Demystifying Genetic Programming based Synthesis

MEGAN CHU, WEI-CHENG HUANG, and MAYANK SHARAN
University of California, San Diego, USA

Program synthesis for well-specified programs, especially programming by example (PBE), is a well studied area with widely accepted solutions that employ constraint-based or enumerative search. Stochastic search methods like genetic programming (GP) have had success in the related domain of program repair. However, there have been surprisingly limited attempts at applying genetic programming methods to program synthesis tasks or performing comparisons on standard benchmarks with existing state-of-the-art methods. In this work, we present our implementation of a genetic programming-based synthesizer (GPSolver) for SyGuS PBE tasks. We also present a comparison with EUSolver which shows that GPSolver can outperform it in the number of benchmarks solved with the correct set of hyperparameters.

1 INTRODUCTION

Program synthesis for well-specified synthesis problems is a well studied area. The specification for these problems is typically in the form of input output (IO) examples or first-order logic constraints. The structural constraints are usually specified using a DSL and the search methods can be broadly categorised into enumerative, constraint-based, representation-based and stochastic. Enumerative and constraint-based methods both take a brute force approach to searching for the solution. However, these approaches are still among the best due to numerous optimisations that have been developed over the years for enumeration and SAT/SMT solvers.

Stochastic search intends to do better than a brute force search of the space. The key idea is to navigate the solution space using an approximate notion of solution quality. This directional search can exponentially reduce the number of programs explored before reaching the correct solution. However, it also means that the structure and guarantees that are associated with brute force search approaches, such as finding the smallest solution, is not applicable in this setting.

Genetic Programming (GP) is a widely used method in the connected discipline of program repair. The principle of stochastically replacing components of a program with different components that belong to a relatively limited set fits well with the core ideas of GP. However, the technique has been rarely applied to the task of program synthesis. In works which propose a GP-based synthesis method, there is an absence of comparisons with state-of-the-art synthesis methods on standard benchmarks [4]. There is some mystery regarding the applicability of GP to program synthesis tasks and how it compares to established methods.

To identify the suitability of Genetic Programming in program synthesis, we implement our own solver with GP[1] and compare it with EUSolver [2] on the SyGUS [1] PBE String Track benchmarks. EUSolver is an enumerative solver that has won several SyGus competitions, and we would like to perform a comprehensive comparison between these two methods.

2 METHODS

2.1 GPSolver

The GPSolver is designed to solve PBE synthesis problems. It can process SyGuS specifications, parsing the grammar, variable names, and IO examples. An initial population of ground programs is randomly generated using the grammar. This is then used to select parents, perform mutations and crossovers to generate children, select the fittest programs as the next generation, and repeat until a solution is found. This flow is detailed in Figure 1. The components of the solver here are focused on solving PBE Strings benchmarks due to limited scope of the work, but can be easily extended to other PBE

* Equally contributed by all authors.

[1]Code is being maintained at: https://github.com/meganchu2/CSE291/

Authors' address: Megan Chu, mec043@ucsd.edu; Wei-Cheng Huang, weh006@ucsd.edu; Mayank Sharan, msharan@ucsd.edu, University of California, San Diego, USA.

Publication date: March 2022.

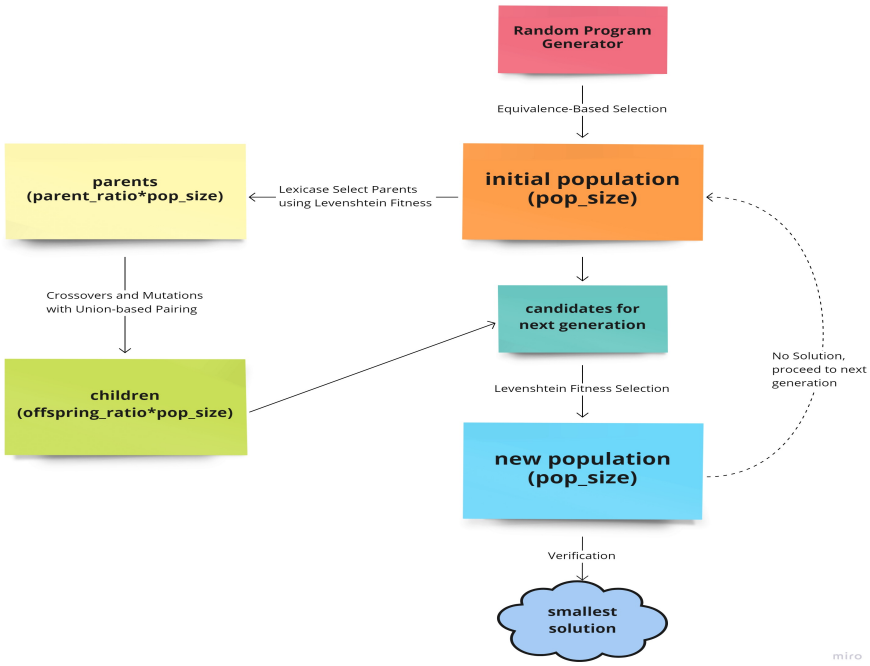

Fig. 1. The flow of GPSolver

task settings and even constraint-based specification settings by using a Counter-Example Guided Synthesis (CEGIS) [5] loop.

2.2 Equivalence-based Initial Population

Initial population for GP based methods is critical to ensure that the search spans a wide enough space and finds the solution quickly. Random generation can lead to numerous observationally equivalent programs in the initial population. This reduces the diversity within the population and reduces the likelihood of finding a solution. We maintain observational equivalence classes during initial population generation and select the smallest solution from each equivalence class. The generation continues till we have generated enough non-observationally equivalent programs to match the desired population size. The choice of the smallest program is due to the fact that subsequent generations are likely to have longer, hence the starting point should be the smallest program possible.

2.3 Crossovers and Mutations

Crossover and Mutation are two methods used to generate new programs from existing programs in the population. As illustrated in Figure 2, for both operations we process the programs as Abstract Syntax Trees (ASTs). Mutation requires a single input program, we select a node in this program at random, delete the subtree at that node and randomly generate a new subtree to replace it. The type of the node is maintained with this mutation, for example if the node deleted was a non terminal string type then the replacement will also be of the same type. Crossover requires 2 programs, say program A and program B. We select a node at random from program A and find a node of matching type from program B. If there are multiple matching type nodes in program B we pick one at random from them. The selected nodes from program A and program B are then exchanged to generate 2 new programs.

2.4 Control Functions

The different steps in GP are controlled by a few control functions. These are typically carefully designed heuristics to guide the search and are critical to the quality of the search. The GPSolver has 3 key control functions: Fitness, Parent Selection, and Parent Pairing.

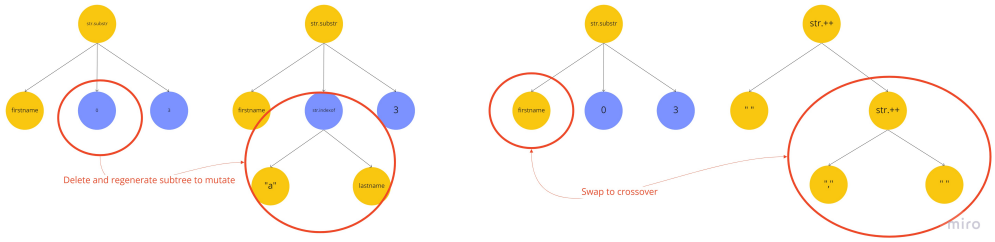

Fig. 2. Schematic of Crossovers and Mutations

Fitness. The fitness function is a mapping from a ground program to a score between 0 and 1 based on a given behavioral specification. This fitness score is supposed to indicate the closeness of the program to the correct solution with a score of 1 representing the correct solution. In the PBE setting this can be computed as a measure of closeness of the generated output to the correct output. We compute a score for each IO example and take the average to produce the fitness score of a program. In this work we considered the following fitness functions, where y' is the generated output, and y is the expected output:

- Binary: The score is 1 if the output is correct and 0 otherwise, $\mathbb{I}[y = y^{'}]$
- Match: The score is the ratio of the length of the longest common substring and length of correct output, $\frac{\max\{j-i|y'[i:j]=y[k:l]\}}{len(y)}$
- Jaccard: The score is computed as the jaccard similarity representing the generated output and correct output as character sets, $\frac{|(\text{set of chars in } y')\cap(\text{set of chars in } y)|}{|(\text{set of chars in } y')\cup(\text{set of chars in } y)|}$
- Levenshtein: The score is computed as 1 minus the ratio of the edit distance between the correct output and the generated output and the longer output length, $1 - \frac{Levenshtein(y,y^{'})}{\max(len(y),len(y^{'}))}$

Parent Selection. At each generation we need to select parents from the population for crossovers and mutations to create the next generation of programs. The parent selection can be done based on the fitness score but this does not work well because two parent programs with very high fitness scores might be solving a very similar set of examples and crossover will not yield better offsprings. To select better parents we employed the Lexicase [3] method that aims at selecting a diverse set of parents where each specializes on a different subset of the examples.

The method randomly selects a permutation of the examples and computes the fitness scores of all candidate parents on the first example in that permutation. All programs with a fitness score over a threshold are selected and then evaluated on the second example in the permutation. This process repeats till we are left with only one candidate program or all examples have been evaluated. If at the end there are multiple candidates we pick one candidate randomly. This process is repeated with different example permutations until we select the required number of parents.

Parent Pairing. Now that our parent selection ensures parents that specialize on solving different subsets of examples, we need to pair them well to ensure high-quality offsprings. Instead of randomly selecting two parents for breeding, we can select two parents that maximize the union of the examples solve. This ensures that offspring generated with crossovers have a good chance of solving more examples than their parents.

3 EVALUATION

3.1 Hyperparameter Selection

We selected each hyperparameter for the GPSolver by performing a search using a subset of 25 PBE Strings tasks from the SyGuS 2018 competition as the validation set. The hyperparameters are:

Table 1. Summarizing metrics comparing EUSolver and GPSolver on 109 SyGuS PBE Strings benchmarks. The results for GPSolver per benchmark are taken as the best across the 3 seeds. The solution time and size are averaged across the benchmarks that were solved.

Metric	EUSolver	GPSolver
Solved	59	73
Solution Time (in seconds)	131.05	299.53
Solution Size	6.95	12.36

Table 2. Correctness Evaluation across platforms

Solver	Seed	Solved Benchmarks		
		Linux	Windows	macOS
EUSolver	-	51	57	59
GPSolver	17	53	53	61
	42	57	58	57
	1729	50	52	59
	Combined	73	73	73

- Population Size: Too small a population size severely limits the number of programs and the diversity of the search. Large population size does not provide an added benefit to the search but significantly increases the run time. Balancing the 2 factors, we select a population size of 2000. It is important to note that the different population size for different problems helped, but it was dependent on the solution complexity rather than anything that was known before solving and hence could not be set dynamically.
- Number of Parents: Selecting too few parents did not give us enough diversity in the children, but selecting too many parents did not remove enough bad programs. We empirically determined 0.25 as the best parent count as a ratio of the total population size. For the population size of 2000, this means we select 500 parents.
- Number of Offspring: We wanted to generate enough good offspring to improve the quality of programs after each generation. However, generating too many children would replace almost every program from the previous generation. The optimal value for this was 1.5 times the population size which is 3000 offspring for the population size of 2000.
- Crossover and Mutation Probability: We experimented with values from 0 to 1 for each to see how that changed the performance. There was no significant difference in performance other than when the probability for one of the 2 was set to 0, which led to fewer benchmarks being solved. The selected values used for our experiments are 0.5 for both probabilities.

3.2 Comparison with EUSolver

We evaluated GPSolver against EUSolver [2] on 109 benchmarks from the SyGus [1] 2018 PBE Strings Track, and we measured correctness, solution size, and synthesis time as metrics. Both solvers were set to a timeout of 1800 seconds. The control functions used by the GPSolver are Levenshtein for fitness, lexicase select for parent selection, and union-based method for parent pairing. The GPSolver was run thrice with different seeds. The experiments were run on a mid-2015 8-core CPU Macbook Pro running macOS Monterey. The results for this evaluation are presented in Table 1. To evaluate platform independence we also reproduced this experimental setup on Linux and Windows platforms, the correctness results for which are summarized in Table 2.

3.3 Discussion

Table 2 shows that in any individual run, EUSolver and GPSolver correctly solve similar number of benchmarks. However, the stochastic nature of the GPSolver ensures that the unique number of benchmarks solved by at least 1 run is 73, much higher than the 59 solved by EUsolver. It is interesting to note that this number comes out to be 73 across all platforms we test on, however even those 73 are not the same set of benchmarks. This is due to random seeds in Python being evaluated differently on different platforms and also shows that there is potential for further increase in number of benchmarks solved by adding runs for additional seeds.

EUSolver is guaranteed by the nature of its search to generate the smallest solution, while there are no such guarantees associated with the GPSolver. Table 1 shows the result that the average solution size for the GPSolver is roughly double that of the EUSolver.

We also noticed that in problems with simple solutions the EUSolver is much faster in arriving at a solution as it does a search by size. The GPSolver on the other hand due to the stochasticity can take some time to generate the correct solution and is comparably much slower on these benchmarks. The situation reverses though in cases with complex solutions where GPSolver is significantly faster compared to EUSolver or solves benchmarks that EUSolver times out on as it has no obligation to follow an order.

Another interesting component is that most solutions to the GPSolver are found in the first few generations which is desirable as this leads to smaller solutions and faster solve times (Figure 5 in appendix). This also indicates that the quality of the initial population is good and leads to quick solution searches.

4 EXAMPLES

In this section we present the solutions to a few benchmarks that GPSolver can solve but EUSolver cannot.

4.1 dr-name

In this benchmark the program is supposed to output the first name prefixed by "Dr.", given the full name. For example if the input is "Mariya Sergienko", the solution should output "Dr. Mariya". GPSolver solves it in 442 seconds, and produced the following program:

```
str.++(
    str.++("Dr.", str.substr(" ", 0, 1)),
    str.substr(name, 0, str.indexof(name, " ", 0)))
)
```

4.2 lastname-long-repeat

In this benchmark the program is supposed to extract the last name given the full name. For example, with "Jan Kotas" as input, it should output "Kotas". GPSolver solves it in 860 seconds, and produced the following program:

```
str.replace(
    name,
    str.substr(
        str.replace(" ", " ", name),
        -(1, 1),
        str.indexof(str.++(" ", name), " ", 1)
    ),
    str.at(int.to.str(0), -(0, 1))
)
```

4.3 name-combine-4

In this benchmark the program is supposed to output the last name, followed by the abbreviated first name, given the first name and the last name. For example, if the input is "Launa" and "Withers", the output should be "Withers, L.". GPSolver solves it in 511 seconds, and produced the following program:

```
str.++(
    str.++(str.++(lastname, ","), " "),
    str.++(str.at(firstname, 0), str.replace(".", firstname, " "))
)
```

4.4 phone-7-long

In this benchmark the program is supposed to extract the middle three numbers from a nine-digit phone number, excluding the country code. For example, with "+45 095-746-635" as input, it should output "746". GPSolver solves it in 299 seconds, and produced the following program:

```
str.substr(
    name,
    +(str.indexof(name, "-", 4), str.len(".")),
    3
)
```

5 CONCLUSION

Through the experiments we show that the performance of GPSolver is sensible to (1) the quality of the initial population, and (2) a good set of control functions and hyperparameters.

The GP technique differs from enumerative solvers in that it can potentially find complex solutions more quickly. Although this randomness makes it hard to provide guarantees about a solution if it is found, it does present opportunities for running many instances of GPSolver in parallel to maximize the chances of finding a solution.

There are rooms for future improvements for the GPSolver, for instance, as our experiments are limited to the PBE Strings track, we can also explore appropriate heuristics and hyperparameters for other types of tasks. Another direction would be to seek guidance from neural networks in estimating fitness scores, breeding, or mutation.

REFERENCES

[1] Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. 2019. SyGuS-Comp 2018: Results and Analysis. arXiv:1904.07146 [cs.PL]

[2] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

[3] Thomas Helmuth, Lee Spector, and James Matheson. 2015. Solving Uncompromising Problems with Lexicase Selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (Oct. 2015), 630–643. https://doi.org/doi:10.1109/TEVC.2014.2362729

[4] Edward Pantridge, Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2017. On the difficulty of benchmarking inductive program synthesis methods. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 1589–1596.

[5] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5 (Oct. 2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

A FIGURES ON EXPERIMENTAL RESULTS

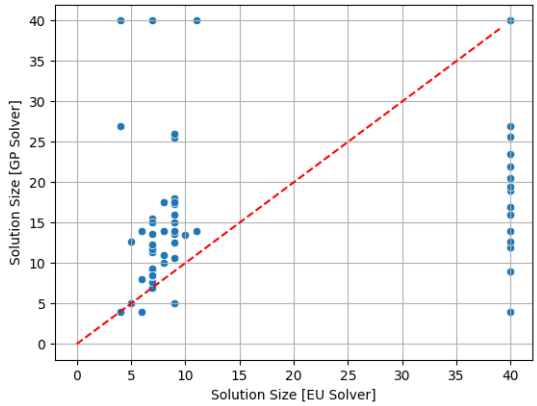

Fig. 3. Solution Size of EUSolver vs. Solution Size of GPSolver. For benchmarks that timed out for a solver, we set the solution size to 40. Note that the solution size of GPSolver is the average value from our three runs.

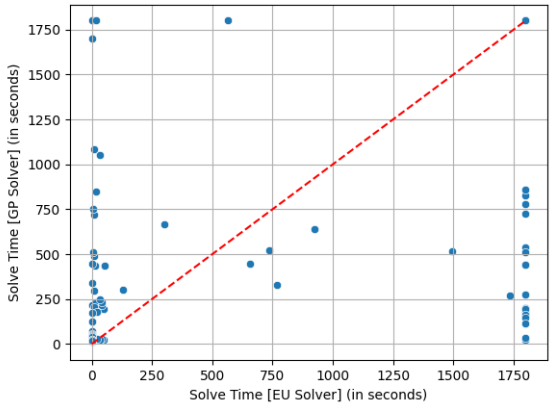

Fig. 4. Solve Time of EUSolver vs. Solve Time of GPSolver. For benchmarks that timed out for a solver, we set the solve time to 1800 seconds. Note that the solve time of GPSolver is the minimum value from our three runs. We use minimum instead of average to avoid skewing the results when one or more runs of GPSolver timed out on the benchmark.

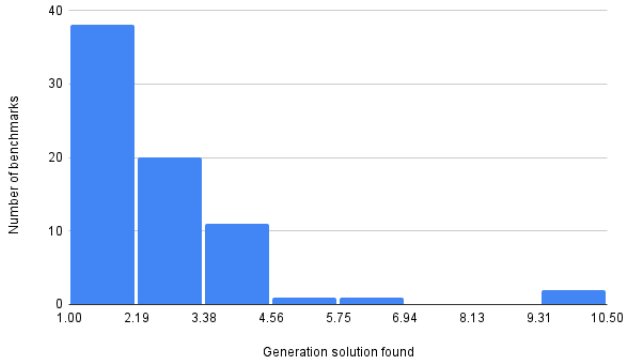

Fig. 5. Histogram counting the number of benchmarks which found a solution during a given generation. The generation for a benchmark is averaged across our 3 runs. Most solutions are found in the first few generations.