MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation

Pedro Orvalho*
pmorvalho@tecnico.ulisboa.pt
INESC-ID/IST - U. Lisboa
Lisboa, Portugal

Mikoláš Janota
mikolas.janota@cvut.cz
Czech Technical University in Prague
Prague, Czech Republic

Vasco Manquinho
vasco.manquinho@tecnico.ulisboa.pt
INESC-ID/IST - U. Lisboa
Lisboa, Portugal

ABSTRACT

There has been a growing interest, over the last few years, in the topic of automated program repair applied to fixing introductory programming assignments (IPAs). However, the datasets of IPAs publicly available tend to be small and with no valuable annotations about the defects of each program. Small datasets are not very useful for program repair tools that rely on machine learning models. Furthermore, a large diversity of correct implementations allows computing a smaller set of repairs to fix a given incorrect program rather than always using the same set of correct implementations for a given IPA. For these reasons, there has been an increasing demand for the task of augmenting IPAs benchmarks.

This paper presents MultIPAs, a program transformation tool that can augment IPAs benchmarks by: (1) applying six syntactic mutations that conserve the program's semantics and (2) applying three semantic mutilations that introduce faults in the IPAs. Moreover, we demonstrate the usefulness of MultIPAs by augmenting with millions of programs two publicly available benchmarks of programs written in the C language, and also by generating an extensive benchmark of semantically incorrect programs.

CCS CONCEPTS

• **Applied computing** → *Computer-assisted instruction*; • **Theory of computation** → **Program semantics**; **Program analysis**; **Program reasoning**; • **Computing methodologies** → Machine learning.

KEYWORDS

Automated Program Repair, Program Transformation, Data Augmentation, Introductory Programming Assignments, MOOCs

ACM Reference Format:
Pedro Orvalho, Mikoláš Janota, and Vasco Manquinho. 2022. MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3540250.3558931

*This work was done while this author was visiting CIIRC, CTU in Prague.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558931

1 INTRODUCTION

The increasing demand for programming education has given rise to all kinds of online evaluations, such as Massive Open Online Courses (MOOCs) [5] focused on introductory programming assignments (IPAs). Providing feedback to novice students in IPAs requires substantial effort and time by the faculty. Therefore, automated program repair has become crucial to provide automatic personalized feedback to each student [22]. Over the last few years, several program repair tools [4, 5, 9, 21, 23] have exploited previously enrolled students to obtain diverse correct implementations for each IPA. Given an incorrect student submission, these frameworks find the most similar correct submission from previous years to provide a minimal set of repairs to the student. Typically, having a similar correct implementation allows computing a smaller set of repairs to fix a given incorrect program rather than always using the set of repairs needed to make the incorrect submission semantically equivalent to a fixed reference solution. Furthermore, an increasing body of research has focused on applying machine learning (ML) models to automated program repair [2, 3, 6–8, 11, 14, 16, 17, 20, 22]. These ML-based tools depend greatly on the existence of many correct/incorrect programs to train their repair models. However, in most cases, the publicly available benchmark sets [13, 18] of students' submissions for IPAs are small, i.e., only hundreds of submissions. Hence, these benchmarks might not be sufficient to effectively train an ML model. Additionally, another problem with some real-world IPAs datasets is that sometimes there is no knowledge about the number and types of defects present in each incorrect student program, which can also negatively impact the training of ML-based program repair tools.

Hence, there is an increasing demand for data augmentation of program benchmarks to (1) achieve minimal sets of program patches by having a more diverse collection of syntactically different correct solutions and (2) have a more representative dataset of programs to train ML-based program repair tools with labelled incorrect programs. This data augmentation task aims to enlarge the real-world datasets of students' programs with more semantically correct implementations for each IPA by syntactically mutating existent correct students' submissions and to create a labelled dataset of incorrect programs with the information about the number and the type of the bugs present in each incorrect program.

Thus, this paper presents MultIPAs, a tool that performs data augmentation by syntactically mutating and/or semantically mutilating IPAs written in the C programming language. The main goal of MultIPAs is to augment IPAs benchmarks with: (1) more semantically correct implementations by applying six different syntactic mutations to pre-existent correct implementations and (2) new semantically incorrect programs by mutilating pre-existent correct

implementations. MultIPAs stores the variable mapping between the original correct implementation and the new mutated/mutilated program. Moreover, for the newly generated set of incorrect programs, MultIPAs also stores the information about the bugs in these programs. Later, these bug(s) annotations can be used to train ML models.

Experimental results show that MultIPAs can augment small-sized publicly available benchmarks of IPAs, ITSP [23] and C-Pack-IPAs [13], generating millions of mutated/mutilated programs. To summarize, this paper makes the following contributions:

- We present MultIPAs, a program transformation framework capable of augmenting small imperative program benchmarks by performing six different syntactic program mutations and three semantic program mutilations;
- MultIPAs is publicly available on GitHub: https://github.com/p-morvalho/MultIPAs, with a demo video at https://arsr.inesc-id.pt/ pmorvalho/MultIPAs-demo.html.
- MultIPAs keeps the information about the types and the number of bugs present in each generated incorrect program, which can be used to train ML-based program repair frameworks;
- MultIPAs produces a variable mapping between the original program given as input and the mutated/mutilated program.

2 MULTIPAS

This section presents MultIPAs, a new tool capable of augmenting IPAs benchmark sets by applying syntactic mutations and semantic mutilations to C programs. MultIPAs is divided into two modules: program mutator and program mutilator. The C programs are parsed and the changes (program mutations and mutilations) happen at the AST level. Section 2.1 presents the six different syntactic program mutations that MultIPAs can perform to change a program syntax while preserving its semantics. Afterwards, Section 2.2 explains three different semantic program mutilations that introduce semantic bugs in an IPA. Finally, Section 2.3 explains briefly the variable mappings produced by MultIPAs. MultIPAs is publicly available on GitHub: https://github.com/pmorvalho/MultIPAs and there is also a demonstration video available at https://arsr.inesc-id.pt/ pmorvalho/MultIPAs-demo.html.

2.1 Program Mutator

The goal of automated program repair when applied to IPAs is to achieve the best possible set of repairs (i.e., program patches) to fix a given student's incorrect submission for a programming exercise. The best repair is usually described as a minimal set of fixes required to make the student's program compliant with the test suite that describes the desired semantic behaviour for that specific IPA. To this end, many program repair tools, such as Verifix [1, 5], try to align the student's submission's control flow graph with another correct submission's control flow graph. Next, these frameworks propose a set of syntactic patches to fix the incorrect program. Hence, applying program mutations to an IPAs benchmark increases the number of different syntactic structures and allows program repair tools to achieve smaller sets of repairs. For that reason, MultIPAs can perform syntactic mutations to a program such that it preserves the program semantics, i.e., both programs, the original and the mutated, have the same behaviour.

The six syntactic program mutations available on MultIPAs are:

- *M1 - Comparison Expression Mirroring (CEM)*: MultIPAs mirrors one or several comparison expressions e.g. $a \geq b$ becomes $b \leq a$;
- *M2 - If-else-statements Swapping (IES)*: MultIPAs swaps the if-branch and the else-branch and negates the if-condition. This operation is done only for simple if-else-statements, i.e., there are no additional if-statements inside the else-branch;
- *M3 - Increment/Decrement Operators Mirroring (IOM)*: MultIPAs mirrors the two increment (and decrement) operators in the C programming language (e.g. c++ and ++c), only when the return value of the expression that contains the increment/decrement operator is discarded e.g. the increment step of a for-loop;
- *M4 - Variable Declarations Reordering (VDR)*: MultIPAs reorders the variables' declarations present in each code block. For this, MultIPAs takes into account the dependencies between the variables' declarations, i.e. if a variable declaration depends on other variables, this is done by computing all possible topological orders of the variables' declarations;
- *M5 - For-2-While Translation (F2W)*: MultIPAs translates for-loops into while-loops. Just in cases of for-loops that do not contain any continue instructions;
- *M6 - Variable Addition (VA)*: MultIPAs introduces a new dummy variable declaration in the program. The mutated program does not have the same set of variables as the original program.

Example 2.1. Consider the two programs in Listings 1 and 2 in the C programming language. Both programs are semantically equivalent since both programs sum all the natural numbers from 1 to n and print the current accumulated value in each iteration. The program in Listing 2, the mutated program, is the result of applying all the program mutations available on MultIPAs to the program in Listing 1, the original program. Note that the comparison expression in the for-loop condition was mirrored (mutation *(1)*). Mutation *(2)* is not applicable since there is no if-else-statement. Regarding mutation *(3)*, one can see that the increment step of the for-loop was also mirrored, line 6 (resp. 9) in the original (resp. mutated) program. Furthermore, the mutated program has a different variable declaration order than the original program (mutation *(4)*). Moreover, the for-loop was translated into a similar while-loop corresponding to mutation *(5)*. Lastly, a dummy variable y was introduced in line 3 of the mutated program (mutation *(6)*).

```
1  int main(){
2    int n;
3    int i, s;
4    scanf("%d", &n);
5    s=0;
6    for(i=1; i<=n; i++){
7      s = s+i;
8      printf("%d\n",s);
9    }
10
11   printf("%d\n",s);
12   return 0;
13 }
```

Listing 1: Original program.

```
1  int main(){
2    int n, s, i, y;
3    scanf("%d", &n);
4    s=0;
5    i = 1;
6    while(n>=i){
7      s = s+i;
8      printf("%d\n",s);
9      ++i;
10   }
11   printf("%d\n",s);
12   return 0;
13 }
```

Listing 2: Mutated program.

2.2 Program Mutilator

In the development of program repair tools, there are two main concerns on using incorrect programs of IPAs datasets: (1) usually, there is no knowledge of how many errors are present in each buggy student program; and (2) since the number of semantic errors on each program is unknown, repair framework's developers cannot divide the set of the incorrect programs into subsets of programs with a specific number of semantic errors (e.g., a subset for programs with one semantic error, another subset for programs with two semantic errors, etc.). Furthermore, dividing the dataset of incorrect IPAs into subsets of different numbers or types of bugs can be important to train ML-based program repair tools [2, 3, 11, 20]. Therefore, having a program mutilator that creates a dataset of programs with a specific number of semantic bugs and only certain kinds of bugs becomes crucial. This way, developers of program repair tools' can evaluate the scalability of their frameworks in terms of the number of semantic errors present in each program and train their tools to repair specific families of bugs.

Thus MultIPAs also contains a program mutilator module. This program mutilator takes a set of students' submissions for a given IPA and alters each program to introduce *n* errors, *n* being passed as a parameter by the user. The errors introduced by MultIPAs are semantic mutilation which modifies the programs' semantics. The following three different program mutilations (bugs) are available on MultIPAs:

- *B1 - Wrong Comparison Operator (WCO)*: MultIPAs swaps an expression's comparison operators for some syntactically similar operator. For example, swaps the operator < for <=. MultIPAs can also swap > for >=, <= for <, >= for >, == for =, and != for ==;
- *B2 - Variable Misuse (VM)*: MultIPAs swaps a variable in the program by another variable of the same type. The resulting mutilated program can be compiled successfully since MultIPAs ensures that both variables are of the same type;
- *B3 - Assignment Deletion (AD)*: MultIPAs deletes an assignment expression in the program.

```
1  int main(){
2    int n;
3    int i, s;
4    scanf("%d", &n);
5    s=0;
6    for(i=1; i<=n; i++){
7      s = s+i;
8      printf("%d\n",s);
9    }
10
11   printf("%d\n",s);
12   return 0;
13 }
```

Listing 3: Original program.

```
1  int main(){
2    int n;
3    int i, s;
4    scanf("%d", &n);
5
6    for(i=1; i<n; i++){
7      s = s+i;
8      printf("%d\n",i);
9    }
10
11   printf("%d\n",s);
12   return 0;
13 }
```

Listing 4: Mutilated program.

Example 2.2. Consider the two programs in Listings 3 and 4 written in the C programming language. The program in Listing 3, hereafter the original program, has been already presented in Listing 1. The program in Listing 4, hereafter the mutilated program, is the result of applying all the program mutilations available in MultIPAs. The first mutilation is located in line 6 of the mutilated program where the operator <= was swapped by the operator <. Furthermore, the *variable misuse* mutilation was performed in line 8. Lastly, MultIPAs removed the assignment expression of value zero to variable `s` (line 5).

The first class of bugs, *wrong comparison operator*, is common among novice programmers [18] and has been used to evaluate ML-based program repair tools [3, 17]. The second family of bugs, *variable misuse*, is also common among students as well as among experienced programmers [10, 19]. This specific task has received a lot of attention from the ML research community [2, 3, 20, 25]. Lastly, the *assignment deletion* bug is also common among students [18]. In the previous example, it is likely for a novice student to forget to initialize variable `s`.

Bug mapping. For each mutilated program generated, MultIPAs stores the information about the location and the types of the bugs introduced in the program. This information can help train ML-based program repair frameworks.

2.3 Variable Mapping

Typically, semantic program repair tools [1, 5] repair an incorrect program using a correct implementation for the same IPA. In order to compare two programs, it is required a relation between both sets of variables. For example, consider the programs presented in Listings 3 and 4. In this case, having a mapping between both programs' variables lets the repair framework reason about which program modifications it should perform to fix the faulty program. Program modifications include the same variable being used in a different comparison expression, the variable being initialized in one program but not in the other one, etc. Moreover, the variable mapping can also be helpful for the task of *code adaption* where the repair framework tries to adapt all the variable names in a pasted snippet of code, copied from another program or a Stack Overflow post to the surrounding preexisting code [11].

Thus, every time MultIPAs mutates or mutilates a program, a mapping between the original program's set of variables and the mutated/mutilated program's sets of variables is generated. This variable mapping can help a program repair framework [5, 11] to find a minimal repair.

Example 2.3. MultIPAs would produce the following variable mapping between the set of variables of the programs in Listings 1 and 2: `{int i: int i; int n : int n; int s : int s; int y : UNK_VAR}`. Moreover, for the program in Listings 3 and 4 the variable mapping would be: `{int i: int i; int n : int n; int s : int s}`.

3 EVALUATION

The experimental results presented in this Section show the evaluation of MultIPAs on two publicly available small-sized datasets of IPAs: ITSP [23] and C-Pack-IPAs [13]. The evaluation consists of using MultIPAs to augment both benchmark sets by mutating or mutilating all correct programs. Table 1 shows the overall results of our evaluation. The first two columns in Table 1 show, for each dataset and for each lab class, the number of IPAs (**#IPAs** column)

Table 1: Number of programs that can be generated by MultIPAs using each different mutation or mutilation for two different small datasets of IPAs: ITSP [23] and C-Pack-IPAs [13].

			Mutations							Mutilations (Bugs)			
ITSP Dataset [23]	#IPAs	#Correct Submissions	M1 (CEM)	M2 (IES)	M3 (IOM)	M4 (VDR)	M5 (F2W)	M6 (VA)	All Mutations	B1 (WCO)	B2 (VM)	B3 (AD)	All Bugs
Lab3	4	67	1.25E+03	9.90E+01	6.70E+01	4.74E+05	6.70E+01	1.34E+02	6.03E+06	1.86E+02	4.51E+03	1.51E+02	4.71E+04
Lab4	8	125	3.99E+04	2.22E+02	3.15E+02	8.02E+05	2.41E+02	2.30E+02	1.90E+11	9.93E+02	1.20E+04	4.16E+02	7.12E+05
Lab5	8	90	1.06E+04	1.59E+02	5.13E+02	3.78E+05	4.45E+02	1.80E+02	3.07E+12	5.24E+02	4.43E+03	3.78E+02	1.52E+05
Lab6	8	87	1.94E+04	1.29E+02	5.36E+03	1.12E+06	1.45E+03	1.74E+02	9.75E+13	5.52E+02	6.32E+03	5.70E+02	4.02E+05
Total	**28**	**369**	**7.12E+04**	**6.09E+02**	**6.25E+03**	**2.77E+06**	**2.20E+03**	**7.18E+02**	**1.01E+14**	**2.26E+03**	**2.73E+04**	**1.52E+03**	**1.31E+06**
Cx-Pack-IPAs Dataset [13]	**#IPAs**	**#Correct Submissions**	**M1 (CEM)**	**M2 (IES)**	**M3 (IOM)**	**M4 (VDR)**	**M5 (F2W)**	**M6 (VA)**	**All Mutations**	**B1 (WCO)**	**B2 (VM)**	**B3 (AD)**	**All Bugs**
Lab02	10	316	1.04E+04	4.49E+02	4.88E+02	3.64E+06	4.07E+02	6.32E+02	1.72E+07	9.68E+02	9.71E+03	1.11E+03	2.93E+05
Lab03	7	145	3.21E+05	3.20E+02	1.09E+03	4.93E+04	6.07E+02	2.90E+02	1.48E+10	1.02E+03	4.94E+03	8.07E+02	3.94E+05
Lab04	8	192	2.83E+04	2.85E+02	1.97E+03	8.72E+03	1.28E+03	3.80E+02	1.93E+11	1.07E+03	5.58E+03	1.08E+03	2.39E+05
Total	**25**	**653**	**3.59E+05**	**1.05E+03**	**3.54E+03**	**3.70E+06**	**2.29E+03**	**1.30E+03**	**2.08E+11**	**3.06E+03**	**2.02E+04**	**2.99E+03**	**9.26E+05**

and the number of correct students' submissions (**#Correct Submissions** column). All of the experiments were conducted on an Intel(R) Xeon(R) Silver computer with 4210R CPUs @ 2.40GHz, using a memory limit of 64GB.

Mutating Programs. Table 1 shows the number of programs that can be generated by MultIPAs when applying each individual mutation described in Section 2.1. One can see that all the program mutations are able to augment at least 100% of both benchmarks. Furthermore, both mutations *M1 (CEM)*, comparison expression mirroring, and *M4 (VDR)*, variable declarations reordering, are able to augment both benchmarks with thousands of mutated programs. These program mutations produce so many programs since the IPAs in both benchmarks use more than one variable, and in several programming exercises, the students need to compare the values of different variables (comparison expressions). Hence, MultIPAs computes all possible mirroring combinations of the comparison expressions and all possible re-orderings of the variable declarations that are valid. Lastly, if the user asks MultIPAs to perform all six program mutations on both benchmarks, the number of mutated programs reaches several billions of programs.

Mutilating Programs. Regarding the program mutilations, the right-hand side of Table 1 shows the number of programs MultIPAs can generate using each different mutilation described in Section 2.2, or all of them together. All the program mutilations are able to generate a dataset with several thousands of incorrect programs. Mutilation *B2 (VM)*, variable misuse, is the mutilation that is able to generate more incorrect programs since typically there are many possibilities when MultIPAs is changing a variable occurrence for another variable.

User Configuration. The number of programs that can be generated by MultIPAs can reach several million. Therefore, MultIPAs has three flags available related to the total number of programs that can be generated. By default, MultIPAs generates only 20% of those programs. The user can choose a different percentage of programs to be generated using the flag `-p | -percentage_total_progs`. Instead of generating all the programs, MultIPAs chooses a sample of size `p`. The user can ask MultIPAs, with flag `-info`, to print the total number of mutated/mutilated programs for a given configuration of program mutations/mutilations. MultIPAs only outputs the number of programs without generating them. If the user desires all the programs and has the time and memory to generate all the possible mutated/mutilated programs, this can be done using the flag `-ea | -enumerate_all`.

4 RELATED WORK

In the last few years, there has been a growing interest in data augmentation by program transformation. Yu et al. [24] proposed to apply several program transformations for big code data augmentation based on a pre-defined set of syntax-based rules to mutate programs written in Java. Liu and Zhong [12] proposed to extract Java code samples from Stack Overflow, and mine repair patterns from the extracted code samples. DeepBugs [17] also uses rule-based mutations to build, and not to augment, a dataset of programs from scratch to train its ML-based program repair tool. BugLab [3] is a Python program repair framework that learns how to detect and fix minor bugs. In order to train BugLab, Allamanis et al. [3] applied four program mutations and four program mutilations, different than MultIPAs's program mutations and mutilations, in order to augment their benchmark of Python programs.

5 CONCLUSION

This paper introduces MultIPAs, an open-source framework for augmenting benchmarks of introductory programming assignments (IPAs). MultIPAs can generate semantically equivalent programs by applying up to six different syntax mutations to a given program. Furthermore, MultIPAs can also produce semantically incorrect programs using three semantic program mutilations. Moreover, MultIPAs saves the variable mappings between the original program and the mutated/mutilated one and the information about the bugs introduced in each program. Experiments on two publicly available datasets of IPAs show that MultIPAs can augment with millions of programs small-sized benchmarks of IPAs.

6 DATA AVAILABILITY STATEMENT

MultIPAs is publicly available in the ACM Digital Library [15].

ACKNOWLEDGMENTS

This research was supported by Fundação para a Ciência e Tecnologia (FCT) through grant SFRH/BD/07724/2020 and projects UIDB/50021/2020, PTDC/CCI-COM/32378/2017 and project ANI 045917 funded by FEDER and FCT.

REFERENCES

[1] Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh, and Abhik Roychoudhury. 2022. Verifix: Verified Repair of Programming Assignments. *ACM Trans. Softw. Eng. Methodol.* (jan 2022). https://doi.org/10.1145/3510418

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

[3] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-Supervised Bug Detection and Repair. In *NeurIPS*.

[4] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: automated data-driven synthesis of repairs for static analysis violations. In *ESEC/SIGSOFT FSE 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 613–624.

[5] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. In *PLDI 2018*. ACM, 465–480.

[6] Rahul Gupta, Aditya Kanade, and Shirish K. Shevade. 2019. Deep Reinforcement Learning for Syntactic Error Repair in Student Programs. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*. AAAI Press, 930–937.

[7] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *AAAI 2017*, Satinder P. Singh and Shaul Markovitch (Eds.). AAAI Press, 1345–1351.

[8] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

[9] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 388–398.

[10] Rafael-Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur?: The ManySStuBs4J Dataset. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup (Eds.). ACM, 573–577.

[11] Xiaoyu Liu, Jinu Jang, Neel Sundaresan, Miltiadis Allamanis, and Alexey Svyatkovskiy. 2022. AdaptivePaste: Code Adaptation through Learning Semantics-aware Variable Usage Representations. *CoRR* abs/2205.11023 (2022).

[12] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 118–129. https://doi.org/10.1109/SANER.2018.8330202

[13] Pedro Orvalho, Mikoláš Janota, and Vasco Manquinho. 2022. C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments. https://doi.org/10.48550/arXiv.2206.08768

[14] Pedro Orvalho, Mikoláš Janota, and Vasco Manquinho. 2022. InvAASTCluster: On Applying Invariant-Based Program Clustering to Introductory Programming Assignments. https://doi.org/10.48550/ARXIV.2206.14175

[15] Pedro Orvalho, Mikoláš Janota, and Vasco Manquinho. 2022. MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation (Artifact). https://doi.org/10.1145/3554335

[16] Pedro Orvalho, Jelle Piepenbrock, Mikoláš Janota, and Vasco Manquinho. 2022. Project Proposal: Learning Variable Mappings to Repair Programs. *7th Conference on Artificial Intelligence and Theorem Proving, AITP* (2022).

[17] Michael Pradel and Koushik Sen. 2018. DeepBugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 147:1–147:25.

[18] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 180–182.

[19] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2020. Learning to Fix Build Errors with Graph2Diff Neural Networks. In *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*. ACM, 19–20.

[20] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural Program Repair by Jointly Learning to Localize and Repair. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[21] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *PLDI 2018*. ACM, 481–495.

[22] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback. In *ICML 2020 (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 10799–10808.

[23] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *ESEC/FSE 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 740–751.

[24] Shiwen Yu, Ting Wang, and Ji Wang. 2022. Data Augmentation by Program Transformation. *J. Syst. Softw.* 190 (2022), 111304. https://doi.org/10.1016/j.jss.2022.111304

[25] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-Agnostic Representation Learning of Source Code from Structure and Context. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.