Analyzing the Impact of Patches in Computer Science Education

Chris Brown
North Carolina State University
dcbrow10@ncsu.edu

ABSTRACT

Programming assignments assigned in introductory computer science courses play a major role in declining retention rates in the field despite increased class sizes. Not every student can finish the assignments successfully, and those who struggle gradually lose interest without effective feedback and guidance. Lack of interest in computer science will hurt our society as our culture becomes more technology-dependent. This project examines using code patches for defects to solve prevalent problems in computer science education by providing valuable feedback and grading assignments in introductory programming classes.

Keywords

automated program repair, computer science education, assignment feedback, automated grading

1. INTRODUCTION

Debugging is a very time-consuming [17] and expensive [2] software development activity. Defects in code costs the economy billions of dollars each year and a large amount of developer time is consumed by manually finding and fixing bugs. Debugging and creating understandable and maintainable patches for defects is important for developing and sustaining usable software.

There has been increased research in automated program repair to help solve this problems for software engineers [1]. Researchers have developed various automated program repair tools and techniques have been introduced to provide efficient solutions to manual debugging and help reduce the cost and time of finding, fixing, and validating bugs in code by automating this process.

Automated program repair is an emerging research area in computer science, however there is limited research exploring applications of code patches and automated program repair outside of the software engineering industry. The systems developed for automated program repair can potentially be used to help solve more problems in the field. This project aims to analyze the impact of automated program repair and generated patches in other areas of computer science, specifically focusing on improving CS education.

[1]http://automated-program-repair.org

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

The ACM notes there is a "crisis" in computer science education, and predicted that a third of the technology jobs needed in 2018 will not be able to be filled as our society becomes more dependent on technology [19]. Computer science education needs improvements in order to produce more higher quality software engineers that have the knowledge and ability to develop, debug, and support software that impacts our daily lives. Automation has been shown to be effective in computer science classes [18], but this research is novel in that it applies automated program repair techniques to computer science education by analyzing the effect of information collected from patches on grading introductory programming assignments.

This research analyzes the impact automated program repair can have on computer science education by making the following contributions:

- Describing how automated program repair and bug patches can improve current issues in computer science education.
- Introducing a new algorithm for grading student assignments that incorporates information from bug patches.
- Evaluating the grading scheme on patches generated from introductory programming assignments.

The rest of this paper is organized as follows: Section 2 discusses the problem, Section 3 describes previous research related to automated program repair and computer science education, Section 4 introduces the proposed grading algorithm, Section 5 outlines the details of the experiment for evaluating the algorithm, Section 6 presents the results, Section 7 discusses the implications, implications, and future work concerning this research, and Section 8 concludes.

2. MOTIVATION

Computer Science education is facing two significant problems that contribute to the crisis and will impact the future of the field. The first is that enrollment in computer science classes is increasing drastically. Many introductory college programming courses accommodate hundreds of students. The students come from a wide variety of backgrounds and have different levels of previous programming experience.

The increase of class sizes make grading and assisting students even more challenging and increase the amount of pressure on the teaching staff. Large classes require more effective course management strategies [4] to avoid problems such as grading inconsistency and unproductive feedback. This increases the workload for professors and teaching assistants as they try to provide resources along with preparing and grading assignments.

Although enrollment in computer science classes is growing, studies have shown that the dropout rate also is increasing. The Computer Science major has a low retention rate among college students at 30-40% [1]. One main reason the field has a low retention rate is related to the programming projects assigned in introductory courses. Kay writes that specifications for programming assignments are "notoriously difficult" for novice programmers and course projects can be "voluminous and require painstaking evaluation" [4]. Not every student can understand and finish the assignments successfully, and those who struggle gradually lose interest without effective feedback and guidance.

This project aims to use information gathered from patches to improve both of these issues in computer science education. Grading with patches can save time and effort for teachers by efficiently analyzing and grading a large number of assignments. It can also help keep students interesting in computer science providing personalized evaluations and fairer grades to each student. The proposed patch grading formula uses data from patches to grade projects and the new algorithm is introduced in Section 4.

3. RELATED WORK

This project builds on previous research in patch generation and automation in computer science education.

3.1 Patch Generation

There have been many advances in research for automated program repair in software engineering. Previous research has introduced various tools for automated program repair including PAR [6], Prophet [9], Angelix [11], SemFix [12], TrpAutoRepair [13], SearchRepair [5], GenProg [7], and more. These tools utilize various techniques to automatically create patches for defects in source code.

Researchers have uncovered many benefits of patches generated from automated program repair in the software engineering industry. Zhong et al. performed an empirical study on bug fixes using BugStat to compare automatically generated patches compared to manual fixes by developers [20]. Le Goues and colleagues used GenProg on real-world C applications to reduce the cost of repairing 55 out of 105 defects to approximately \$8 each [7]. My project specifically focuses on the impact patches can have on improving computer science education.

Previous research studying patches have used student assignments for evaluation. Smith utilized the IntroClass benchmark to evaluate GenProg and TrpAutoRepair to study overfitting patches to test cases [15]. D'Antoni and colleagues evaluated Qlose on student submissions from an introductory programming course, but focused on observing syntactic and semantic distances between source code and tests in program repair [10]. These projects use introductory programming assignments to analyze the quality of generated patches while my research will evaluate the impact of patches from student assignments on improving computer science education.

3.2 Computer Science Education

Researchers have also examined various ways to improve assignments and feedback in computer science education. Wilcox analyzed the importance of automation in undergraduate computer science courses as enrollment increases [18]. He concluded that automated tools for grading, peer instruction, and tutorials can help improve student learning and help teaching staffs. My project aims to improve automatic grading by integrating data from patches to evaluate assignment submissions using additional grading criteria.

Kaleeswaran and colleagues created a new method *semi-supervised verified feedback generation* to automatically provide feedback for dynamic programming assignments [3]. Their tool CoderAssist divides student submissions into clusters based on solution strategies, requires the instructor to provide a solution to each cluster, and then compares the programs in each cluster to the given solution to provide personalized feedback for each student. This project focuses on using information gathered from patches to provide personalized evaluations to each student's assignment.

Singh et al. created an approach to automatically generate feedback for programming assignments using synthesis [14]. They evaluated their technique in an introductory Python programming class taught as a massive open online course (MOOC) at MIT and were able to provide automated feedback and solutions to thousands of students. This project aims to improve computer science education by automatically grading submissions based on bug patches in addition to focusing on student feedback.

4. PATCH GRADING ALGORITHM

This project introduces a new grading scheme to calculate more appropriate grades for assignment submissions based on fixes for errors. Programming assignments have mostly been graded by manually running student submissions against a test suite provided by the instructor for grading. The grade is normally determined using a basic formula that divides the number of passing tests by the total number of test cases:

$$Grade = \frac{\sum_{i=1}^{n} g(t_i)}{n}$$

$$g_{\mathrm{b}}(t) = \begin{cases} 1 & \text{if pass,} \\ 0 & \text{if fail and no fix} \end{cases}$$

This research proposes a new method for grading that improves on traditional methods by utilizing information from patches to evaluate assignments. The patch grading scheme uses a weighted average to calculate a grade based on two components, edit distance and class performance:

$$g_{\mathrm{p}}(t) = \begin{cases} 1 & \text{if pass,} \\ (0.5) * (1 - \frac{\text{edit}}{\max(\text{len1, len2})}) + (0.5) * \frac{\text{defects}}{\text{students}} & \text{if fail and fix,} \\ 0 & \text{if fail and no fix} \end{cases}$$

where n represents the total number of tests, t_i is the ith test case, $defects$ is the number of submissions that failed the same test case, $students$ is the total number of students, *edit* is the Levenshtein distance[2] between the code with and without the patch, and *len*1 and *len*2 are the length of the program before and after the patch is applied. The edit distance is calculated using the percent match between the buggy code and the patched version. Class performance incorporates the percentage of students in the class who had the same failing test case in their submission, which can represent confusion in the project requirements or implementation.

The advantages of the new method are that students will be evaluated based on how close they were to the correct implementation and how the rest of the class performed on the same assignment. Patch grading prevents all submissions from being penalized the same amount for failing test cases. For example, two students can submit code that fail the same test cases, but in one case the student may have only made a typo in their submission while the other student did not implement the functionality in their program. If the grader uses g_b to evaluate the assignments, then both students will receive the same grade. However, if g_p is used for grading then the first student will receive credit for trying and putting effort into the the code submitted instead of treating both versions equally.

The above formulas were implemented as Python scripts [3], which were then evaluated and run on student submissions to assignments from an introductory C programming class. The study was designed to analyze the following research questions:

RQ1 How often do grades calculated from patches differ from basic grading?

RQ2 How do human patches differ from automatically generated patches when calculating grades?

The first research question aims to determine if using information from patches can improve grades evaluated only using the number of passing and failing tests. The second question investigates how automatically generated patches compare to developer patches in the context of grading assignments.

5. METHODOLOGY

This section describes the evaluation benchmark, automated program repair tool, and experimental details used to complete this research study.

5.1 IntroClass

The patch grading algorithm was evaluated using the IntroClass[4] program repair benchmark [8]. IntroClass is a collection of student programs from an introductory C programming class at UC Davis. The benchmark data consists of 1,143 submissions from 259 students for six different projects. It also provides black and white box test cases for evaluating correctness. This study uses both sets of test cases for grading in increase the amount of patches available to use in the study.

[2]https://rosettacode.org/wiki/Levenshtein_distance

[3]https://github.ncsu.edu/dcbrow10/auto-grader/tree/CSC791

[4]https://github.com/ProgramRepair/IntroClass

5.2 SearchRepair

SearchRepair [5] is an automated program repair tool that implements *semantic code search* [16]. This technique uses the input-output behavior of a program to index code in a database and search for fragments with the desired functionality. After finding match among potential candidate programs using symbolic execution, the tool synthesizes the results and applies the fix to produce a correct version of the code as output. Compared to other automated program repair techniques, semantic code search creates higher quality patches that generalize to the expected program behavior and avoid deleting or modifying behavior by overfitting to the test cases.

5.3 Study Design

The experiment was designed to analyze the impact of using information from patches in grading programming assignments. The base and patch grading formulas from Section 4 were implemented and run on projects in the IntroClass benchmark to automatically grade assignments using only the test cases, data from SearchRepair patches, and data from student patches. IntroClass contains multiple student submissions for each project, which were used to simulate human patches in this experiment. Projects with code that did not compile were discarded in the study, leaving 933 total assignments used in the evaluation.

6. RESULTS

6.1 Base Grading vs. Patch Grading

RQ1 aims to compare grades calculated with and without patches. Table 1 displays the average grades from the results of evaluating IntroClass projects using the basic grading formula g_b and the new patch grading formula g_p presented in Section 4. Patch grading with g_p was calculated using an average of patches generated from the SearchRepair automated program repair tool and patches written by students. Each project saw an increase in the average grade students earned when patches were used for grading, however the difference was not statistically significant. The statistical analysis was completed using a oneway analysis of variance t-test to compare the grade values with and without a patch ($p = 0.3321$). Still, there is a noticeable difference when using data from patches to grade assignments and project grades for students increased by an average of approximately 4.47 points per submission when using patches for the evaluation.

6.2 Automated Patches vs. Human Patches

RQ2 focused on comparing automatically generated patches to patches written by humans. Table 2shows the number of student patches greatly outweighed the number of patches generated by SearchRepair. The study compared 930 human-written patches to 341 automatically generated patches. This is expected since student pushes to their repository were used to generate human patches and automated program repair tools are not able to find and fix defects at the same rate as human developers.

Table 3 provides an outline of how automated patches and human patches both compare to no patch. There was no difference between the average grades calculated using no patches and the SearchRepair patches for the first two projects because the tool did not produce any patches for digits and only generated one patch for *checksum*, which did

Project	No Patch	Patch
checksum	74.89	78.40
digits	86.70	88.07
grade	72.22	77.08
median	80.75	85.30
smallest	64.13	74.00
syllables	68.15	70.51
Overall	74.47	78.89

Table 1: Average grades calculated with and without patches

Project	Automated	Human
checksum	1	57
digits	0	164
grade	118	232
median	95	197
smallest	108	143
syllables	19	137
Overall	341	79.04

Table 2: Number of patches for each patch generation type

not make an impact in the overall average grade. However, as the number of automatically generated patches increased number automated patches the patch grading algorithm improved grades at a higher rates than the human patches.

The average difference between grades using the two different types of patches was 1.76 points in favor of human-written patches. Again, using the oneway analysis of variance t-test the results show that there is not statistically significant difference between using human or machine generated patches when grading student assignments (p = 0.9435).

7. DISCUSSION

This section discusses implications of the results, limitations of this research, and future work.

7.1 Impact

Using patches to grade assignments can be beneficial for students and instructors in introductory programming classes. Patch grading can impact the problems of increasing class sizes and drop out rates among universities to improve computer science education.

7.1.1 Enrollment

Implementing patch grading can benefit instructors by providing a scalable solution to automatically evaluating student assignments as enrollment in introductory programming courses rises. The mean time to calculate the grades using information from patches in our experiment was approximately 14 minutes (846 s) for each IntroClass project ranging from approximately 2.5 minutes (*checksum*) to 30 minutes (*grade*). Although the average time for grading without using patches was shorter around 8 minutes (476.5), utilizing patch grading is much more efficient than manually inspecting programs to analyze additional evaluation criteria such as edit distance and class performance for each student submission, and provides better results.

Project	No Patch	Automated	Human
checksum	74.89	74.89	81.90
digits	86.70	86.70	89.44
grade	72.22	77.92	76.24
median	80.75	86.13	84.47
smallest	64.13	76.43	71.57
syllables	68.15	70.40	70.61
Overall	74.47	78.75	79.04

Table 3: Average grades calculated with no patch, patches from SearchRepair, and student patches

Grade	No Patch	Automated	Human
A	323	422	385
B	190	211	197
C	71	63	76
D	82	53	58
F	267	184	217

Table 4: Distribution of letter grades generated with no patch, automated patches, and human patches

7.1.2 Retention

Patch grading can also help increase the retention rate in computer science by improving student grades on assignments. Frustrations with grades and feedback on assignments can discourage beginning programmers from continuing in CS, and patch grading significantly improved the evaluations of student projects compared to not using patches. Table 4 shows the distribution of letter grades from the experiment. Patch grading calculated at least the same grade for both types of patches, and when patches were available then less students failed and they performed better overall in general. The impact of patch grading for class sizes can also benefit students in increase retention by providing more time for the teaching staff to be available to work with students and help with questions.

7.2 Limitations

There are several threats to the validity of this evaluation and disadvantages of the proposed patch grading formula. Project submissions were used as human-written patches in the experiment, but student pushes to their project repository are not always representative of a patch to fix a bug. The grades calculated may also be skewed lower because of early submissions from students while they work on their implementation that will fail more test cases and account for the high number of F's observed. Failing test cases were used to determine students with the same error, but it's possible student submissions can fail the same test cases but have different problems in the code. A disadvantage of using IntroClass is that some submissions are non-deterministic and are able to produce different results when they are run, which could change the grading because test results can vary. Patches from SearchRepair were used to represent automated program repair tools, but different program repair tools and techniques use different methods to find bugs and generate patches.

Disadvantages of the patch grading algorithm include dependencies on code size and an inability to generalize to other patch generation tools. The percent match and Levenshtein distance utilize the number of characters in the code, which could penalize student submissions with more lines of code and longer variable or method names. The formula also requires the generated patches to be applied to the source code, but not all automated program repair tools present patches in the same way, such as Angelix [11] which generates .patch files that show changes similar in a format only displaying the difference of what lines should be changed by adding and removing from the original code.

7.3 Future Work

Future work includes improving the patch grading formula and implementing it into an automated program repair tool. The edit distance in the proposed algorithm can be improved by using tokens instead of the text to calculate the edit distance, which would prevent code size from factoring into the grades. Class performance will be improved by making advancements to the search for similar defects. Rather than just using failing test cases, patch grading can be used to identify submissions that have similar errors in the code. This information would be used to provide better feedback to instructors on what programming concepts and class materials students are having trouble understanding and may not have been clear in the lecture.

Future research also involves improving patch grading to include more evaluation criteria that instructors may expect from students in their classes, such as documentation and comments, student-written unit tests, code style, design, etc. Efforts can also be made to the implementation used in this study to improve the performance and efficiency, make the tool more usable, and modifying the script to allow it to generalize to different projects and folder structures other than IntroClass.

Another research area to develop for patch grading is how to determine the actual knowledge of a student, such as differentiating between a typo and confusion on a programming concept, and what information students should receive in their customized feedback to help them increase their programming knowledge and become better and software engineers. Assignment grading and feedback to students should contain information that will be useful to them not only for the class but also for their future careers as developers.

The patch grading formula will also be implemented into an automated program repair tool that will be able to automatically repair, grade, and provide feedback on assignments to improve the experience for students and teachers in introductory programming courses. Teachers will receive feedback on the performance of the class and students will receive patches to view correct implementations of their program. The tool will be evaluated on student projects and integrated it into introductory programming courses to analyze it's impact on instructor effort, feedback quality, and computer science retention rates.

8. CONCLUSION

Patches can impact computer science education by providing information to improve grading introductory programming assignments. This research discussed two growing trends that are problematic for computer science education and for our society in general, introduced a new grading technique that incorporates information gather from patches to grade project submissions using edit distance and class performance, and implemented and evaluated the algorithm on introductory student programming assignments in C. The patch grading formula was able to increase the performance of students on their submissions on average and compare the results with automatically generated and human-written patches. The automated grading technique can assist teachers in grading a large number of student assignments and the evaluation based on patches can calculate fairer and personalized grades for students to help reduce frustrations and increase their interest in computer science.

9. ACKNOWLEDGEMENTS

Special thanks to Yalin Ke for providing the SearchRepair patches for IntroClass at the last minute, Dr. Stolee for teaching the class and extending the final deadline, and Dr. Stolee and Dr. Jin for their advising and feedback on this project.

10. REFERENCES

[1] T. Beaubouef and J. Mason. Why the high attrition rate for computer science students: Some thoughts and observations. *SIGCSE Bull.*, 37(2):103–106, June 2005.

[2] T. Britton, L. Jeng, G. Carver, and P. Cheak. Reversible debugging software âĂIJquantify the time and cost saved using reversible debuggersâĂİ, 2013.

[3] S. Kaleeswaran, A. Santhiar, A. Kanade, and S. Gulwani. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 739–750, New York, NY, USA, 2016. ACM.

[4] D. G. Kay. Large introductory computer science classes: Strategies for effective course management. In *Proceedings of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '98, pages 131–134, New York, NY, USA, 1998. ACM.

[5] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. Repairing programs with semantic code search (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 295–306, Washington, DC, USA, 2015. IEEE Computer Society.

[6] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press.

[7] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press.

[8] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering (TSE)*, 41(12):1236–1256, December 2015. DOI: 10.1109/TSE.2015.2454513.

[9] F. Long and M. Rinard. Automatic patch generation by learning correct code. *SIGPLAN Not.*, 51(1):298–312, Jan. 2016.

[10] R. S. Loris D'Antoni, Roopsha Samanta. Qlose: Program repair with quantiative objectives. In *27th International Conference on Computer Aided Verification (CAV 2016)*, July 2016.

[11] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM.

[12] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[13] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 180–189, Washington, DC, USA, 2013. IEEE Computer Society.

[14] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA, 2013. ACM.

[15] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 532–543, New York, NY, USA, 2015. ACM.

[16] K. T. Stolee, S. Elbaum, and D. Dobos. Solving the search for source code. *ACM Trans. Softw. Eng. Methodol.*, 23(3):26:1–26:45, June 2014.

[17] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 1–, Washington, DC, USA, 2007. IEEE Computer Society.

[18] C. Wilcox. The role of automation in undergraduate computer science education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 90–95, New York, NY, USA, 2015. ACM.

[19] C. Wilson, A. for Computing Machinery, and C. S. T. Association. *Running the Empty: Failure to Teach K-12 Computer Science in the Digital Age.* Association for Computing Machinery, 2010.

[20] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 913–923, Piscataway, NJ, USA, 2015. IEEE Press.