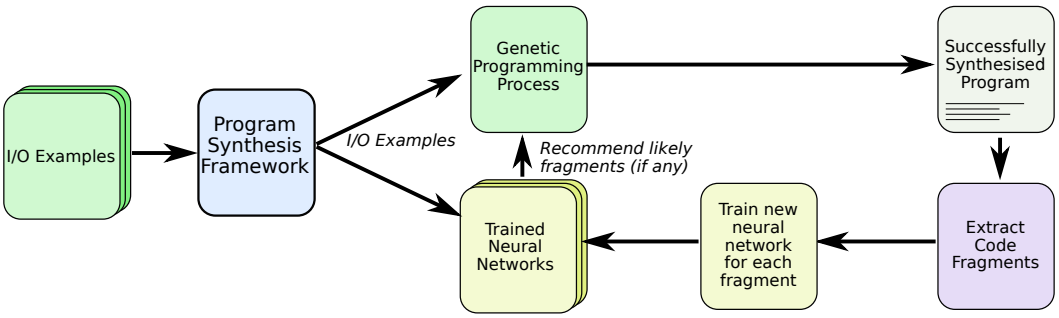

Fig. 1. Diagram of our end-to-end code synthesis system, combining GP with fragment extraction and prediction.

3 METHODOLOGY

Given the limitations of both NNs and GP in themselves, we hypothesise that the combination of the two techniques may provide the best features of both while mitigating their respective limitations in the context of code synthesis in a Turing-complete language with a very large program search space. While a NN cannot feasibly model all of program space, and so cannot be expected to predict each line to be synthesised, it does have the potential to predict a small number of higher-level features which only require a very limited internal model of program space. This can be combined with a GP search process which fixes these lines in place to constrain the search area, and can use this partially-constructed program in combination with one or more generic fitness functions to guide the search to a successful result.

Our overall system is illustrated in Fig. 1. It takes up to ten input/output examples from a user, then synthesises the source code of a function which correctly converts those inputs to their corresponding outputs. Prior to any existing problem solutions being available, our system uses only GP to locate a solution. Once at least one solution is available, our system selects one of more source code fragments from those solutions. For each separate fragment identified, a NN is then trained to recognise whether or not the solution to an I/O example is likely to involve that fragment. We do this by synthesising 2,500 unique programs that do include the fragment, and 2,500 unique programs that do not include that fragment, then synthesise random input arrays to feed in to those programs and gain their corresponding outputs. Using these examples, our NN is then trained to infer whether or not a given I/O pair does, or does not, involve a program containing the fragment of interest. We train one NN in this way per fragment extracted; when a new problem is presented as a set of I/O examples, we then query each trained NN with these I/O examples to determine which fragments are most likely to appear in the solution. Each time another new solution is found to a new problem, we again perform fragment extraction from that new solution and NN training for each fragment.

In the remainder of this section we elaborate on each element of our system: the programming target language and problem description format that we use for synthesis; the GP system; the NN architecture; and our fragment extraction process.

3.1 Target language and problem description format

We use a Turing-complete language for synthesis (previously used by the authors of [39]), which can be cross-compiled directly into C/Java/Python. Unusually, for general program synthesis, the language features primitive loop operators, variable declarations, and conditional branch operators; a full listing of its operators is given in Appendix A.

Our system uses a human-configured fixed upper limits on program length, which we set to 30 lines of code for all elements of this study; this value was chosen based on human analysis of how many lines would be needed to implement any problem in the two corpora used. We also employ a fixed number of variables based on a similar analysis, allowing 12 integer variables, two 1-D array-of-integer variables and 1 2-D array representing a basic bitmap canvas. All variables are automatically initialised to zero when the program is called, setting the integers to 0 and each array to a newly-instantiated zero-length array.

We use two different problem sets which represent human-useful problems of the kind that may be input into our system. Our first problem set is of 35 array-to-array problems such as *extract even numbers*, *append arrays*, or *sort*. Our second problem set of 30 problems is provided with a 2D black-and-white image to draw on a canvas, encoded as a matrix of boolean values, and must synthesise the program which draws that image; this problem set is taken from So and Oh [32].

For the array to array tasks, each problem represents a program which takes one 1D array of integers and one integer variable. Not all problems employ the integer variable (such as the sort or reverse functions), but the GP is not informed as to which are required to and which are not, and must simply learn to discard the input integer variable as the problem corpus problems do.

For the second problem corpus, the programs are able to write to a 2D array by addressing specific pixels, and must be able to output the image at different scales, depending on an input integer variable which specifies both the height and width of the canvas. The images themselves range from the simplistic 'square' which fills the canvas with black pixels, to requiring obtuse triangles (which requires lines of pixels to be drawn on non-orthogonal or diagonal vectors).

Each problem within each problem set is presented to the system as a set of 10 I/O examples. Based on a human-crafted example solution to each problem, we generate our I/O examples by randomly sampling an array for input, feeding this input into our program, and gaining the corresponding output. Using randomly-generated examples is designed to reduce internal variability between tests, allowing more accurate evaluation of the changes made by alterations to parameters or by guidance to the GP. This is designed to represent an unbiased input set. The inputs are consistent across each run, to remove this particular random component when comparing baseline to TL system.

3.2 Genetic Programming

We use a relatively standard GP process, aside from our fitness function. We use fixed-size populations of either 1,500 or 2,000 individuals (depending on the experiment setting), and a generation limit of 3,000 after which the system reports a failure to synthesise its target program. Every new GP run starts from a set of individuals which each represent the empty program, which we define as a program of maximum length set to only NO-OPERATION on every line. For each new generation during the GP search, parents are chosen using tournament selection with tournament size of 10 [23]. Crossover occurs by taking the first half of the first parent, and the second half of the second parent, with both parents padded to the maximum program length by appending NO-OP lines. This approach makes crossover simple and fast to compute, irrespective of the two programs being considered, but means that in initial generations no crossover may occur in practice while programs are relatively short. Following crossover, we apply a single mutation to a program with a probability of 0.35. Following each mutation a random boolean value is selected, and if true another mutation is applied, to a maximum of 8 mutations.

Each mutation takes one of the following forms, selected with equal probabilities for all choices other than delete, which is weighted twice as highly as other mutations:

(1) Insert: inserts a random line from all possible lines available in the language, and deleting the last line of program if it is already at maximum length. It automatically adds an ENDBLOCK operator if a flow-control operator was inserted.
(2) Delete: sets a random line to NO-OP.
(3) Mutate: changes a random operator/parameter on a line, ensuring the line remains valid.
(4) Swap: exchanges position of any two lines of the program.
(5) Insert Fragment: This mutation is only available if a set has been provided. Selects a set of lines from a previously generated program and inserts them into the candidate program's source code. See Experiment 3 for details on this mutation.

Each program in a population is ranked using a general-purpose fitness function which was experimentally found to be good at locating programs from various search space starting points. Our fitness function includes an implementation of niching [13] which operates across generations. This was intended to act as a form of novelty search [9, 22] to avoid falling into the same local minima repeatedly, which was found to further boost search success and was deployed in all of our experiments. It is based on the Levenshtein distance [27] of the programs produced to the elites gathered in precious generations.

Our fitness function, in detail, is formulated as $error * novelty\_penalty$. The value for *error* is calculated as the number of elements in the output of a candidate function which do not match those expected by the input/output pair for the corresponding input; this value is normalised into the range $[-1.0, 0.0]$ by dividing it by the corresponding error which would have been produced by an *empty* output for that input. If the output arrays do not match in length, a penalty value of $-10,000$ is applied instead, as item-wise comparison becomes impossible. This means that if a program outputs $[1, 2, 4]$, when the desired output was $[1, 2, 3]$ it will be considered to have an error of 1, as will a program which outputs $[1, 2, 256]$, as they both fail to match a single element's value. The actual magnitude of this difference is ignored; while directly using this magnitude as part of fitness may be useful in some scenarios, it was not found to yield any improvements within our testing.

To calculate the value for *novelty_penalty*, we aim for an approach which is fast to calculate (as we need to perform many novelty checks across thousands of individuals) while representing a good approximation of novelty. To encode novelty we first extract the highest-fitness member of a population and remove all lines of code which do not contribute to its behaviour. This program is stored as a *repulsor* which indicates how well-explored a given region of program space is. This is similar to the use of repulsors in [37], where they are also used to drive a search process away from a part of the search space by keeping records of points and using them as a way to compute a negative penalty.

A new repulsor is created and stored each GP generation. When calculating the fitness of any new program, we compare that new program against every repuslor in turn to derive an overall novelty value. For each comparison between a program A and a repulsor program B, our procedure is as follows: we take the first line from A, then check every line from B to discover which line is *most similar*. Two lines that are identical, and appear on the same line of the two programs, have a similarity score of 0.0; a different operator adds 1.0 to the score; each parameter that is different adds 0.2; and each line of code that we are away from the code line in program A adds 0.35. When we have found the closest matching line in program B, we remove that line from program B, then consider the second line in program A, and repeat the procedure. When we have a final similarity score, having considered every line from program A against those in program B, this final score D is added to $novelty\_penalty$ using the equation $max(0, 1 - D/15)$ to $novelty\_penalty$. We then repeat this for every store repulsor program. This approach, along with the values for each tunable

parameter, was found experimentally to provide a good balance between speed of computation and accuracy in novelty estimation.

3.3 Neural network design

Initially, when no programs have been successfully synthesised by our system, we rely on GP alone to locate programs. When at least one program has been correctly synthesised it is added to our set of found programs S_F. We then augment GP with a NN which esimtates one or more likely lines of code for a given unsolved problem, corresponding to likely areas of program space in which the GP will search (rather than the GP starting its search from an empty program).

Our overall process is to extract a set of code fragments from a successfully-found program, and train a new NN for each code fragment. Each NN is trained to recognise whether or not a given input/output example is likely to have a solution which involves that code fragment. We do this by synthesising a set of programs which do contain the fragment of interest, and a set of programs which do not. Each generated program in these sets is required to be distinct in its input/output transformation. We then train the network to correctly predict which programs from these sets do and do not contain the fragment of interest. When the NN is then presented with a novel I/O example for an unsolved problem, we hypothesise that it will be able to predict whether or not the solution involves the corresponding code fragment that the network was trained to recognise. We train one separate NN for each code fragment, then check every trained network for its estimate of code fragment presence for a new problem.

In detail, the input layer of a NN takes the data representing 10 input/output examples, with one neuron per bit of this data where integers are represented as 10-bit signed values. The output layer has a single neuron which yields a value between 0.0 and 1.0 representing the estimate of whether or not this input/output example set includes the code fragment of interest. The NN never receives any source code; rather it is trained to recognise whether or not an input/output set is likely to have a code fragment of interest by using gradient descent to train the output neuron's value for each program in our training set, based on whether the input-output transform that each program yields does or does not involve the code fragment of interest. In this sense our NNs do not need to directly encode program space, but an abstracted correlation of input/output behaviour. The only data they receive regarding the code itself is a single binary value, indicating whether the program which generated the IO mapping contained the target fragment or not, and only receive this during testing.

To train each NN we synthesise 5,000 programs in total: 2,500 that do have the fragment of interest, and 2,500 that to not. From these programs, we use 4,000 for NN training and 1,000 for testing. Each program in each set of 2,500 is assured to be distinct in functionality from every other program in the set, tested on its behaviour with respect to a fixed 10 IO examples. Each new program in a training set is generated by selecting uniformly at random two programs already accepted, applying a crossover, then applying between 1 and 8 mutations (as described above in the GP section) while assuring that the fragment of interest still exists or does not exist, according to the sub-corpus requirement. This value of 2,500 was chosen as it was the lowest value which appeared to not degrade performance of the NN, in preliminary testing. Lower values would reduce accuracy, while higher values produced no noticeable improvements.

We use a Feed Forward Neural Network (FFNN) architecture for our array-to-array tasks, and a Convolutional Neural Network (CNN) for our 2D canvas tasks. Our FFNN architecture takes the form of 4 layers of 128 neurons, seLu activation, interspaced with dropout layers set to 0.75 keep rate. Each layer was connected to all previous layers (dense block architecture). Our single output node was a sigmoidal node (values constrained to between 0 and 1.0), representing the probability of a given fragment being present in the program assumed to have implemented the presented IO

example. Our CNN architecture was two sets of: one convolutional layer of stride 2, kernel width 3, 32 channels, reLu activation followed by one max-pooling layer with a pool width of 2.

The evaluation time for a single NN, to predict presence of a particular fragment in a new problem, is on the order of 100ms, and the total cost of evaluating all NNs never reached 1% of the total computational cost of our overall code synthesis system. This is due to the speed difference between a NN evaluation and a GP run.

3.4 Fragment extraction and usage

Our automated fragment extraction works as follows. For every newly-solved problem, we extract every 'cleanly isolatable' fragment up to 4 lines of code long, such that no variables used in the fragment have an assignment before the code in that fragment (this condition applies transitively to variables used by other variables, etc.). We then filter all fragments out from this set that we've seen before from other solved problems, and for which we already have a corresponding trained NN. From the remaining fragments we train a NN on each one, and test if that NN has a prediction accuracy of greater than 50%. If it does, we keep that trained NN; if not, we discard it and try the next fragment from our list. We continue this process until we have either found 4 trained NNs with good prediction accuracy, or until we have made 8 attempts to train such NNs, whichever is sooner. We limit the number of NNs that we train in this stage to reduce overall computation cost as part of the synthesis framework.

When facing a new problem, specified by an I/O example set, we submit the I/O example to all fragments' NNs. Each of these NNs returns a probability estimate P as to whether the code which produced the I/O example contained their associated fragment.

For each fragment with predicted presence in the solution, a mutation weighting is then set, which determines how frequently this particular GP run will select this fragment as a mutation whenever an 'insert' mutation is chosen. We configure this weighting by using the NN-returned fragment presence probability P together with a rarity estimate R, where $R = 0.1 + N_p f$. The weighting for the fragment is then defined as P/R. The value $N_p f$ represents the number of problem solutions that are expected to contain the fragment, and is simply the proportion of problems the GP has been presented with which the NN for that fragment has returned a presence estimate > 0.5. A very commonly occurring fragment (or one which the NN estimates to be commonly occurring) will have a value near 1.0. The NN is assumed to have returned a 1.0 for the program for which the fragment was extracted, so the expected number of problems with that fragment can never be 0. This weighting equation process favours fragments which are (i) the novel elements of difficult problems, thus useful learning targets; (ii) believed to be rarer, thus more specialised; and (iii) have higher confidence of presence.

When the GP process selects mutations during the next run it has an 0.2 probability to select the 'Insert Fragment' mutation. If this is selected, it selects a fragment by roulette selection weighted by the above-defined weighting.

Because this fragment originates from a solution to a different problem, it may to some extent have elements of its source code which do not fit into a new piece of source code, in particular with variables that have conflicting names. We therefore adjust a chosen fragment, at the point of insertion, to fit into the new source code; for each variable we decide at random to either generate a new variable name, or use an existing variable name of compatible type that is already declared in the target source code. We also randomise the way in which a multi-line fragment is inserted; after inserting the first line at a random position in the target source code, we then decide whether to insert the next line immediately below it, or to 'skip' a line of the target source code and insert our next fragment line of code after that skipped line. This provides a probability distribution between inserting a fragment contiguously or interwoven with existing lines of code, and was found to be

useful in covering a wider range of fragment merge permutations – such as the ability to insert a double-loop fragment in such a way that some lines of existing code were executed before the inner loop was reached.

If a program ever exceeds the maximum line length, the last line is repeatedly removed until the program is returned to our configured maximum length of 30 lines.

4 EVALUATION

We evaluate our approach in three ways. We first study how the use of code fragment suggestion alone, when code fragments are manually-chosen, affects the ability of GP to locate programs from our two problem sets. This indicates the extent to which we might expect improvements in our end-to-end system. We then study the extent to which our NNs are able to accurately predict which code fragments should be suggested for each problem; this indicates the extent to which automated prediction is possible. Finally, we study our end-to-end system, which automatically identifies fragments to extract from successfully found programs, trains NNs for each fragment, and uses those trained NNs to suggest code fragments for as-yet-unsolved problems. All stages of our evaluation use the two problem sets introducted earlier: one integer array problem set, including sorting, and one canvas-drawing set in which a program must be synthesised to draw the given black-and-white image.

Throughout our evaluation we use Tensorflow 1.14 and Python 3.6.9 for our NN models, and Java OpenJDK 11.0.6 for our GP system. Our source code is made available as an artifact accompanying this paper submission.

4.1 Fragment Suggestion effects on GP

In our first experiment we examine the baseline performance of GP alone on our two problem sets, and study the effect of each possible code fragment that can be provided as a starting point for each problem (in terms of its effect on success rate).

We start with a ground truth implementation of every problem, which is a hand-crafted version of a correct program. To generate this set the principle researcher wrote what they considered to be the shortest possible implementation of each possible program, attempting to keep implementation as common between all programs. Within these, we then examine every 1- or 2-line code fragment that can be cleanly isolated from those programs, such that no variables used in the fragment have an assignment before the code in that fragment; this condition applies transitively to variables used by other variables.

We then run a GP process with each code fragment as a forced requirement, such that any program produced by the GP which does not include them automatically receives a penalty fitness of -10,000. Our first generation of individuals has the appropriate fragment inserted as its first lines of code, with individuals of subsequent generations checked for the continued existence of this fragment (anywhere in the program). Each experiment in this series is repeated 30 times.

These results show the extent to which a GP algorithm can have its probability of finding a solution increased by constraining its sampled programs to contain certain lines of code. This also serves as a guide and comparison point to the highest-utility hints that our fully automated end-to-end system can seek to find.

Sample results are shown in Table 1, with the full set of results given in Appendix E. Each result shows the 'baseline' pure GP success rate, without any suggested fragments of source code, and the best effect of a GP process that was given a suggested starting fragment together with which fragment(s) offered the best performance. Our sample results show 8 problems from our array-to-array corpus and 6 problems from our canvas corpus. From the first corpus we select mostly low-success-rate problems, to study which form of fragments would be useful to achieve success,

Problem	Baseline	Maximum	Best Operators
Append	0%	27%	Var=Literal 1; Addition
Cumulative Abs Sum	0%	3%	Loop; Read
Keep Evens	0%	7%	Var=Literal 2; Make Array
Retain First Half	0%	13%	Var=Literal 2; Divide
Reverse	58%	80%	Var=Literal 1; Make Array
Shift Right	0%	13%	Var=Literal 1; Loop
Shift Right Lossy	84%	80%	Var=Literal 1
Sort (Bubblesort)	0%	0%	(None)
Parallelogram	7%	30%	Var=Literal 2; Divide
Mirrored Hollow Parallelogram	13%	60%	Var=Literal 2; Divide
Hollow Right Triangle	87%	90%	Var=Literal 1; Subtract
Hollow Mirrored Right Triangle	63%	93%	Var=Literal 1; Subtract
Inverted Isosceles Triangle	46%	23%	Var=Literal 2
Trapezoid	7%	10%	Var=Literal 1

Table 1. Success rates for guided Genetic Programming Algorithm with forced inclusion of code fragments from ground truth. Baseline is unguided GP. Maximum is single best performing fragments. Best operators are those used in the highest-scoring fragment (first if tied) (n=30 per fragment, percentage success)

while in the second corpus we select a more representative sample, to study how constraint-forcing behaves more generally.

Here we clearly see that forcing the inclusion of even simple code elements (such as an addition operator) into the GP's population allows the GP to find previously unsolvable problems. We can also see that fragments containing arithmetic operators (especially literal assignment) appear to have a stronger influence on success than other fragments taken from the same program. This may possibly be because they are harder to find, as their effects are far more subtle and complex than, say, presence of a loop operator. These should therefore be studied as high-utility candidates for deployment into GP processes we wish to guide in the future.

We also note that some examples show a *decrease* in success rate (e.g. "Shift Right Lossy"). While no fragment reduced success rates to zero, we speculate that in some cases the provision of a fragment places the GP into an area of program space from which it is harder to reach the solution using our general-purpose fitness function than it would be by starting from an empty program with no lines of code (for example, meaning that this point in program space has larger regions of neutral landscape around it which are harder to traverse over).

A full breakdown of all baseline GP success rates is presented in Appendix A, with fragments and their successes presented in Appendix E.

4.2 Fragment Recognisability by Neural Networks

Having established that code fragments can be used to improve GP success rates (including from zero to non-zero success rates), we now study the extent to which NNs can predict the existence of those fragements in the source code solution for a given I/O problem.

For this experiment we need to assume that some programs have already been found by the GP alone, and useful fragments identified, from which to automatically synthesise NN training sets based around each fragment. To do so, human selection of fragments is employed, which

is intended to test if a successful fragment extraction process could usefully feed into a neural network-powered fragment deployment system. If the NN could not recognise the fragments, no fragment extraction mechanism would allow the end-to-end system to succeed as intended.

Using the results of our GP-only runs from the study in the above subsection, we select the single highest success-rate program using the GP alone for each 'class' of problem in each problem set. Both corpora are divided into logical sets of problems (such as 'triangles' or 'programs requiring use of a conditional'). These programs represent those which are most likely to have been found first without the aid of the NN. From within the source code of these GP-found programs we select a set of individual fragments from which to generate our synthetic NN training data. These fragments are chosen partly using high-utility fragments identified from our first study reported above, and partly with further fragments of interest that were manually chosen to gain a wider coverage of fragment predictability by a NN.

Having extracted these fragments and trained corresponding NNs, one per fragment, we then test each NN's ability to correctly predict its fragment's existence in every other problem in the corpus.

Altogether, these experiments indicate how well NNs can predict the presence of different program features based on our synthetic training sets – a mechanism by which the solution to easy (high-success-rate) problems can be used to find solutions to hard (low-success-rate) problems.

The results are shown in Table 2, demonstrating the percentage of true positives and true negatives that the NNs for a set of example fragments achieved. The first row of the table, for example, indicates that the NN trained on the fragement 'declare nonstandard array' accurately predicted that this fragment either should, or should not, appear in 73% of the other problems. For 27% if those other problems, the NN either provided a false positive or a false negative in its prediction.

This data shows significant variance of prediction success (from 45% up to 85%) but overall shows that our NNs do exhibit the ability to accurately predict that a particular source code fragment will exist in the solution to a given I/O problem – even though these NNs are trained on entirely synthetic data.

In practice we use these trained networks when determining which fragments to recommend for inclusion in a GP search; we present this in the following section as part of our full end-to-end system which uses any programs found so far as sources for new fragments, trains NNs to recognise those fragments on I/O problems, and deploys those fragments for new problems.

4.3 End-to-End System

In this experiment we use fully *automated* fragment extraction, sourcing our fragments from solved problems and redeploying those fragments to new problems based on NN guidance.

For both the array- and canvas-based corpora, we run through the entire problem set twice, in the same order both times. This allows our system to successfully find any problems that are possible in its first pass, then use extracted fragments on unsolved problems in a second pass. In order to provide a fair comparison against a baseline GP-only system, we allow that baseline system to also have two full attempts at both problem sets, yeilding a roughly equivalent number of total GP generations available.

Our high-level results are shown in Table 3 against our GP-only baseline, measured in both success rate and total problems solved. The *problems solved* column indicates, as a total across 30 repeated runs, how many of problems from each corpus were solved under each approach. The *success rate* column is calculated by determining, for each problem individually, how many of the 30 repeated runs found a solution to that problem; we then average this percentage across all problems in the corpus to determine the above average success rate.

Fragment	Accuracy
Nonstandard Array	73 %
Loop + Iterator Minus One	N/A
Loop + Read	65 %
Literal 2	76 %
Read	67 %
Add	48 %
Loop	67 %
Loop + Iterator Plus One	N/A
Conditional	63 %
Loop + Iterator Mod 2	73 %
Two Draw Operators	67 %
Add	77 %
Loop + Draw On Iterator X	62 %
Half	84 %
Half + Loop to Half	80 %
Half+Loop+Draw Depends on Iterator	85 %
Conditional	62 %
Loop + Conditional	59 %
Loop + Loop, Add Iterators	53 %
Loop + Loop, Subtract	45 %
Fragment	**FFNN**

Table 2. Percentage accuracy on NN estimates of fragment presence in non-seed programs, after training on synthetic datasets derived from seed programs (one corpus from seeds with fragment, one for those without). Results are averages of networks which passed validation (> 0.5 accuracy on seed programs). N/A results are those where non passed validation. Half here refers to the two line fragment "var = literal value 2; var2 = input size (passed into function) / var". (n=20)

Corpus	Success Rate (vs baseline)	Problems Solved
1D Arrays	**76%** (37%)	**33** (28)
2D Arrays	**49%** (37%)	**29** (22)

Table 3. Success of the transfer learning system in a fully automated pass through both the 1st corpus (1D array to array problems) and 2nd corpus (2D array/canvas pattern generation problems).

Our detailed per-problem results for our array-based corpus are shown in Table 4. Out of a total of 30 experiment repeats, this table shows the number repeats for which the baseline (B') succeeded at each problem, and the number of repeats for which our synthesis framework succeeded. It also shows the number of those successes which were found to *generalise* (g) to an additional 1,000 I/O examples beyond the 10 examples used to synthesis the code. Fisher's Exact Test was used to establish a statistical significance for each separate problem. In this paper, we consider a statistical significance of 0.01 to be sufficient to report as a result. This lets us see which problems the TL

Problem	B' Success (g)	Success (g)	Fisher Significance
Add	14 (7)	25 (17)	0.002
Append	0 (0)	28(15)	$4.193 * 10^{-15}$
CumulativeAbsoluteSum	2 (1)	12 (5)	0.002
CumulativeSum	5 (4)	21 (14)	$2.917 * 10^{-5}$
KeepEvenIndices	7 (2)	30 (23)	$8.705 * 10^{-11}$
ClipToMin	16 (2)	27 (17)	0.001
RetainSecondHalf	0 (0)	10 (3)	$3.985 * 10^{-4}$
Sort	0 (0)	0 (0)	1.0
Subtract	10 (5)	25 (23)	$8.247 * 10^{-5}$
Abs	11 (5)	22 (19)	0.003
GreaterThan	3 (2)	5 (5)	0.25
IndexParity	29 (22)	30 (29)	0.5
FirstElementOnly	10 (3)	30 (23)	$7.167 * 10^{-9}$
Identity	29 (25)	30 (30)	0.5
DivergentSequence	12 (0)	28 (19)	$8.975 * 10^{-6}$
Double	14 (4)	30 (28)	$9.720 * 10^{-7}$
ShiftRight	0 (0)	26 (16)	$3.921 * 10^{-13}$
ShiftRightLossy	18 (8)	30 (28)	$6.181 * 10^{-5}$
ShiftLeft	5 (2)	30 (24)	$2.744 * 10^{-12}$
ShiftLeftZeroPadded	20 (8)	28 (24)	0.009
RetainFirstHalf	0 (0)	17 (9)	$3.092 * 10^{-7}$
LessThan	4 (0)	9 (5)	0.076
Multiply	16 (8)	24 (22)	0.020
Negative	23 (15)	30 (27)	0.005
Pop	6 (1)	30 (26)	$1.646 * 10^{-11}$
KeepPositives	18 (3)	30 (27)	$6.181 * 10^{-5}$
KeepEvens	0 (0)	0 (0)	1.0
ArrayLength	29 (25)	30 (30)	0.5
ArrayToZero	29 (25)	30 (30)	0.5
KeepNegatives	18 (9)	29 (22)	$5.022 * 10^{-4}$
KeepOdds	0 (0)	1 (1)	0.5
Reverse	13 (8)	24 (21)	0.003
CatToSelf	3 (0)	21 (21)	$1.611 * 10^{-6}$
CatZerosToSelf	7 (2)	30 (24)	$8.705 * 10^{-11}$
ClipToMax	13 (10)	27 (18)	$1.159 * 10^{-4}$

Table 4. Full breakdown of all problems in the first corpus, 1D arrays, with success rates for 30 runs of both baseline (GP without TL enabled) and our Transfer Learning system. Two attempts were permitted for both approaches, and success counted if either attempt succeeded. Programs which were found to generalise to 1,000 additional random inputs are shown in brackets. Statistical significance established per problem by Fisher's Exact Test.

system can be considered to have near-certainly improved the performance, and which it showed limited success on.

For the unsolved problems, such as 'Sort' and 'Keep Evens', our system clearly did not improve upon the baseline at all, and the results were unchanged. Similarly for those which the baseline GP process solved with near 100% accuracy, such as the Identity function, there is not sufficient

data to indicate that our system had an influence, and that the results aren't due to pure chance. On many tasks the results are clear, however, that our approach has improved significantly on the performance of the baseline. We see multiple tasks, such as 'Shift Right' and 'Pop' which have probabilities of both results being drawn from the same distribution (that is to say the probability that the null hypothesis is wrong) of far less than our chosen limit of 0.01.

One core set of problems on which our approach was more successful was those which required the output array length to differ from that of the input array. Various problems of this class existed in the corpus, specifically "append" and "pop", which changed the length by 1, "retain first half" and "retain second half" which halve the length of the output array compared to the input's length, and "cat to self" and "cat zeros to self" which concatenate an array onto the end of the input array, and "first element only" which requires an output array of length 1. This suggests particular success of transferring useful material between problems which have similar traits.

In terms of generalisability of the synthesised code, we see similar increases in good solutions using our approach across each problem compared to the baseline, though both the baseline and our approach have fewer generalisable solutions than total solutions that work only on the given 10 I/O examples.

A similar detailed breakdown of problems from our canvas-based problem set is shown in Table 5. These results again demonstrate that our approach generally performs better than the baseline, indicating successful inference of which code fragments from solved problems are useful in unsolved ones. The overall success rates are more similar here than our previous problem set (i.e., across the total 30 runs, the number of programs found at least once by each approach), with the more significant difference being the frequency with which our approach finds a solution to a given problem. This suggests the most significant result on this corpus was in its ability to improve the baseline GP on already-findable problems, rather than allowing new problems to be found.

Again, in these results, we see that a significant number of solutions generalise to 1,000 additional input examples, and again this number of generalisable solutions is often significantly higher using our approach. In both problem sets, in fact, both approaches appear to be roughly equivalent in terms of their generalisability as a proportion of their solutions; the fact that our approach generates more solutions overall thereby increases its proportion of solutions which generalise. Fischer significance for these generalised results presented in Appendix D.

Computational Cost. Finally in this section we consider the overall computation cost of our baseline and our transfer-learning-based approach; although both experiments had the same number of primary GP generations available to solve each problem, our approach uses additional steps such as NN training. Measured in compute time, the first problem corpus took 15 hours for the baseline on our hardware, while the full transfer learning system took 24 hours. On the second corpus, the baseline took on average approximately 49 hours, while the TL system took 80 hours. The training time of the neural networks was not recorded specifically, but can be assumed to represent a non-negligible proportion of this additional time, as approximately 350 networks were trained each time, one per fragment. We also note that our hardware setup entailed NNs being trained using CPU resources, rather than dedicated high-performance GPU hardware; using GPUs for NN training may offer a significant time saving.

Summary. The results in this section clearly demonstrate the advantages of our transfer learning approach in this context, and the success on two distinct program domains suggests the approach has a degree of generality to it. We illustrate the kind of program generated by our system in Algorithm 1, demonstrating a solution to the 'abs' problem converted to Java code; while some of the program is certainly unusual, it does represent a generalised solution to the problem in question.

Problem	B' Success (g)	Success (g)	Fisher Sig.
Square	30 (24)	30 (26)	1.0
HollowSquare	30 (21)	30 (25)	1.0
Parallelogram	0 (0)	2 (1)	0.25
HollowParallelogram	1 (0)	2 (1)	0.5
MirroredParallelogram	11 (2)	13 (7)	0.2
MirroredHollowParallelogram	9 (2)	6 (6)	0.166
RightTriangle	30 (23)	30 (25)	1.0
HollowRightTriangle	30 (15)	30 (25)	1.0
MirroredRightTriangle	15 (8)	30 (21)	$2.916 * 10^{-6}$
HollowMirroredRightTriangle	12 (4)	27 (19)	$4.398 * 10^{-5}$
InvertedRightTriangle	29 (18)	30 (23)	0.5
HollowInvertedRightTriangle	15 (3)	24 (16)	0.011
InvertedMirroredRightTriangle	30 (24)	30 (25)	1.0
Inv'HollowMirr'RightTriangle	30 (14)	30 (25)	1.0
IsoceleseTriangle	0 (0)	2 (1)	0.25
HollowIsoceleseTriangle	1 (0)	7 (4)	0.024
InvertedIsoceleseTriangle	15 (6)	24 (18)	0.011
HollowInv'IsoceleseTriangle	9 (1)	15 (12)	0.062
RectangleWithEmptyTrapezoid	2 (2)	2 (0)	0.5
Inv'dRect'WithEmptyTrapezoid	0 (0)	7 (2)	0.005
ObtuseTriangle	4 (0)	9 (5)	0.076
HollowObtuseTriangle	10 (2)	16 (11)	0.066
MirroredObtuseTriangle	0 (0)	0 (0)	1.0
MirroredHollowObtuseTriangle	2 (0)	2 (0)	0.5
InvertedObtuseTriangle	0 (0)	1 (0)	0.5
HollowInvertedObtuseTriangle	1 (0)	4 (2)	0.166
InvertedMir'ObtuseTriangle	0 (0)	3 (0)	0.125
HollowMir'Inv'ObtuseTriangle	0 (0)	2 (1)	0.25
VShape	18 (2)	27 (18)	0.006
Trapezoid	0 (0)	2 (1)	0.25

Table 5. Full breakdown of all problems in the second corpus, 2D arrays representing images, with success rates for 30 runs of both baseline (GP without TL enabled) and transfer learning system. Two attempts were permitted for both approaches, and success counted if either attempt succeeded. Statistical significance established per problem by Fisher's Exact Test. (n=30)

In the following two subsections we examine two specific elements of our approach in more detail, to provide further context to our main results. We first examine how fitness curves over time appear for both the baseline and our system, to help understand how transfer learning has affected GP population fitness behaviour. We then examine the effect of injecting random large fragments of code into a GP process, to confirm that *NN-selected* fragments are the source of improvements seen in this section, rather than those improvements coming simply from the injection of larger fragments of code in general.

4.4 Fitness curves

In this section we examine how fitness-over-time plays out in both our baseline and our transfer-learning-based system, providing further insight into the particular affect of transferred material.

Algorithm 1 A translated example of solution produced to the 'abs' problem, generated by the GP process, a problem which gained a large performance increase by code fragment guidance (from 36% to 85%). (Translated from the internal language into Java) In this solution, the GP uses a loop as a conditional (since a loop will only execute its instruction block if the bounding variable is positive). It must iterate through each value, and write the negative of the value to the output array if the input array's value at that index is negative (thus rendering it positive). The GP accomplishes this by generating a negative, writing it to the output, then erasing it if the value read in was already positive, using a loop as a conditional. Inefficient but an effective solution to the problem as presented to it.

```
static int nArrays = 2;
static int nVars = 12;
public static int[] generatedProgram(
          int[] inputArray,int param){
            int[][] arrays = new int[nArrays][];
            arrays[0] = inputArray;
            int[] variables = new int[nVars];
            variables[0] = inputArray.length;
            variables[1] = param;
            arrays[1] = new int[variables[0]]
            for (variables[2]=0;variables[2]<variables[0];
            variables[2]++){
                variables[3] = arrays[0][2]
                variables[4] = arrays[0][2]
                variables[5] = variables[6] - variables[3]
                arrays[1][i] = variables[5]
                for (variables[7]=0;variables[7]<variables[4];
                variables[7]++){
                   arrays[1][i] = variables[4]
                }
            }
        }
return arrays[1];
```

Figure 2 shows the average and median fitness for all problems over time for our array-based problem set. Here we see a very similar overall shape between the two systems, but with notably higher fitness throughout for our approach (particularly clear in the median).

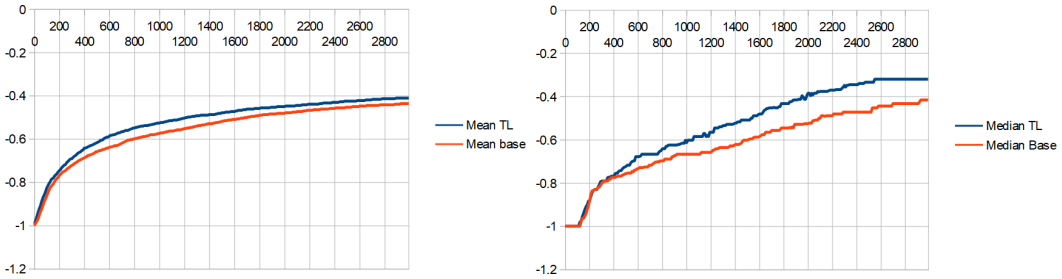

Fig. 2. Fitness curves for GP and TL systems on the first corpus, with mean on the left and median on the right. TL system in blue (higher curve both times). Y is the fitness, which reaches 0 when the problem is solved, and is normalised to a fitness of -1 if the sampled program is returning an all-zero array. Penalty value of -10,000 changed to -1 to allow mean to be viewed usefully. X is generation count. (n=900)

Figure 3 shows fitness over time for a specific problem ('Keep Positives') in our array-based problem set. In this particular one, our approach on the right shows interesting behaviour in the median graph. Our approach had a success rate of 100% on this problem, and the median graph shows a dramatic jump around generation 500. This implies that certain programmatic elements were found which allowed fitness to be increased very rapidly. The baseline system, by comparison, does not have this characteristic. Our system, working on this problem, also has a much stronger early increase than the baseline GP alone, with a high slope from generation 0. This, too, is likely due to genetic material being transferred from previous problems, such as read and writing to and from the input and output arrays. Further work would need to be done to analyse exactly how code changes over time, and fragment usage correlates and corresponds to these fitness changes, but this data is a useful confirmation of the expected overall behaviour.

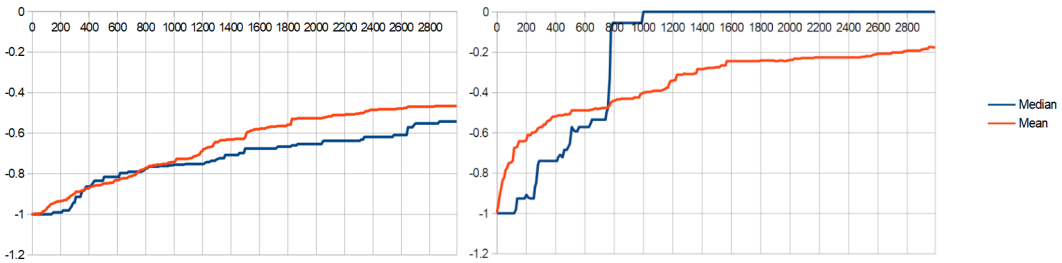

Fig. 3. Fitness curves on the 21st problem of the 1D Array corpus, 'Keep Positives'. The left graph represents the median and mean from non-TL GP runs, with the right from the TL runs (median in blue, ends lower on the non-TL runs, higher on TL). Y is the fitness, which reaches 0 when the problem is solved, and is normalised to a fitness of -1 if the sampled program is returning an all-zero array. Penalty value of -10,000 changed to -1 to allow mean to be viewed usefully. X is generation count. (n=30)

Figure 4 shows a problem from the second corpus, the 2D arrays/canvas set representing geometric images. This one also shows a stronger early trend, especially on the median graph, where the GP system spends time with fewer than 50% of the population elite individuals able to write anything usefully to the output array (fitness is at -1, indicating it has not achieved anything better than a program returning an all-zero 2D array would achieve). Our approach by comparison has clearly provided the GP system with a good starting point. This problem is interesting, however, in that it had lower overall performance with our approach than the baseline GP, and this appears

to be visible in the graphs, with our approach achieving a lower fitness towards the end of the run. It could be hypothesised that our approach transferred data which may initially have boosted fitness, but lead to the GP process becoming trapped in a local minimum which did not lead to useful progress towards a 0-fitness solution.

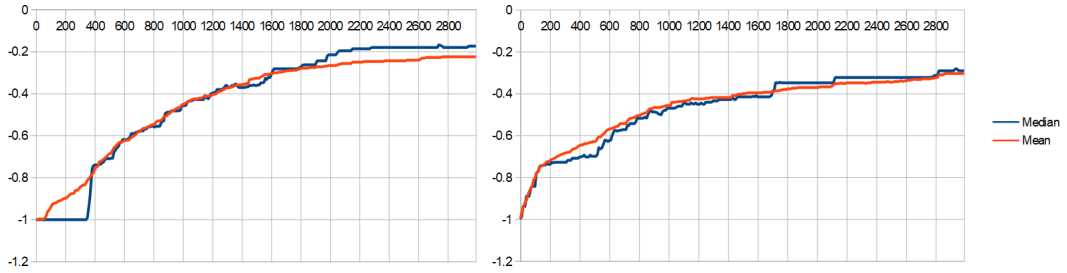

Fig. 4. Fitness curves on the 6th problem of the 2D Array corpus, 'Mirrored Hollowed Parallelogram'. The left graph represents the median and mean from non-TL GP runs, with the right from the TL runs (median in blue, ends higher on the non-TL runs, TL median has more step-function-like jumps in the curve). Y is the fitness, which reaches 0 when the problem is solved, and is normalised to a fitness of -1 if the sampled program is returning an all-zero array. Penalty value of -10,000 changed to -1 to allow mean to be viewed usefully. X is generation count. (n=30)

Problem Set	**Mean gens. to threshold (std. dev)**	**Mean transfer Ratio (std. dev)**
1D-array	1439.76 (848.74)	0.75 (0.25)
2D-array / canvas	1414.12 (878.64)	0.62 (0.31)

Table 6. Average transfer learning metrics of generations-to-threshold and transfer ratio, for each of our two problem sets.

Beyond the evidence of learning enhancements shown by fitness curves, existing research into transfer learning has also suggested a set of specific metrics to help understand the effects of transferred information [36]. Considering the specific way in which transfer works in our approach, the most suitable metric from this set is the 'time the threshold', which reports how many generations it takes for the average fitness of the TL system to exceed the average fitness the baseline reaches after the full 3000 generations. As part of this measure we also report the 'transfer ratio', which computes the average fitness of the TL system divided by the average fitness of the baseline. We show the average values for this metrics across all problems in each problem set in Tables 6 (with a full per-problem breakdown provided in Appendix C). On both measures we see clear evidence of the benefits of transfer learning; our approach reaches the best fitness of the baseline in less than half the number of generations on average, and shows a clear relative improvement in fitness over the baseline.

4.5 Analysis of Large Mutations

Finally, we examine whether *NN-suggested* specific fragments are likely to be the real source of improvements, or whether this may be simply a factor of inserting larger fragments of code (up to 4 lines at a time) in general. To do this, we ran a follow-up experiment which simulates random large fragments of code being added as insert mutations. These fragments are not recommended by an NN, but instead are generated at random each time the 'fragment injection' mutation is called.

Problem	Rand Success	B' Success	Success	Fisher Sig.
Add	46%	47%	83%	0.003
Append	33%	0%	93%	$3.300 * 10^{-6}$
CumulativeAbsoluteSum	0%	7%	40%	$2.522 * 10^{-4}$
CumulativeSum	8%	17%	70%	$3.636 * 10^{-6}$
KeepEvenIndices	8%	23%	100%	$3.536 * 10^{-13}$
ClipToMin	46%	53%	90%	$4.805 * 10^{-4}$
RetainSecondHalf	0%	80%	33%	0.001
Sort	0%	0%	0%	1.0
Subtract	71%	33%	83%	0.166
Abs	46%	37%	73%	0.028
GreaterThan	4%	10%	17%	0.142
IndexParity	100%	97%	100%	1.0
FirstElementOnly	100%	33%	100%	1.0
Identity	100%	97%	100%	1.0
DivergentSequence	4%	40%	93%	$6.202 * 10^{-12}$
Double	79%	47%	100%	0.013
ShiftRight	17%	0%	87%	$2.076 * 10^{-7}$
ShiftRightLossy	21%	60%	100%	$2.314 * 10^{-10}$
ShiftLeft	21%	17%	100%	$2.314 * 10^{-10}$
ShiftLeftZeroPadded	88%	67%	93%	0.333
RetainFirstHalf	13%	0%	57%	$7.547 * 10^{-4}$
LessThan	4%	13%	30%	0.014
Multiply	58%	53%	80%	0.055
Negative	75%	77%	100%	0.005
Pop	46%	20%	100%	$2.252 * 10^{-6}$
KeepPositives	38%	60%	100%	$1.510 * 10^{-7}$
KeepEvens	0%	0%	0%	1.0
ArrayLength	100%	97%	100%	1.0
ArrayToZero	100%	97%	100%	1.0
KeepNegatives	75%	60%	97%	0.023
KeepOdds	0%	0%	3%	1.0
Reverse	54%	43%	80%	0.032
CatToSelf	92%	10%	70%	0.041
CatZerosToSelf	100%	23%	100%	1.0
ClipToMax	42%	43%	90%	$1.688 * 10^{-4}$

Table 7. Comparison of random fragments (Rand) generated when the 'inject fragment' mutation is called and with the full TL results, for the first corpus (1D arrays). As this is a secondary experiment, note the random fragments only were run for 24 runs. Statistical significance established per problem by Fisher's Exact Test. (n=24, n=30)

They are all of maximum length (to avoid one-line fragments being generated which would be no different than the pre-existing 'inject' mutation which injects as single line of code).

As can be seen in Table 7, there is a clear statistical difference between the use of randomly generated fragments and deployment of fragments by the NN, which were sourced from previous successes. The random-fragment approach used here achieved an average success rate of 46%, compared to the TL system's 76%; the use of random fragments is, however, slightly better than the

performance of the baseline GP. Comparison to the baseline GP indicates that certain problems were benefited by this new mutation strategy. Specifically, a number of those which required the array length to be changed saw large improvements to their performance. This could be hypothesised to be a result of the fitness function, which does not have a shaped landscape towards the correct array length, but rather simply awards a penalty if the output array is of the wrong length. In this case, the fitness function would be unable to guide navigation through problem space, and the much larger jumps performed by this mutation operation appear to lead to success.

Statistical differences between the two were even larger in the second corpus, the 2D arrays representing geometric images, with the new baseline achieving a success rate of 31%, which is lower than those seen in the GP baseline, and far lower than those seen in the our approach. This suggests that for this corpus the larger mutations were harmful, and that our approach strongly benefited from more targetted fragments being injected. Full results are seen in Tables 8, with eight problems demonstrating statistically significant improvements over this secondary baseline by the TL system.

We can conclude from this that the larger mutations assisted on some problems (and harmed on others), but were not solely contributory towards the success seen on the full end-to-end transfer learning system.

Problem	Rand Success	B' Success	Success	Fisher Sig.
Square	100%	100%	100%	1.0
HollowSquare	100%	100%	100%	1.0
Parallelogram	4%	0%	7%	0.5
HollowParallelogram	0%	3%	7%	0.333
MirroredParallelogram	20%	37%	43%	0.045
MirroredHollowParallelogram	0%	30%	20%	0.020
RightTriangle	100%	100%	100%	1.0
HollowRightTriangle	96%	100%	100%	0.5
MirroredRightTriangle	48%	50%	100%	$3.583 * 10^{-6}$
HollowMirroredRightTriangle	28%	40%	90%	$2.318 * 10^{-6}$
InvertedRightTriangle	96%	97%	100%	0.5
HollowInvertedRightTriangle	28%	50%	80%	$1.147 * 10^{-4}$
InvertedMirroredRightTriangle	100%	100%	100%	1.0
Inv'HollowMirr'RightTriangle	100%	100%	100%	1.0
IsoceleseTriangle	0%	0%	7%	0.333
HollowIsoceleseTriangle	0%	3%	23%	0.010
InvertedIsoceleseTriangle	32%	50%	80%	$3.441 * 10^{-4}$
HollowInv'IsoceleseTriangle	24%	30%	50%	0.033
RectangleWithEmptyTrapezoid	0%	7%	7%	0.333
Inv'dRect'WithEmptyTrapezoid	0%	0%	23%	0.010
ObtuseTriangle	8%	13%	30%	0.037
HollowObtuseTriangle	8%	33%	53%	$3.028 * 10^{-4}$
MirroredObtuseTriangle	4%	0%	0%	0.5
MirroredHollowObtuseTriangle	0%	7%	7%	0.333
InvertedObtuseTriangle	0%	0%	3%	1.0
HollowInvertedObtuseTriangle	0%	3%	13%	0.083
InvertedMir'ObtuseTriangle	0%	0%	10%	0.166
HollowMir'Inv'ObtuseTriangle	0%	0%	7%	0.333
VShape	24%	60%	90%	$5.527 * 10^{-7}$
Trapezoid	0%	0%	7%	0.333

Table 8. Comparison of random fragments (Rand) generated when the 'inject fragment' mutation is called and with the full TL results for the second corpus (2D arrays). As this is a secondary experiment, note the random fragments only were run for 25 runs. Statistical significance established per problem by Fisher's Exact Test. (n=25, n=30)

5 CONCLUSION

In this paper we have presented a novel mechanism for performing transfer learning in GP. We study its core components, then demonstrate an end-to-end system in operation.

Our framework demonstrates a way to create a system which searches for solutions to problems within a corpus, finds a subset, extracts code fragments from the successes, then trains a NN to recognise the presence of these fragments and determines which unsolved problems would benefit from NN-guidance – thus boosting GP success rates on a subsequent pass. We demonstrate that this process can render previously unfindable problems findable, and boost overall success rates to a large degree.

One area which may well prove fruitful for further exploration is that of how fragments are selected from the generated programs. While we select a number from each program, there are far

more which are simply discarded. Our initial heuristic, to select only fragments which do not have variables which depend on prior lines of code, was successful in cutting down the possible options, but may well have discarded useful fragments. Testing alternative fragment extraction approaches may lead to superior results.

We are confident that this solution is broadly applicable to the field of GP, and that this initial architecture can be refined and expanded upon. We are hopeful that future work into fragment selection and deployment can further boost performance, allowing reliable gains in any GP system which will find itself faced with multiple problems.

REFERENCES

[1] Qurrat Ul Ain, Harith Al-Sahaf, Bing Xue, and Mengjie Zhang. 2020. A Genetic Programming Approach to Feature Construction for Ensemble Learning in Skin Cancer Detection. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 1186–1194. https://doi.org/10.1145/3377930.3390228

[2] Matej Balog, Alexander Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *Proceedings of ICLR'17* (proceedings of iclr'17 ed.). https://www.microsoft.com/en-us/research/publication/deepcoder-learning-write-programs/

[3] Peter L. Bartlett, Nick Harvey, Christopher Liaw, and Abbas Mehrabian. 2019. Nearly-tight VC-dimension and Pseudodimension Bounds for Piecewise Linear Neural Networks. *Journal of Machine Learning Research* 20, 63 (2019), 1–17. http://jmlr.org/papers/v20/17-612.html

[4] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276* (2018).

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[6] Qi Chen, Bing Xue, and Mengjie Zhang. 2020. Genetic Programming for Instance Transfer Learning in Symbolic Regression. *IEEE Transactions on Cybernetics* PP (02 2020), 1–14. https://doi.org/10.1109/TCYB.2020.2969689

[7] X. Chen, C. Liu, and D. Song. 2019. Execution-Guided Neural Program Synthesis. In *ICLR*.

[8] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70 (ICML'17)*. JMLR.org, 990–998.

[9] Stephane Doncieux, Alban Laflaquière, and Alexandre Coninx. 2019. Novelty Search: A Theoretical Perspective. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 99–106. https://doi.org/10.1145/3321707.3321752

[10] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2017. A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming. In *EuroGP*.

[11] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2018. Extending Program Synthesis Grammars for Grammar-Guided Genetic Programming. In *PPSN*.

[12] Richard Forsyth. 1981. BEAGLE A Darwinian Approach to Pattern Recognition. *Kybernetes* 10, 3 (1981), 159–166. https://doi.org/doi:10.1108/eb005587

[13] David E Goldberg, Jon Richardson, et al. 1987. Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, Vol. 4149. Hillsdale, NJ: Lawrence Erlbaum.

[14] Thomas Helmuth, Edward Pantridge, Grace Woolson, and Lee Spector. 2020. Genetic Source Sensitivity and Transfer Learning in Genetic Programming *(ALIFE 2021: The 2021 Conference on Artificial Life)*, Vol. ALIFE 2020: The 2020 Conference on Artificial Life. 303–311. https://doi.org/10.1162/isal_a_00326 arXiv:https://direct.mit.edu/isal/proceedings-pdf/isal2020/32/303/1908486/isal_a_00326.pdf

[15] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15)*. Association for Computing Machinery, New York, NY, USA, 1039–1046. https://doi.org/10.1145/2739480.2754769

[16] Erik Hemberg, Jonathan Kelly, and Una-May O'Reilly. 2019. On Domain Knowledge and Novelty to Improve Program Synthesis Performance with Grammatical Evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 1039–1046. https://doi.org/10.1145/3321707.3321865

[17] Wojciech Jaskowski, Krzysztof Krawiec, and Bartosz Wieloch. 2007. Knowledge Reuse in Genetic Programming Applied to Visual Learning. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO '07)*. Association for Computing Machinery, New York, NY, USA, 1790–1797. https://doi.org/10.1145/1276958.1277318

[18] Maarten Keijzer, Conor Ryan, and Mike Cattolico. 2004. Run Transferable Libraries — Learning Functional Bias in Problem Domains. In *Genetic and Evolutionary Computation – GECCO 2004*, Kalyanmoy Deb (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 531–542.

[19] Stephen Kelly and Malcolm Heywood. 2015. Knowledge Transfer from Keepaway Soccer to Half-field Offense through Program Symbiosis. 1143–1150. https://doi.org/10.1145/2739480.2754798

[20] Stephen Kelly and Malcolm I. Heywood. 2018. Discovering Agent Behaviors Through Code Reuse: Examples From Half-Field Offense and Ms. Pac-Man. *IEEE Transactions on Games* 10, 2 (2018), 195–208. https://doi.org/10.1109/TCIAIG.2017.2766980

[21] K. E. Kinnear. 1994. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. 142–147 vol.1.

[22] Joel Lehman and Kenneth Stanley. 2010. Efficiently evolving programs through the search for novelty. *Proceedings of the 12th Annual Genetic and Evolutionary Computation Conference, GECCO '10*, 837–844. https://doi.org/10.1145/1830483.1830638

[23] B. Miller and D. Goldberg. 1995. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Syst.* 9 (1995).

[24] Ícaro Marcelino Miranda, Claus Aranha, and Marcelo Ladeira. 2019. Classification of EEG Signals Using Genetic Programming for Feature Construction. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 1275–1283. https://doi.org/10.1145/3321707.3321737

[25] Brandon Muller, Harith Al-Sahaf, Bing Xue, and Mengjie Zhang. 2019. Transfer learning: a building block selection mechanism in genetic programming for symbolic regression. 350–351. https://doi.org/10.1145/3319619.3322072

[26] Luis Muñoz, Leonardo Trujillo, and Sara Silva. 2020. Transfer learning in constructive induction with Genetic Programming. *Genetic Programming and Evolvable Machines* 21 (12 2020). https://doi.org/10.1007/s10710-019-09368-y

[27] Gonzalo Navarro. 2001. A Guided Tour to Approximate String Matching. *ACM Comput. Surv.* 33, 1 (mar 2001), 31–88. https://doi.org/10.1145/375360.375365

[28] Damien O'Neill, Harith Al-Sahaf, Bing Xue, and Mengjie Zhang. 2017. Common subtrees in related problems: A novel transfer learning approach for genetic programming. 1287–1294. https://doi.org/10.1109/CEC.2017.7969453

[29] Edward Pantridge and Lee Spector. 2020. Code building genetic programming. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO '20)*. 994–1002. https://doi.org/10.1145/3377930.3390239

[30] Joseph Renzullo, Westley Weimer, Melanie Moses, and Stephanie Forrest. 2018. Neutrality and Epistasis in Program Space. In *Proceedings of the 4th International Workshop on Genetic Improvement Workshop (GI '18)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3194810.3194812

[31] Rishabh Singh and Pushmeet Kohli. 2017. AP: Artificial Programming. *Snapl '17* 16 (2017), 1–12. https://doi.org/10.4230/LIPIcs.SNAPL.2017.16 arXiv:1703.08694

[32] Sunbeom So and Hakjoo Oh. 2018. Synthesizing Pattern Programs from Examples. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 1618–1624. https://doi.org/10.24963/ijcai.2018/224

[33] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. Association for Computing Machinery, New York, NY, USA, 313–326. https://doi.org/10.1145/1706299.1706337

[34] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. *SIGPLAN Not.* 45, 1 (Jan. 2010), 313–326. https://doi.org/10.1145/1707801.1706337

[35] Milad Taleby Ahvanooey, Qianmu Li, Ming Wu, and Shuo Wang. 2019. A Survey of Genetic Programming and Its Applications. *KSII Transactions on Internet and Information Systems* Vol.13 (04 2019), 1765–1793. https://doi.org/10.3837/tiis.2019.04.002

[36] Matthew E. Taylor and Peter Stone. 2011. An Introduction to Inter-task Transfer for Reinforcement Learning. *AI Magazine* 32, 1 (2011), 15–34.

[37] Leonardo Vanneschi and Steven Gustafson. 2009. Using Crossover Based Similarity Measure to Improve Genetic Programming Generalization Ability. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09)*. Association for Computing Machinery, New York, NY, USA, 1139–1146. https://doi.org/10.1145/1569901.1570054

[38] Leonardo Vanneschi and Riccardo Poli. 2012. *Genetic Programming — Introduction, Applications, Theory and Open Issues*. Springer Berlin Heidelberg, Berlin, Heidelberg, 709–739. https://doi.org/10.1007/978-3-540-92910-9_24

[39] Alexander Wild and Barry Porter. 2019. General Program Synthesis using Guided Corpus Generation and Automatic Refactoring. In *Search-Based Software Engineering (Lecture Notes in Computer Science)*, Shiva Nejati and Gregory Gay (Eds.). Springer-Verlag, 89–104. https://doi.org/10.1007/978-3-030-27455-9_7

[40] Amit Zohar and Lior Wolf. 2018. Automatic Program Synthesis of Long Programs with a Learned Garbage Collector. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 2094–2103.

A LANGUAGE OPERATORS

Our target programming language uses the operators shown below, and can be automatically translated to C/Java code (Java equivalents are shown for each operator).

Operator	Java Equivalent
No op (no operation performed)	
Assign Variable To Array	array[PARAM_A] = $var
Assign Variable From Array	$var = array[PARAM_A]
Make Array	array = new int[$var]
Variable To Literal (values are one of -1,0,1,2)	$var = LITERAL
Add	$varC = $varA + $varB (A,B,C may be identical here and elsewhere)
Subtract	$varC = $varA - $varB
Multiply	$varC = $varA * $varB
Divide	if (varB != 0) { $varC = $varA / $varB}
Modulo	if (varB != 0) { $varC = $varA % $varB}
Assign Var from Var	$varA = $varB
Loop	for ($varA=0;$varA<$varB;$varA++){
Conditional (var > 0)	if ($varA>0){
Conditional (var1 == var2)	if ($varA==$varB){
Conditional (var1 > var2)	if ($varA>$varB){
End Block	}
Else	}else{
Create 2D Array	canvas = new int[$varA][$varA]
Get 2D Array Size	$varA = canvas.length
Read from XY Point from 2D Array	$varC = canvas[$varA][$varB]
Set 2D Array to 0 at XY Point	canvas[$varA][$varB]=0
Set 2D Array to 1 at XY Point	canvas[$varA][$varB]=1

Table 9. Operators employed in the language, all available for the GP to deploy, with the exception of 2D array operations if the problem does not employ this data structure.

B CORPUS 1 PROBLEMS

Problem	Description
Add	Adds the supplied integer to every value in the input array
Append	Extends the array by adding the input integer to its end
CumulativeAbsoluteSum	Creates a new array of equal length to the input array. Iterates through the input array, summing the absoluate of each value and writing the current sum into the new, output, array
CumulativeSum	Creates a new array of equal length to the input array. Iterates through the input array, summing all values and writing the current sum into the new, output, array
KeepEvenIndices	Returns the input array with all odd-numbered indices set to zero
ClipToMin	Returns the input array with all values less than the input integer set to the input integer
RetainSecondHalf	Returns an array of half the length of the input array, which contains the second half of its values
Sort	Returns the input array, but with all the values in numeric order
Subtract	Subtracts the supplied integer from every value in the input array
Abs	Returns the input array, but with every value set to its absolute
GreaterThan	Returns an array of equal length to the input array, with a value of 1 if the input array's value is greater than the input integer, 0 if it is not
IndexParity	Returns an array of equal length to the input array, filled with alternating 0s and 1s, starting with 0
FirstElementOnly	Returns one-element array, with its element set to the input integer
Identity	Returns a copy of the input array
DivergentSequence	Returns an array of equal length to the input array with values following the sequence [1,-1,2,-2,3,-3...]
Double	Returns an array with all its value equal to twice that of the input array's corresponding value
ShiftRight	Returns a copy of the input array extended by one element, with values transposed by one to the right, leaving a zero as first value
ShiftRightLossy	Returns a copy of the input array, with values transposed by one to the right, losing the last value and leaving a zero as first value

Table 10. Descriptions of all functions used in the first corpus. (Part 1)

Problem	Description
ShiftLeft	Returns a copy of the input array reduced by one element, with values transposed by one to the left, losing the first value
ShiftLeftZeroPadded	Returns a copy of the input array, with values transposed by one to the left, losing the first value and leaving a zero as last value
RetainFirstHalf	Returns an array of half the length of the input array, which contains the first half of its values
LessThan	Returns an array of equal length to the input array, with a value of 1 if the input array's value is less than the input integer, 0 if it is not
Multiply	Returns the input array with all its elements multiplied by the input integer
Negative	Returns the input array with the input integer subtracted from all its values
Pop	Returns the array without the first element (reduces length)
KeepPositives	Returns the input array with all negative values set to zero
KeepEvens	Returns the input array with all odd values set to zero
ArrayLength	Returns an array of equal length to the input array with all values zero, except the first which is equal to its length
ArrayToZero	Returns an array of equal length to the input array with all values zero
KeepNegatives	Returns the input array with all positive values set to zero
KeepOdds	Returns the input array with all even values set to zero
Reverse	Returns the input array in reverse order
CatToSelf	Returns an array consisting of two copies of the input array concatenated to one another
CatZerosToSelf	Returns a copy of the input array, concatenated to an array of all zeros of equal length to its end
ClipToMin	Returns the input array with all values greater than the input integer set to the input integer

Table 11. Descriptions of all functions used in the first corpus. (Part 2)

C TRANSFER LEARNING METRICS IN DETAIL

In Section 4.4 we reported averaged metrics for transfer learning; here we report the per-problem metrics for time-to-threshold and transfer ratio, in Tables 12 and 13.

Problem	Generations to threshold	Transfer Ratio
Add	2388.8	0.89
Append	0.0	0.95
CumulativeAbsoluteSum	2346.4	1.00
CumulativeSum	2205.2	0.89
KeepEvenIndices	1888.8	0.66
ClipToMin	1932.4	0.70
RetainSecondHalf	0.0	0.93
Sort	2277.6	0.98
Subtract	1996.8	0.71
Abs	1855.2	0.76
GreaterThan	2362.4	1.39
IndexParity	774.8	0.46
FirstElementOnly	0.0	0.79
Identity	604.4	0.42
DivergentSequence	1846.8	0.72
Double	1322.8	0.55
ShiftRight	0.0	0.92
ShiftRightLossy	1404.0	0.55
ShiftLeft	948.8	0.52
ShiftLeftZeroPadded	1825.2	0.76
RetainFirstHalf	0.0	0.92
LessThan	2718.8	1.35
Multiply	1818.8	0.69
Negative	1386.8	0.49
Pop	1257.6	0.49
KeepPositives	1499.2	0.63
KeepEvens	2314.4	0.92
ArrayLength	0.0	0.24
ArrayToZero	0.0	0.22
KeepNegatives	1856.4	0.74
KeepOdds	1783.6	0.94
Reverse	1504.8	0.58
CatToSelf	2111.2	0.81
CatZerosToSelf	2242.4	0.80
ClipToMax	1917.2	0.86

Table 12. Transfer Learning metrics for 25 runs of the TL system compared to the baseline for the first corpus. Generations to Threshold represents the average number of generations take for the TL system to outperform the baseline's asymptotic performance (TL will gain performance after this point). Transfer Ratio is the average fitness of the TL system divided by the average fitness of the baseline (fitnesses less than -1 set to -1).(n=25)

Problem	Generations to threshold	Transfer Ratio
Square	211.0	0.12
HollowSquare	1060.9	0.27
Parallelogram	1158.3	0.57
HollowParallelogram	2129.4	0.87
MirroredParallelogram	1830.6	0.75
MirroredHollowParallelogram	1968.1	0.72
RightTriangle	152.0	0.07
HollowRightTriangle	1003.0	0.18
MirroredRightTriangle	787.5	0.40
HollowMirroredRightTriangle	1605.4	0.59
InvertedRightTriangle	271.0	0.14
HollowInvertedRightTriangle	2458.2	0.80
InvertedMirroredRightTriangle	149.0	0.06
Inv'HollowMirr'RightTriangle	792.7	0.24
IsoceleseTriangle	0.0	0.89
HollowIsoceleseTriangle	2662.2	0.91
InvertedIsoceleseTriangle	1338.5	0.52
HollowInv'IsoceleseTriangle	1851.3	0.64
RectangleWithEmptyTrapezoid	2693.9	1.20
Inv'dRect'WithEmptyTrapezoid	1952.4	0.95
ObtuseTriangle	1832.9	0.74
HollowObtuseTriangle	2556.0	0.96
MirroredObtuseTriangle	465.3	0.48
MirroredHollowObtuseTriangle	2140.6	0.78
InvertedObtuseTriangle	0.0	0.95
HollowInvertedObtuseTriangle	2513.9	0.93
InvertedMir'ObtuseTriangle	1981.7	0.95
HollowMir'Inv'ObtuseTriangle	2273.9	0.95
VShape	2022.1	0.63
Trapezoid	561.8	0.45

Table 13. Transfer Learning metrics for 25 runs of the TL system compared to the baseline for the second corpus. Generations to Threshold represents the average number of generations take for the TL system to outperform the baseline's asymptotic performance (TL will gain performance after this point). Transfer Ratio is the average fitness of the TL system divided by the average fitness of the baseline (fitnesses less than -1 set to -1).(n=25)

D FISHER SIGNIFICANCE FOR GENERALISED RESULTS

Problem	B' Success	Success	Fisher Significance
Add	7	17	0.006
Append	28	15	$1.742 * 10^{-4}$
CumulativeAbsoluteSum	1	5	0.090
CumulativeSum	4	14	0.004
KeepEvenIndices	2	23	$1.705 * 10^{-8}$
ClipToMin	2	17	$2.547 * 10^{-5}$
RetainSecondHalf	0	3	0.125
Sort	0	0	1.0
Subtract	5	23	$2.797 * 10^{-6}$
Abs	5	19	$2.159 * 10^{-4}$
GreaterThan	2	5	0.166
IndexParity	22	29	0.011
FirstElementOnly	3	23	$1.182 * 10^{-7}$
Identity	25	30	0.026
DivergentSequence	0	19	$2.671 * 10^{-8}$
Double	4	28	$1.149 * 10^{-10}$
ShiftRight	0	16	$9.720 * 10^{-7}$
ShiftRightLossy	8	28	$7.062 * 10^{-8}$
ShiftLeft	2	24	$3.695 * 10^{-9}$
ShiftLeftZeroPadded	8	24	$3.350 * 10^{-5}$
RetainFirstHalf	0	9	$9.680 * 10^{-4}$
LessThan	0	5	0.026
Multiply	8	22	$2.896 * 10^{-4}$
Negative	15	27	$6.811 * 10^{-4}$
Pop	1	26	$9.342 * 10^{-12}$
KeepPositives	3	27	$1.393 * 10^{-10}$
KeepEvens	0	0	1.0
ArrayLength	25	30	0.026
ArrayToZero	25	30	0.026
KeepNegatives	9	22	$7.320 * 10^{-4}$
KeepOdds	0	1	0.5
Reverse	8	21	$7.320 * 10^{-4}$
CatToSelf	0	21	$1.791 * 10^{-9}$
CatZerosToSelf	2	24	$3.695 * 10^{-9}$
ClipToMax	10	18	0.025

Table 14. Full breakdown of all problems in the first corpus, 1D arrays, with the values shown being those cases which passed generalisation on 1000 examples. Fisher significance calculated for these two success rates per problem.

Problem	B' Success	Success	Fisher Significance
Square	24	30	0.011
HollowSquare	21	30	$9.680 * 10^{-4}$
Parallelogram	0	2	0.25
HollowParallelogram	0	2	0.25
MirroredParallelogram	2	13	$9.794 * 10^{-4}$
MirroredHollowParallelogram	2	6	0.111
RightTriangle	23	30	0.005
HollowRightTriangle	15	30	$2.916 * 10^{-6}$
MirroredRightTriangle	8	30	$4.135 * 10^{-10}$
HollowMirroredRightTriangle	4	27	$9.721 * 10^{-10}$
InvertedRightTriangle	18	30	$6.181 * 10^{-5}$
HollowInvertedRightTriangle	3	24	$2.739 * 10^{-8}$
InvertedMirroredRightTriangle	24	30	0.011
Inv'HollowMirr'RightTriangle	14	30	$9.720 * 10^{-7}$
IsoceleseTriangle	0	2	0.25
HollowIsoceleseTriangle	0	7	0.005
InvertedIsoceleseTriangle	6	24	$2.981 * 10^{-6}$
HollowInv'IsoceleseTriangle	1	15	$3.110 * 10^{-5}$
RectangleWithEmptyTrapezoid	2	2	0.5
Inv'dRect'WithEmptyTrapezoid	0	7	0.005
ObtuseTriangle	0	9	$9.680 * 10^{-4}$
HollowObtuseTriangle	2	16	$6.839 * 10^{-5}$
MirroredObtuseTriangle	0	0	1.0
MirroredHollowObtuseTriangle	0	2	0.25
InvertedObtuseTriangle	0	1	0.5
HollowInvertedObtuseTriangle	0	4	0.058
InvertedMir'ObtuseTriangle	0	3	0.125
HollowMir'Inv'ObtuseTriangle	0	2	0.25
VShape	2	27	$1.543 * 10^{-11}$
Trapezoid	0	2	0.25

Table 15. Full breakdown of all problems in the second corpus, 2D arrays, with the values shown being those cases which passed generalisation on 1000 examples. Fisher significance calculated for these two success rates per problem.

E FULL BREAKDOWN OF FRAGMENTS EVALUATED IN BASELINE GP FRAGMENT SUGGESTION EXPERIMENTS

The tables in this section show each fragment evaluated by the exhaustive fragment testing process. Each fragment is at most two lines, and has no variables which depend on being set in lines outside the fragment (therefore the fragment stands alone in terms of functionality). The source code of the ground-truth implementation is given, firstly as simply the operator used on that line, and secondly in a C-like fashion (excluding braces). This C-like fashion is a programmatically generated translation of the source code of the custom language implementation, provided for ease of readability (due to the difficult-to-parse structure of the custom language). We then refer to the lines in this source code by line number. Fragments cannot contain end-of-block operators (used to indicate the end point of blocks started by the flow-control operators loop and conditional), nor can they contain the initial definition of the 2D canvas.

Line	Operator	As Code
1	Literal	variables[6] = 1;
2	Add	variables[7] = variables[0] + variables[6];
3	Make Array	arrays[1] = new int[vars[7]]
4	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
5	Read	variables[5] = arrays[0][variables[2]];
6	Write	arrays[1][variables[2]] = variables[5];
7	Endloop	
8	Write	arrays[1][variables[2]] = variables[1];

Fragment	Success Rate
1	3%
1, 2	27%
1, 4	10%
4	0%
4, 5	0%
4, 6	0%
4, 8	0%

Table 16. Fragments assessed from program “Append". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Make Array	arrays[1] = new int[vars[0]]
2	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
3	Literal	variables[5] = -1;
4	Read	variables[3] = arrays[0][variables[2]];
5	Condition	if (variables[3]>0)
6	Else	else
7	Multiply	variables[3] = variables[3] * variables[5];
8	Endloop	
9	Add	variables[4] = variables[4] + variables[3];
10	Write	arrays[1][variables[2]] = variables[4];
11	Endloop	

Fragment	Success Rate
1	0%
1, 2	0%
1, 3	0%
1, 5	0%
1, 6	0%
2	0%
2, 3	0%
2, 4	3%
2, 5	3%
2, 6	0%
3	0%
3, 5	0%
3, 6	0%
3, 7	0%
5	0%
5, 6	3%
6	0%

Table 17. Fragments assessed from program “Cumulative Absolute Sum". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Literal	variables[4] = 2;
2	Make Array	arrays[1] = new int[vars[0]]
3	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
4	Read	variables[3] = arrays[0][variables[2]];
5	Modulo	variables[5] = variables[3] % variables[4];
6	Condition	if (variables[5]==variables[6])
7	Write	arrays[1][variables[2]] = variables[3];
8	Endloop	
9	Endloop	

Fragment	Success Rate
1	0%
1, 2	6%
1, 3	3%
1, 5	3%
2	3%
2, 3	0%
3	0%
3, 4	0%
3, 7	0%

Table 18. Fragments assessed from program “Keep Evens". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Literal	variables[6] = 2;
2	Divide	variables[3] = variables[0] / variables[6];
3	Make Array	arrays[1] = new int[vars[3]]
4	Loop	for (variables[2]=0;variables[2]<variables[3];variables[2]++)
5	Read	variables[5] = arrays[0][variables[2]];
6	Write	arrays[1][variables[2]] = variables[5];
7	Endloop	

Fragment	Success Rate
1	0%
1, 2	13%

Table 19. Fragments assessed from program “Retain First Half". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Literal	variables[7] = 2;
2	Make Array	arrays[1] = new int[vars[0]]
3	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
4	Subtract	variables[6] = variables[0] - variables[2];
5	Subtract	variables[6] = variables[6] - variables[7];
6	Read	variables[5] = arrays[0][variables[6]];
7	Write	arrays[1][variables[2]] = variables[5];
8	Endloop	

Fragment	Success Rate
1	63%
1, 2	80%
1, 3	77%
2	73%
2, 3	60%
3	63%
3, 4	80%
3, 7	77%

Table 20. Fragments assessed from program “Reverse". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Literal	variables[6] = 1;
2	Add	variables[8] = variables[0] + variables[6];
3	Make Array	arrays[1] = new int[vars[8]]
4	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
5	Add	variables[7] = variables[2] + variables[6];
6	Read	variables[5] = arrays[0][variables[2]];
7	Write	arrays[1][variables[7]] = variables[5];
8	Endloop	

Fragment	Success Rate
1	3%
1, 2	13%
1, 4	20%
4	0%
4, 6	0%

Table 21. Fragments assessed from program “Shift Right". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Literal	variables[6] = 2;
2	Add	variables[8] = variables[0] + variables[6];
3	Make Array	arrays[1] = new int[vars[0]]
4	Subtract	variables[9] = variables[0] - variables[6];
5	Loop	for (variables[2]=0;variables[2]<variables[9];variables[2]++)
6	Add	variables[7] = variables[2] + variables[6];
7	Read	variables[5] = arrays[0][variables[2]];
8	Write	arrays[1][variables[7]] = variables[5];
9	Endloop	

Fragment	Success Rate
1	80%
1, 2	73%
1, 3	63%
1, 4	63%
3	67%

Table 22. Fragments assessed from program “Shift Right Lossy". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Literal	variables[5] = 1;
2	Subtract	variables[1] = variables[0] - variables[5];
3	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
4	Loop	for (variables[3]=0;variables[3]<variables[1];variables[3]++)
5	Add	variables[6] = variables[3] + variables[5];
6	Read	variables[4] = arrays[0][variables[3]];
7	Read	variables[7] = arrays[0][variables[6]];
8	Subtract	variables[8] = variables[4] - variables[7];
9	Condition	if (variables[8]>0)
10	Write	arrays[0][variables[6]] = variables[4];
11	Write	arrays[0][variables[3]] = variables[7];
12	Endloop	
13	Endloop	
14	Endloop	
15	Make Array	arrays[1] = new int[vars[0]]
16	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
17	Read	variables[5] = arrays[0][variables[2]];
18	Write	arrays[1][variables[2]] = variables[5];
19	Endloop	

Fragment	Success Rate
1	0%
1, 2	0%
1, 3	0%
1, 8	0%
1, 15	0%
1, 16	0%
3	0%
3, 8	0%
3, 15	0%
3, 16	0%
3, 17	0%
4	0%
4, 6	0%
8	0%
8, 9	0%
8, 15	0%
8, 16	0%
15	0%
15, 16	0%
16	0%

Table 23. Fragments assessed from program “Sort". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Make 2D Array	new 2DArray(size=variables[0]);
2	Literal	variables[6] = 2;
3	Divide	variables[4] = variables[0] / variables[6];
4	Loop	for (variables[2]=0;variables[2]<variables[4];variables[2]++)
5	Loop	for (variables[3]=0;variables[3]<variables[4];variables[3]++)
6	Add	variables[7] = variables[2] + variables[3];
7	Write to 2D	array[variables[7][variables[3]]=1;
8	Endloop	
9	Endloop	

Fragment	Success Rate
2	23%
2, 3	30%

Table 24. Fragments assessed from program “Mirrored Parallelogram". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Make 2D Array	new 2DArray(size=variables[0]);
2	Literal	variables[6] = 2;
3	Divide	variables[4] = variables[0] / variables[6];
4	Loop	for (variables[2]=0;variables[2]<variables[4];variables[2]++)
5	Add	variables[5] = variables[2] + variables[4];
6	Write to 2D	array[variables[5][variables[10]]=1;
7	Write to 2D	array[variables[2][variables[4]]=1;
8	Subtract	variables[6] = variables[4] - variables[2];
9	Write to 2D	array[variables[2][variables[6]]=1;
10	Write to 2D	array[variables[5][variables[6]]=1;
11	Endloop	
12	Literal	variables[8] = 1;
13	Subtract	variables[7] = variables[0] - variables[8];
14	Write to 2D	array[variables[7][variables[10]]=1;

Fragment	Success Rate
2	13%
2, 3	60%
2, 12	10%
12	13%
12, 13	40%

Table 25. Fragments assessed from program “Mirrored Hollow Parallelogram". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Make 2D Array	new 2DArray(size=variables[0]);
2	Literal	variables[1] = 1;
3	Subtract	variables[4] = variables[0] - variables[1];
4	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
5	Write to 2D	array[variables[2][variables[4]]=1;
6	Write to 2D	array[variables[5][variables[2]]=1;
7	Write to 2D	array[variables[2][variables[2]]=1;
8	Endloop	

Fragment	Success Rate
2	80%
2, 3	90%
2, 4	80%
4	90%
4, 6	63%
4, 7	87%

Table 26. Fragments assessed from program “Hollow Right Triangle". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Make 2D Array	new 2DArray(size=variables[0]);
2	Literal	variables[3] = 1;
3	Subtract	variables[4] = variables[0] - variables[3];
4	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
5	Write to 2D	array[variables[2][variables[4]]=1;
6	Write to 2D	array[variables[4][variables[2]]=1;
7	Subtract	variables[5] = variables[0] - variables[2];
8	Subtract	variables[5] = variables[5] - variables[3];
9	Write to 2D	array[variables[2][variables[5]]=1;
10	Endloop	

Fragment	Success Rate
2	67%
2, 3	93%
2, 4	80%
4	67%
4, 7	80%

Table 27. Fragments assessed from program “Hollow Mirrored Right Triangle". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Make 2D Array	new 2DArray(size=variables[0]);
2	Literal	variables[4] = 2;
3	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
4	Multiply	variables[6] = variables[2] * variables[4];
5	Subtract	variables[5] = variables[0] - variables[6];
6	Loop	for (variables[3]=0;variables[3]<variables[5];variables[3]++)
7	Add	variables[7] = variables[3] + variables[2];
8	Write to 2D	array[variables[7][variables[2]]=1;
9	Endloop	
10	Endloop	

Fragment	Success Rate
2	23%
2, 3	20%
3	20%

Table 28. Fragments assessed from program “Inverted Isoceles Triangle". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Make 2D Array	new 2DArray(size=variables[0]);
2	Literal	variables[7] = 1;
3	Literal	variables[4] = 2;
4	Divide	variables[5] = variables[0] / variables[4];
5	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
6	Loop	for (variables[3]=0;variables[3]<variables[5];variables[3]++)
7	Subtract	variables[8] = variables[0] - variables[5];
8	Divide	variables[8] = variables[8] / variables[4];
9	Subtract	variables[8] = variables[8] - variables[3];
10	Add	variables[9] = variables[8] + variables[7];
11	Condition	if (variables[9]>0)
12	Subtract	variables[9] = variables[2] - variables[8];
13	Condition	if (variables[9]>0)
14	Subtract	variables[9] = variables[0] - variables[8];
15	Subtract	variables[9] = variables[9] - variables[2];
16	Condition	if (variables[9]>0)
17	Write to 2D	array[variables[2][variables[3]]=1;
18	Endloop	
19	Endloop	
20	Endloop	
21	Endloop	
22	Endloop	

Fragment	Success Rate
2	10%
2, 3	10%
2, 5	10%
3	3%
3, 4	0%
3, 5	10%
5	3%

Table 29. Fragments assessed from program “Trapezoid". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)