Parameter Tuning in Scratch Program Repair. Can parameters affect the quality of the solution?

Mariia Koroleva

[a]*University of Passau, Innstraße 41, Passau, 94032, Germany*

Abstract

Teachers often face a lack of support when teaching programming to their students. Automatically fixing students' programs can provide important help, especially in visual programming languages like Scratch. The Whisker tool uses a genetic algorithm to fix programs, similar to GenProg but adapted for Scratch. However, this process can be time-consuming and requires parameter customization. The objectives of the study are to compare different sets of parameters and their impact on the solution quality and repair time of the program. Two algorithms will be used for parameter selection - Random Search Algorithm and Differential Evolution Algorithm. As evaluation criteria we will use fitness value, number of iterations, and execution time.

Keywords: Scratch Programming Language, WHISKER, Parameter tuning, Genetic Programming, Random Search, Differential evolution, Automated Program Repair

1. Introduction

Learning to program can be challenging for both students and teachers. Students often face numerous questions that they may struggle to formulate correctly. Simultaneously, teachers may find it difficult to provide quality assistance to each student due to time constraints. Automatic program fixing and generating hints for students can alleviate these challenges. However, fixing a program in the visual programming language Scratch is not a trivial task. It requires the use of genetic algorithms to find suitable solutions.

The Whisker tool employs an algorithm similar to the GenProg [3] genetic algorithm, but it is adapted to the specific language and task [7]. Finding the correct solution can be time-consuming, necessitating parameter tuning to reduce the time required to discover a program that passes all test cases. This work focuses on investigating different sets of parameters for a genetic algorithm and comparing algorithms for finding these parameters. The script code and a description of the installation process can be found in the following GitLab repositories [1] [2].

We plan to compare different sets of parameters for the GenProg algorithm to identify patterns and how they may affect solution quality, number of iterations, and repair execution time. We will use two algorithms to generate parameters: **Random Search Algorithm** (RS) and **Differential Evolution Algorithm** (DE).

There are two types of parameter sets planned to be used:

1. Integer. Population size and elitism size.
2. Decimal. Mutation and crossover rates.

- **RQ1**: Does population size and elitism affect solution quality, execution time, number of iterations? Are there any other patterns?
- **RQ2**: Does mutation and crossover rate affect solution quality, run time, number of iterations? Are there any other patterns?
- **RQ3**: Are there differences in the performance of the two algorithms (RS and DE) in the context of parameter generation?

2. Background

2.1. Structure of Scratch program

The Scratch programming language is designed specifically for students who have had no previous programming experience. It has a colorful user friendly interface and consists of simple commands that make it easier and faster to understand basic programming principles. Scratch consists of many different blocks that vary in shape and purpose. Thanks to color coding, students can get visual cues to more quickly recognize the function of each block (Figure 1) [4].

These block types providing an intuitive and visually structured approach to help users, especially beginners, grasp fundamental programming concepts.

2.2. Types of bugs in Scratch. Why should we fix them?

While Scratch's block-based nature prevents syntactical errors, it does not prevent problematic or incorrect code. The concept of "code smells" is introduced, denoting code idioms that increase the likelihood of bugs.

[1] `https://gitlab.infosun.fim.uni-passau.de/koroleva/parameter-tuning-genprog`

[2] `https://gitlab.infosun.fim.uni-passau.de/se2/students/23ws-seminar-parameter-tuning/koroleva`

(a) Block Shapes

HAT	STACK	BOOLEAN	REPORTER	C BLOCK	CAP
Hat blocks start every script in Scratch. They are always at the top - like a hat!	Stack blocks perform the main commands.	Boolean blocks are the conditions. They are either true or false.	Reporter blocks are the values. They hold numbers.	These blocks look like the letter 'c.' They can loop blocks inside of them.	Cap blocks end a script.

(b) Block Categories

Category	Notes	Category	Notes
Motion	Movements of sprites like angles and position	Sensing	Sprites can interact with the surroundings
Looks	Controls the visuals of the sprite	Operators	Mathematical operators, comparisons
Sound	Plays audio files and effects	Variables	Variable and List usage and assignment
Events	Event handlers	My Blocks	Allows defining functions which have no return value
Control	Conditionals and loops etc.	Extensions	Add extra blocks and features to Scratch

Figure 1: An overview of Scratch blocks, covering both Block Shapes and Block Categories

Figure 2: An example of the Forever Inside Loop bug pattern, as evidenced in a publicly shared project: The outer forever-loop blocks are confined to a single execution cycle, as the program fails to exit the inner forever-loop, hindering the continuous execution of subsequent statements. (The picture was taken from the paper [7])

[1] categorizes bug patterns into three types: syntax errors, general bugs, and Scratch-specific bugs. Examples include ambiguous custom block signatures, endless recursion, and missing backdrop switches. One of them you can see in the Figure 2.

Resolving bugs not only improves the overall reliability of Scratch programs but also enhances the learning experience for beginners. It promotes a smoother understanding of programming concepts and prevents potential frustration caused by unexpected issues. Moreover, addressing bugs in Scratch contributes to the creation of more polished and effective projects, fostering a positive and productive coding environment for users.

2.3. Repair Algorithm Overview

The repair algorithm outlined in Algorithm 1 is inspired by GenProg but adapted specifically for Scratch programming [7].

According to the article [7], there is a corrected subject and a set of tests that evaluates the expected behavior of the program. To preserve the genetic material of the original program, the method starts by creating an **initial population** of program variations (**chromosomes**) by mutation. The results of test performance are then used to evaluate the suitability of the test. The algorithm creates a new population for the next generation by selecting parents, performing **crossover and mutation**, and evaluating fitness. The most successful genetic material is retained. Until the fittest program is obtained, the process is repeated.

The fitness function considers program repair as a problem of maximizing the number of passed tests in a predefined test set. The method distinguishes between execution states (pass/fail) and computes the proportion of successful test passes to the total number of tests.

The "closeness" of failed assertions is measured using assertion distance metrics, which direct the search for program variations that fix particular assertion failures. The calculation rules take into account a variety of assertion kinds, such as comparisons of strings and numbers, objects that resemble arrays, and Boolean values. The assertion distance adds to the algorithm's overall success by offering useful data for efficient repair techniques.

The fitness evaluation entails executing the software in a browser window without a graphical user interface, utilizing the Whisker testing framework.

Data: Program p to repair
Data: Fix source F, defaults to $F = \{p\}$ if not explicitly specified
Data: Test suite T
Result: Repaired program p'
$P \leftarrow$ generate_initial_population(p);
evaluate_fitness$_T(P)$;
$E \leftarrow$ Set of fittest $p \in P$;
while *stopping condition not reached* **do**
  $P \leftarrow$ filter_viable(P);
  $P \leftarrow$ select_parents(P);
  $O \leftarrow$ cross_over(P);
  $O \leftarrow$ mutate$_F(O)$;
  evaluate_fitness$_T(O)$;
  $P \leftarrow O \cup E$;
  $E \leftarrow$ Set of fittest $p \in P$;
end
return $p' \in E$;

Algorithm 1: Repair algorithm for Scratch

The repair algorithm outlined in Algorithm 1 is designed for solving issues in Scratch programs. Given an input program p, a fix source F (defaulting to $F = \{p\}$ if unspecified), and a test suite T, the algorithm initializes a population P with variants of the original program. It then evaluates the fitness of each program in P against the test suite T. The algorithm performs crossover and mutation operations, filters viable programs, chooses parents for reproduction, and assesses the fitness of the offspring in the recurring main loop. The process keeps on until a stopping condition is met, thereby keeping the programs that are most likely to fit in the population. The end result is a fixed version of p' that performs better on the provided testing set. The approach applies genetic programming principles to repeatedly generate and improve program variations, aiming for increased accuracy and speed.

3. Experimental set-up

3.0.1. Parameter Tuning. Which parameters should be chosen?

The performance of a Genetic Algorithm (GA) is highly impacted by the selection of parameters like mutation rate,

crossover rate, population size, elitism size, etc. Identifying ideal parameter settings beforehand is known as the parameter tuning challenge. Factors such as parallel programming, genetic coding, goal selection, and termination conditions are design characteristics that might impact GA performance [6].

First, let's provide some brief information about the parameters we have to tune.

Population Size and Elitism Size. These parameters determine the number of candidates in each generation [2]. A bigger number of individuals demands more computer resources for searching the solution space. Finding the right balance between exploration and exploitation is crucial. If programs are complicated and demonstrate varied challenges, a bigger population could be useful. Conversely, for relatively basic difficulties, a smaller population may sufficient.

Constraints. The following constraints were stated to generate population size and elitism:

1. Population size - [12, 48]. It must be divisible by 2, as it is stated in the code of the Wisker project, otherwise the repair process of the program will not start in principle. Hence: the available values for generation are: 12, 16, 20, ... , 48.
2. Elitism size - [2, 8]. Since the size of elitism is only a small part of the population - the best of the best, by definition it must be smaller than the size of the population. At the same time, it must necessarily be divisible by 2. Hence: available values for generation: 2, 4, 6, 8.

Crossover and Mutation Rates. Whereas the mutation rate establishes the likelihood of applying mutation to a single person, the crossover rate establishes the likelihood of applying crossover to two parent individuals. The issue being addressed determines the values of these rates.

A crucial part of setting the parameters of the genetic algorithm is adjusting the crossover rate, which affects how well the algorithm can explore the solution space and produce a varied range of high-quality offspring [2].

In genetic algorithms for repairing Scratch programs, the mutation rate controls the frequency of random changes introduced to the population. A low mutation rate may lead to premature convergence and being trapped in local optima. Increasing the mutation rate improves diversity and helping the algorithm avoid local optima. Determining the optimal mutation rate typically requires empirical testing with multiple runs of the algorithm, observing its impact on convergence, diversity, and overall performance [7].

Constraints. Since crossover and mutation rates are probabilistic values, the generated values are limited from 0 to 1.

3.1. Random Search Algorithm

The advantages of this algorithm are its simplicity, stability and intuitiveness. The disadvantages are the low speed of convergence, as well as the uncertainty in choosing the stopping condition. Another disadvantage is that the random search algorithm needs to be customized separately for each task [10].

In our case, this algorithm was chosen because of the simplicity of its implementation, and also because in one of the articles it showed proven effectiveness in solving problems related to tuning parameters.

```
Data: Initialize parameters
Result: Termination
while stopping criterion not met do
    if timeout reached or desired fitness reached then
        Terminate the algorithm;
    else
        Choose mode (RS or DE);
        if mode = RS then
            Execute RS;
            Optimization Function (RS);
        else
            Execute DE;
            Optimization Function (DE);
        end
    end
end
Write all results to a file;
```

Algorithm 2: Main Algorithm with Results Recording

```
Data: Fitness function f : ℝ^n → ℝ, Initial position x ∈ ℝ^n
Result: Optimal position
Function Initialize(x)
    x ← Random position in the search-space;
end
while Termination criterion not met do
    y ← SampleNewPosition(x);
    if f(y) < f(x) then
        MoveToNewPosition(x, y);
    end
end
```

Algorithm 3: Random Search Algorithm

3.2. Differential Evolution Algorithm

Differential evolution (DE) is a direct-search optimization method. According to the paper [8], the algorithm includes include includes mutation, crossover and selection. Mutation generates a mutant vector by combining two population vectors, and crossover introduces diversity into the perturbed parameter vectors. A candidate is created by mixing the mutant vector with the target vector. If the trial vector has a lower value of the cost function (assuming it is a minimization problem) than the target vector, it replaces the target vector in the next generation.

This algorithm is better suited for decimal numbers, but can be optimized by rounding to an integer or one of the acceptable values [5], which is what was done in this work.

The differential evolution algorithm is already implemented in the python library **SciPy**[3]. It is already optimized enough and supports parallel execution, so it makes sense to use it. According to the paper [9], the scipy.optimize.differential_evolution [4].

[3] https://docs.scipy.org/doc/scipy/

[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html

Data: Fitness function f, Population size NP, Crossover probability CR, Differential weight F, Maximum iterations, Termination criterion
Result: Best-found candidate solution **m**
Initialization: Randomly initialize the population $\{\mathbf{x}_1, \ldots, \mathbf{x}_{NP}\}$;
while *Termination criterion not met* **do**
  for *each agent* $\mathbf{x}_i$ *in the population* **do**
    Randomly select three distinct agents **a**, **b**, and **c**;
    Randomly select an index $R \in \{1, \ldots, n\}$ where n is the dimensionality;
    for *each dimension* $i \in \{1, \ldots, n\}$ **do**
      Generate a uniform random number $r \sim U(0, 1)$;
      if $r < CR$ *or* $i = R$ **then**
        Compute trial solution component: $y_i = a_i + F \times (b_i - c_i)$;
      end
      else
        Maintain current solution component: $y_i = x_i$;
      end
    end
    if $f(\mathbf{y}) \leq f(\mathbf{x}_i)$ **then**
      Replace $\mathbf{x}_i$ with **y** in the population;
    end
  end
  Identify the agent with the best fitness in the population;
end
return *Best-found candidate solution* **m**;

Algorithm 4: Differential Evolution (DE)

function is a stochastic global optimizer that works by evolving a population of candidate solutions. In each iteration, trial candidates are generated by combination of candidates from the existing population. If the trial candidates represent an improvement, then the population is updated.

4. Evaluation. Comparison of different sets of parameters and two algorithms that generate them

Repair subjects are divided into two main types: SIMPLE and COMPLEX. It may turn out that the appropriate parameters for different problems are fundamentally different.

Fitness value in the context of renovation Scratch programs is based on the ratio of passed tests to the total number of them, as well as assertion distance which estimates “how close” control flow comes to taking the opposite branch, and is used to guide the search towards yet unexplored parts of the program.

Repair time is the time it takes Whisker to find a solution - that is, a repaired program.

Number of iterations. How many steps it took to fix the program.

4.1. Program repair process

Scratch program repair will be performed in so-called ”headless” mode to reduce execution time for complex programs. In this mode, browser windows do not open, however, the necessary logs will still be output to the console to evaluate performance. Moreover, since we will be conducting experiments on clusters where no GUI is available, there is simply no alternative to the ”headless” mode.

We will compare three types of programs that differ in level of complexity: from the most primitive example consisting of several blocks to a complicated program with nested loops and complex condition blocks.

5. Observations

Despite high expectations, the impact of the parameters on solution quality and program repair time was negligible. However, some interesting observations can still be highlighted.

It should also be noted that the maximum fitness value varies with each program, since it directly depends on the number of tests.

Reconsidering Population Size: Impact on Solution Quality Across Program Complexity. If the size of elitism is large and the population size is small, the solution is often of insufficient quality. This was observed in all three program repairs. Tables 1, 2 and 3 show that even for simple and medium complexity programs it is necessary to have a sufficient population size to obtain a repaired program of high quality.

Population Size	Elitism Size	Fitness
12	8	9
32	8	11
44	8	11

Table 1: Comparative analysis of parameter sets for the repair of the simple Max.sb3 program, under conditions of equal elitism size across diverse population sizes.

Population Size	Elitism Size	Fitness
16	6	3.3
32	6	4
36	6	4

Table 2: Comparative analysis of parameter sets for the repair of the medium difficulty program ForeverInsideLoop.sb3 program, under conditions of equal elitism size across diverse population sizes.

Population Size	Elitism Size	Fitness
16	2	13
40	2	17
44	2	17

Table 3: Comparative analysis of parameter sets for the repair of the hard difficulty program BoatRace.sb3 program, under conditions of equal elitism size across diverse population sizes.

Balancing Mutation Rates: Impact on Program Repair Time and Solution Quality. Often, too high a mutation level can lead to a longer program repair time, which can be seen in Figures 3 and 4 . However, this has almost no effect on the quality of the solution. The reason for this lies in the tendency of redundant mutations to introduce more changes to the program, but most of them may not contribute significantly to the improvement of the final solution.

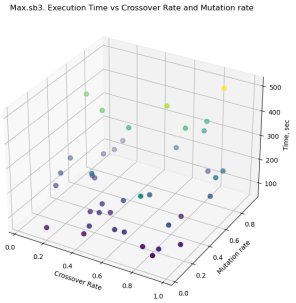

Figure 3: Easy difficulty Scratch program - Max.sb3. The Impact of Mutation Rate on Program Repair Time.

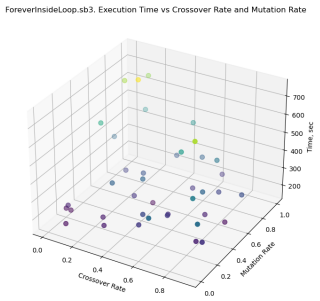

Figure 4: Medium difficulty Scratch program - ForeverLoop.sb3. The Impact of Mutation Rate on Program Repair Time.

Repetitive Patterns in Random Search: Impact on Efficiency and Solutions. The random search algorithm is not immune to repeated generation of identical sets of parameters. The use of random values in a random search algorithm is a key part of its concept. Differential Evolution may use random values in the process of creating new individuals, but this process is controlled by a more complex mechanism that may not result in values being repeated as often as in random search.

Exploring Elitism Size: Minor Impact on Iteration Numbers in Program Repair. Figure 5 shows that when the elitism

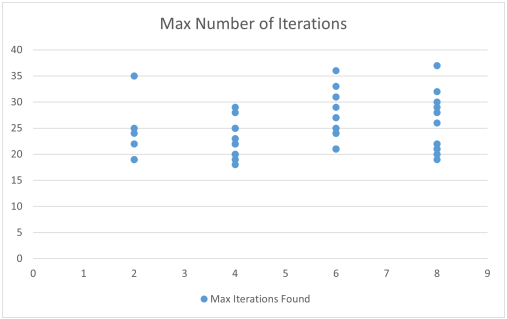

Figure 5: Impact of Elitism Size on Iteration Numbers

size increases, the number of iterations that it took to repair the program increases slightly.

Mutation Rate Impact: Excessive Mutations Extend Iteration Requirements. If the mutation is too high, namely around 0.7, more iterations are required to reach the desired solution quality, as confirmed by the three-meter dot plots 6 and 7. This observation correlates with the one mentioned earlier, when we said that a higher mutation rate increases the execution time of program repair. Since more iterations require more execution time, our suspicion is confirmed once again.

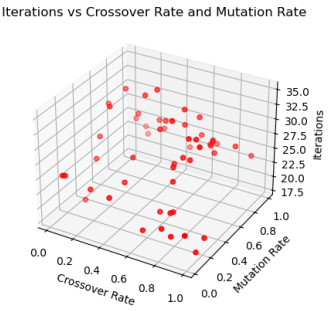

Figure 6: The effect of mutation rate on the number of iterations. 3d dot plot.

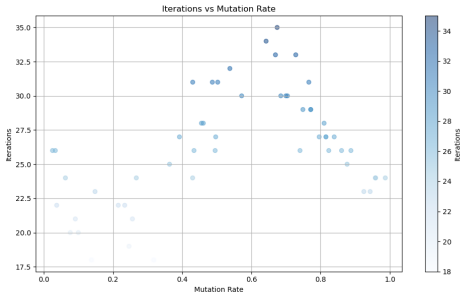

Figure 7: The effect of mutation rate on the number of iterations. 2d dot plot.

In this case, the highest number of iterations is achieved when the crossover rate is too low and the mutation rate is too high. An example can be seen in Table 4.

Crossover Rate	Mutation Rate	Number of Iterations
0,395645162	0,674255117	35
0,325787637	0,642341689	34
0,126797434	0,728027364	33

Table 4: An example of the effect of mutation rate and crossover rate on the number of iterations, taken from the output of experiments with the program BoatRace.sb3

Population Size: A Key Driver for Early Solution Discovery in Program Repair.

There is a correlation between the population size and the maxIter - solutionFoundIter difference. The larger this difference is, the more times the final solution has been passed from generation to generation, and the earlier it was found in the process of program repair. In Figure 8 we can see that if the population size is small, such as 16 or 20, the solution was found quite late, and if the population size is large, such as 32, the final version of the repaired program was found early.

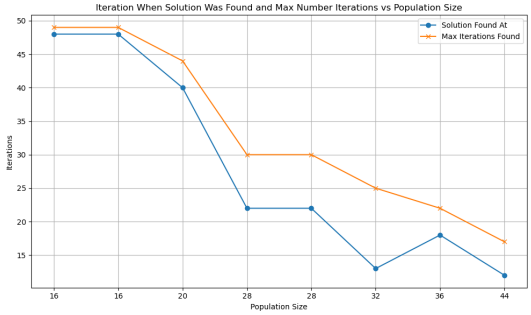

Figure 8: **Correlation between population size and number of iterations**

Negligible Impact of Mutation and Crossover Rates on Iteration Maximum and Final Solution Lifetime. The mutation and crossover rates are found to have no significant effect on both the maximum number of iterations and the lifetime of the final solution. As can be seen in Figures 9 and 10, all values are approximately in the same range.

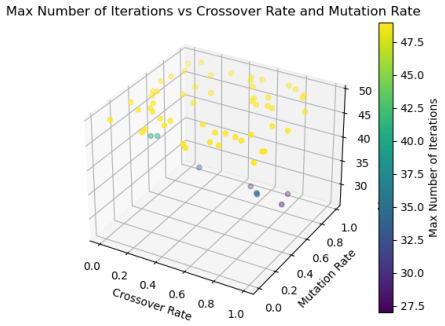

Figure 9: High complexity program Spaceship.sb3. Effect of crossover and mutation rate on the maximum number of iterations.

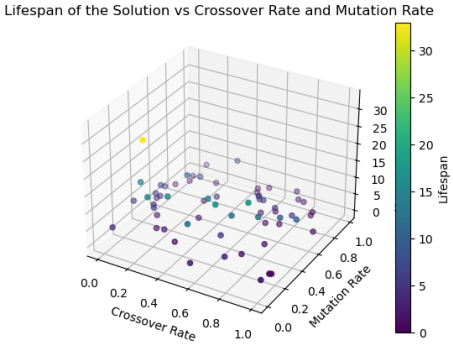

Figure 10: High complexity program Spaceship.sb3. Effect of crossover and mutation rate on the lifespan of the solution.

6. Conclusion

Based on the observations above, we can conclude that although there is no ideal set of parameters that universally optimizes program repair, we can identify several valuable and interesting points that can help in parameter tuning:

1. Optimal population size. This is a critical parameter that has implications for even the simplest repair subjects. Ensuring adequate population size is critical to achieving high quality solutions.
2. Balanced mutation rate. It is important to achieve a balance in the mutation rate to avoid unnecessary iterations and increased repair time.
3. Balanced crossover rate. A low crossover rate combined with a high mutation rate can potentially increase repair time and the number of iterations.

By carefully considering and adjusting these parameters based on the specific characteristics of the repair task, practitioners can streamline the repair process, leading to improved solution quality and reduced repair time.

References

[1] Christoph Frädrich, Florian Obermüller, Nina Körber, Ute Heuer, and Gordon Fraser. Common bugs in scratch programs. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, page 89–95, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368742. doi: 10.1145/3341525.3387389. URL `https://doi.org/10.1145/3341525.3387389`.

[2] Ahmad Hassanat, Khalid Almohammadi, Esra'a Alkafaween, Eman Abunawas, Awni Hammouri, and V. B. Surya Prasath. Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach. *Information*, 10(12):390, December 2019. ISSN 2078-2489. doi: 10.3390/info10120390. URL `http://dx.doi.org/10.3390/info10120390`.

[3] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[4] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Trans. Comput. Educ.*, 10(4), nov 2010. doi: 10.1145/1868358.1868363. URL `https://doi.org/10.1145/1868358.1868363`.

[5] David G Mayer, BP Kinghorn, and Ainsley A Archer. Differential evolution–an easy and efficient evolutionary algorithm for model optimisation. *Agricultural Systems*, 83(3):315–328, 2005.

[6] Mohsen Mosayebi and Manbir Sodhi. Tuning genetic algorithm parameters using design of experiments. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, GECCO '20, page 1937–1944, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371278. doi: 10.1145/3377929.3398136. URL `https://doi.org/10.1145/3377929.3398136`.

[7] Sebastian Schweikl and Gordon Fraser. Automated repair of block-based learners' programs (work in progress). In *Proceedings of the Conference*, page 26, New York, NY, USA, 2018. ACM. doi: XXXXXXX.XXXXXXX.

[8] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11:341–359, 1997.

[9] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[10] Zelda B Zabinsky et al. Random search algorithms. *Department of Industrial and Systems Engineering, University of Washington, USA*, 2009.