# Automatically Finding Patches Using Genetic Programming

Name Lastname

## Problem

The authors propose an automated approach for finding and repairing bugs in legacy systems based on genetic programming (GP).

## Main Idea

Genetic programming is a methodology inspired by biological evolution. By using computational analogs to biological crossover and mutation new versions of a program are generated automatically. This population of new programs is then evaluated by an user defined fittness function to only select the programs that show an improved behavior as compared to the original program. In this case the desired behavior is to retain all original functionality and additionally fixing bugs found in the program code.

## Contribution/Implementation details

Their approach works on legacy software that is written in C. Besides the program to be analyzed it requires a set of successful test cases that encode the required behavior and a failing test case that discovers a defect. Then, genetic programming is used to create new variants of the original program that both passes the successful and the failing test cases. Since GP operates on a potentially infinite-sized search space the authors had to impose some restrictions in order to make it a feasible method for fixing errors in programs. Changes in the program can only be based on statements found in other parts of the program. Furthermore, changes are only be made in areas which are in the execution path that produced the error. In order to find new variants of the program they use crossover and mutation operations. Starting with some initial population of variants, crossover generates new versions by combining the rst part of some variant with the second part of an other variant. For each variant in the resulting population the mutation operation is applied. The mutation operation either deletes some statement, inserts some statement or swaps two statements. They define a statement to be either a simple instruction like an assignment or method invocation, an return instruction, an if-conditional, or a loop. This approach is different to the traditional genetic programming in which only one operation is applied to generate new variants for a generation. After a new population of variants is created, only the best ones are selected for the next generation. The selection is based on a custom user-defined fitness function which evaluates the effectiveness of the program variants. Their fitness function computes a weighted sum of all test cases passed by a variant. Variants that do not compile or pass the positive test cases are directly discarded. The remaining variants form the initial population for the next generation. The algorithm is repeated until some threshold is exceeded or some variant is found that passes all test cases. If a successful variant is found it is compared to the original program to create a minimal patch that can be used to fix the defect.

## Evaluation

They evaluated their genetic algorithm on ten open-source software projects that contained a total of 63,000 LOC. They fixed one error in each project by using an average of five positive test cases for each project either created manually or automatically. They were able to fix all errors with an average processing time of 184 seconds and 37 fitness evaluations per successful repair.

## Questions / Suggestions

- They only use statements for mutation which are used somewhere in the program. Unfortunately the evaluation lacks information how this affects the effectiveness of the approach, for example if the programs are rather small or there are no statements to fix certain kinds of errors.
- They defined mutation on statement level with addition, deletion or swapping. Wouldn't it be more effective to also introduce statement modication? For instance by modifying/swapping variables in statements? When we have something like "b = b - a" then do some mutation to get "a = b - a"? Or if we have a "if a == null" then mutate to "if b == null". Maybe this could be useful for a certain kind of errors.
- It would be interesting to know how much time the approach takes on a rather large (complete) set of test cases. In the evaluation they just manually create or select a small number of test cases which is sufficient to fix the error. Then they state that the approach can fix errors in just a few minutes.