University of Passau

Dr. Mohammad Rezaalipour
Prof. Dr.-Ing. Christian Hammer

Automated Program Repair

Task 3: Project and Report

Required Task **Deadline: 07.02.2026 23:59**

The current task is the **final required task of the course, and it will directly affect your final grade**. Failure to complete it will result in **automatic failure of the course**.

This task has three main goals:

- Implementing an automated program repair (APR) tool as your group project (Section 1).
- Writing a report for your group project (Section 2).
- Signing the *Statement of Originality* document (Section 3).

Please read this document carefully, pay close attention to the deadline above, and ensure the submission (Section 5) is made on time. Redistribution of this file is prohibited.

1 Project Specification

In this task, you will implement a version of GenProg, as described in [1], as a tool. Specifically, you are expected to implement the high-level pseudocode in Fig. 2 of [1], which is also shown here in Figure 1. Note that implementing this tool is a group task and all members must participate.

```
Input:  Program $P$ to be repaired.
Input:  Set of positive test cases $PosT$.
Input:  Set of negative test cases $NegT$.
Input:  Fitness function $f$.
Input:  Variant population size pop_size.
Output: Repaired program variant.
 1: $Path_{PosT} \leftarrow \bigcup_{p \in PosT}$ statements visited by $P(p)$
 2: $Path_{NegT} \leftarrow \bigcup_{n \in NegT}$ statements visited by $P(n)$
 3: $Path \leftarrow$ set_weights$(Path_{NegT}, Path_{PosT})$
 4: $Popul \leftarrow$ initial_population$(P,$ pop_size$)$
 5: repeat
 6:     $Viable \leftarrow \{\langle P, Path_P \rangle \in Popul \mid f(P) > 0\}$
 7:     $Popul \leftarrow \emptyset$
 8:     $NewPop \leftarrow \emptyset$
 9:     for all $\langle p_1, p_2 \rangle \in$ select$(Viable, f,$ pop_size$/2)$ do
10:         $\langle c_1, c_2 \rangle \leftarrow$ crossover$(p_1, p_2)$
11:         $NewPop \leftarrow NewPop \cup \{p_1, p_2, c_1, c_2\}$
12:     end for
13:     for all $\langle V, Path_V \rangle \in NewPop$ do
14:         $Popul \leftarrow Popul \cup \{$mutate$(V, Path_V)\}$
15:     end for
16: until $f(V) =$ max_fitness for some $V$ contained in $Popul$
17: return minimize$(V, P, PosT, NegT)$
```

Figure 1: High-level GenProg pseudocode (snapshot of Fig. 2 in [1]).

You may implement your tool in any programming language of your choice (e.g., Java, Python, C/C++, JavaScript). However, the implemented tool must target programs written in either Java or Python. You may choose either of these two languages; that is, your tool is expected to repair bugs in Java or Python programs, but support for both is not required. The rest of this section provides more details about the required implementation.

1.1 Simplifying Assumptions

To make the current task feasible, you may assume that your tool fixes **single-file buggy programs** (benchmarks), meaning that all the functionality of the buggy program to which your tool is applied is contained within a single file. You are welcome to go beyond this assumption and make your tool more powerful by supporting buggy programs with multiple files, modules, packages, etc.; however, for the purpose of this project, the single-file assumption for benchmarks is sufficient.

Note that we do not expect you to implement a tool capable of fixing bugs in real-world projects such as those in Defects4J or BugsInPy. Also, your tool can have a **simple interface**, such as a command-line tool.

GenProg has a minimization step, in which the generated patch is reduced in size (line 17 in Fig. 2 of [1]). You may ignore this step in your implementation; **your patches do not need to be minimized**.

You are also not expected to support all features of the programming language you target (i.e., Java or Python). **You are allowed to make simplifying assumptions**, as long as they are justified and documented. If you make any simplifying assumptions (for instance, if your tool does not support certain language features) document these limitations in your project report (Section 2).

1.2 Benchmarks

You must prepare five benchmarks that your APR tool supports. These benchmarks must also be included in your submission (Section 5). We recommend preparing these benchmarks before starting the implementation of your tool, since you can use them to test your tool during development.

The benchmarks do not need to be large; they can be small, simple programs of 5, 10, or 20 lines of code, or longer if you prefer.

For each benchmark, you must provide the following:

- A buggy version containing **a single bug** that your APR tool is expected to fix.
- A fixed version that represents the intended behavior of the benchmark.
- At least one failing test that fails on the buggy version (revealing the bug) and passes on the fixed version.
- Some passing tests that pass on both the buggy and the fixed versions.
- Sufficient information in your report (Section 2) about the benchmark, including the bug in the buggy version and its location, as well as the expected patch in the fixed version and its location.
- Sufficient information in your report (Section 2) describing how to run the tests for the benchmark, using both the buggy and fixed versions.

Your tool must not crash when applied to the benchmarks you provide. Given the stochastic nature of GenProg, patches may not always be generated. However, your tool must be able to fix at least one of your benchmarks in some runs; this affects grading (Section 4).

Note that your tool must not be implemented in a way that is tied specifically to your benchmarks. Instructors should be able to create new benchmarks similar to yours (and consistent with your simplifying assumptions) and apply your tool to them.

1.3 Fault Localization

For the fault localization step, which involves constructing the *weighted path* (lines 1–3 in Fig. 2 of [1]), you may use any external tools you can find; implementing fault localization yourself is **not** required.

You may even perform fault localization manually for your benchmarks (Section 1.2), store the information in a file, and keep that file next to the corresponding benchmark so that your tool can read it and obtain the fault localization data. This is only a suggestion, however. The key point is that implementing fault localization is not required, although you may choose to do so if you wish.

However, note that assuming perfect fault localization is **not** allowed. Also, you must apply the same fault localization weighting scheme introduced in [1], as defined below:

- Statements covered only by failing tests are assigned a weight of 1.
- Statements covered by both failing and passing tests are assigned a weight of 0.1.
- All other statements are assigned a weight of 0.

1.4 GenProg Parameters

In the GenProg algorithm described in [1], several parameters must be configured. You are free to choose these parameters; however, we recommend setting them according to Table 1. These values are taken from [1] and are recommend by the authors. For the selection operator, you may use either *Stochastic Universal Sampling* or *Tournament Selection*, similar to [1].

Parameter	**Value**
Population size (pop_size)	40
Positive test weight (W_{PosT})	1
Negative test weight (W_{NegT})	10
Mutation weight (W_{mut})	0.06

Table 1: Suggested GenProg parameter settings based on [1].

2 Project Report

Preparing the project report is also a group task, and all members must participate. Your project report must be submitted as a PDF file and must **not** exceed 20 pages of content. The project report must include the following:

- Name of all group members.
- Group slot name on Stud.IP (e.g., Group 16).
- The contributions of each group member.
- Sufficient documentation, including your project structure and information on where to find the different parts of your implementation.
- Answers to the questions in Section 2.1.
- Instructions on how to run your project (Section 2.2).

- Sufficient information about your benchmarks (Section 1.2), along with instructions on how to create new benchmarks similar to yours (consistent with your simplifying assumptions) and apply your tool to them.

You may include as many README, documentation, or Markdown files in your project structure as you wish. However, you should not expect the instructors to read all of them. The project report is the only file that will be read in full by the instructors, so make sure it contains all necessary information.

2.1 Questions

Please answer the following questions in your project report:

1. What were the most challenging parts of implementing this project? Please explain.
2. Does your implementation run and behave as expected? If not, what was the issue?
3. Did your tool manage to fix any of your benchmarks? If yes, which ones, and include the patch or patches.
4. Were there any benchmarks you created that were not fixed by your tool? If yes, what was the reason?
5. What are the limitations of your implementation (e.g., which simplifying assumptions did you make)?
6. How did you test your tool?

2.2 Environment Setup and Testing

It is your responsibility to ensure you have provided sufficient automated scripts and tests for your implementation to facilitate running and testing your project. You must provide clear instructions in your project report on how to set up the environment and run your project. Note that compilation errors will not be accepted.

If your project requires any API Keys (e.g., OpenAI API Key), **you must provide them**; otherwise, testing cannot be done and this may affect grading. Ensure that the keys remain active at least six month after the submission deadline. You may remove or deactivate the keys after you have received your final grade.

3 Statement of Originality

A document named "Task03_Originality_Declaration.pdf" will be released on Stud.IP. Please read the document carefully before starting your project. You must print the document, and all group members must sign the hard copy. Deliver the signed hard copy to the secretary of the Chair of Software Engineering I,[1] and include a scanned copy of the signed document in your submission, as described in Section 5.

4 Grading

The factors that affect your grade include, but are not limited to, the following:

- Whether your implementation runs correctly and behaves as expected.
- Whether your tool was able to fix at least one of your benchmarks.

[1]https://www.fim.uni-passau.de/software-engineering-i/lehrstuhlteam

- The quality of your tests and automated scripts, as described in Section 2.2.
- Your project report (Section 2).
- Code cleanliness and how well you structured your code.

5 Submission

To make this submission, create a ZIP file (no password) containing the following:

- The directory containing your source code, including your benchmarks.[2]
- Your project report PDF file (Section 2).
- The Statement of Originality document (Section 3) signed by all group members.

The name of the submission file must be "Project_Group_*ID*.zip" where *ID* is your group number (e.g., Project_Group_16.zip). Submit this file on Stud.IP by uploading it to directory *SubmissionsDir*.[3] The submission must be made by the group representative.

References

[1] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[2] If your submission file is too large, you can remove the ".git" directory from the root of your project directory to reduce the file size.

[3] This directory is write-only, so you may not be able to see your submission.