Commit-min: Minimize Bug Fix Commits

Jacob Ward: jward23; Evan Blajev: eblajev, Sarah Zhou: sarahsz,
Julius Christenson: juliusc, Conner Knight: connerk

Problem & Motivation
Commits in a version control system should have a single purpose. However in the real world, commits often include extra code or refactorings beyond the purpose of the commit. For example, while a developer is working on finding a bug, they may also clean up surrounding code so it is easier to read and reason about. Once the bug has been fixed, the commit will include the code that fixed the bug as well as the extra refactoring. Ideally, our proposed tool, Commit-min, would separately commit only the lines of code that fixed a bug.

Large commits that encompass many changes can be problematic for future developers. When a developer needs to improve an old feature or revist a bug fix, they search the commit history, but with refactorings included in bug fix commits they often spend a large amount of time sifting through version control system diff statements to isolate the code that is significant. Commits consisting of only the necessary changes for a bug fix would improve a developer's workload since less time would need to be spent trying to understand old code, as it is more compartmentalized into individual, and manageable commits.

How developers address the problem today
Git provides a couple methods that require developer initiative to split up commits. The first is for unstaged code using "git add -patch" which interactively allows a developer to select "hunks" of code to stage for commit. The second is for code already committed using "git rebase -i" which allows the developer to walk through old commits and split them using the first method. These both work well if the developer already knows what lines of code fixed a bug. If they do not, there is a automated approach called delta debugging[1][2] that could help. Delta debugging is a process that can be automated in which you continuously narrow down the scope of a change set until you isolate exactly that lines that fixed or caused a bug. This process is effective on small change sets and is better than randomly removing lines, but it could be slow on large changes. We see delta debugging as a useful tool, but we want to remove as many unnecessary lines as possible first to speed up its execution. Git bisect, which can be improved by our tool, uses binary search to find the commit that caused a bug, instead of finding where in your newest commit you fixed a bug. Since our tool minimizes commits, the output will be a single commit, which will allow developers to search their commit history easily.

Our Approach
Commit-min will automate the bug minimization search process, freeing the developer from needing to manually decide which lines to include in a bug fix commit. Our tool denotes that a bug has been fixed based on some specified test passing that was previously failing. Our tool will attempt to reduce a commit by determining which lines of code can be removed from it while still allowing the test for the bug fix to pass.

Our tool will be a script that is executed from a Git pre-commit hook that will be placed into a user's repository on installation. If a user tries to commit, our tool will ask them if they would like to minimize their commit. If they say yes, our tool will run a two phase algorithm to minimize the bug fix as described below. Once it has found the minimized commit, it will show the user the new commit and ask if they would like to commit it. If they do, our tool follows the commit procedure described in phase two below. Otherwise, the tool aborts and just commits the users original commit.

---

[1] Zeller, Andreas. "Yesterday, my program worked. Today, it does not. Why?." *ACM SIGSOFT Software engineering notes*. Vol. 24. No. 6. Springer-Verlag, 1999.
[2] Zeller, Andreas, and Ralf Hildebrandt. "Simplifying and isolating failure-inducing input." *IEEE Transactions on Software Engineering* 28.2 (2002): 183-200.

In order to get the commit diff in a usable format to be used by our program, we will use git diff for our tool. Since it has all the features necessary to extract what has changed about the project in the most recent commit, which is what we need to run our tool.

Commit-min aims to be successful by combining two simpler minimization steps into one cohesive tool. Each phase of the algorithm on its own would provide the developer with useful information to determine which lines of code were involved in the bug fix. Phase one eliminates changes that are basic refactorings, and phase two experimentally determines which lines are unnecessary. Combining two relevant approaches into one systematic tool should increase the speed of the delta debugging since less lines are under consideration.

The first phase looks for lines that are refactored. Our tool will determine the simple cases of logically equivalent code, such as new comments, whitespace, variable name changes and new lines. Figure 1 illustrates a type of refactoring. Plagiarism detection tools may help in finding if a commit is a refactor. In "*A comparison of plagiarism detection tools"*[3], we found and tested a helpful tool JPlag. This tool detects similarities by source code tokenization. We will run JPlag on any modified files in order to compare them to their previous version. This will allow us to notice that some modifications are simply refactoring, and avoid spending additional computational time analyzing those files for bug fixes.

```
public void insertTuple(Tuple t) throws DbException {
    // some code goes here
    // not necessary for lab1
    if(!t.getTupleDesc().equals(td)){
         throw new DbException("mismatch");
    }

    for(int i = 0; i < numSlots; i++){
        if(!isSlotUsed(i)){
            markSlotUsed(i, true);
            t.setRecordId((new RecordId(pid, i)));
            tuples[i] = t;
            return;
        }
    }
    throw new DbException("full");
}
```

```
public void insertTuple(Tuple t) throws DbException {
    // some code goes here
    // not necessary for lab1
    if(!t.getTupleDesc().equals(td)){
         throw new DbException("mismatch");
    }

    refactor(t);

    throw new DbException("full");
}

public void refactor(Tuple t){
    for(int i = 0; i < numSlots; i++){
        if(!isSlotUsed(i)){
            markSlotUsed(i, true);
            t.setRecordId((new RecordId(pid, i)));
            tuples[i] = t;
            return;
        }
    }
}
```

Figure 1, Phase 1: Refactoring that would be caught and removed

The second phase will remove subsets of changed lines until it finds the largest subset of lines it can remove without causing the test to fail. This will happen in place, meaning it will modify the original file. An approach to determine the 'minimum change set' is delta debugging, which essentially takes a binary search approach to removing lines and seeing if they cause a failure. We found an interesting paper on delta debugging about AutoFlow[4], but unfortunately we couldn't find the actual AutoFlow tool anywhere,

---

[3] Jurrian Hage, Peter Rademaker, Nike van Vugt, "A comparison of plagiarism detection tools", *Technical Report 2010.* Untrecht Univeristy, Department of Information and Computing Sciences, 2010, ISSN 0924-3275

[4] Sai Zhang, Zhongxian Gu, Yu Lin, Jianjun Zhao, "AutoFlow: An automatic debugging tool for AspectJ software", *Software Maintenance 2008. ICSM 2008. IEEE International Conference on*, pp. 470-471, 2008, ISSN 1063-6773.

only research papers, so we decided to use another tool called Delta[5]. This tool uses a delta debugging algorithm to determine a minimal set of changes that are pertinent to a given test, in this case our tool would use the tests that were failing before but are now passing.

We modified Delta to work on the granularity of hunks instead of lines. Git has the notion of a hunk which is essentially some lines of code that are close together. We think it is reasonable to use hunks because typically if a developer modifies some lines of code that are close to each other, the changes are related. This assumption could reduce our accuracy, but it comes with the tradeoff of a significantly faster runtime since we will have fewer hunks than lines to consider.

To work with hunks, commit-min uses a tool called splitpatch[6] which takes a diff file and splits it into many diff files, one for each hunk. We modified this tool slightly so it would work for the formatting of "git diff" rather than linux "diff". Delta will then use these hunk files as the set of changes it is considering rather than the lines of a file. It will try combinations of applying some of the changes in the hunk file while leaving others out. Once delta has finished, commit-min will commit the bug fix. Finally, the backup files before commit-min was run will be brought back leaving the repository in a state with the bug fix committed and the irrelevant changes unstaged.

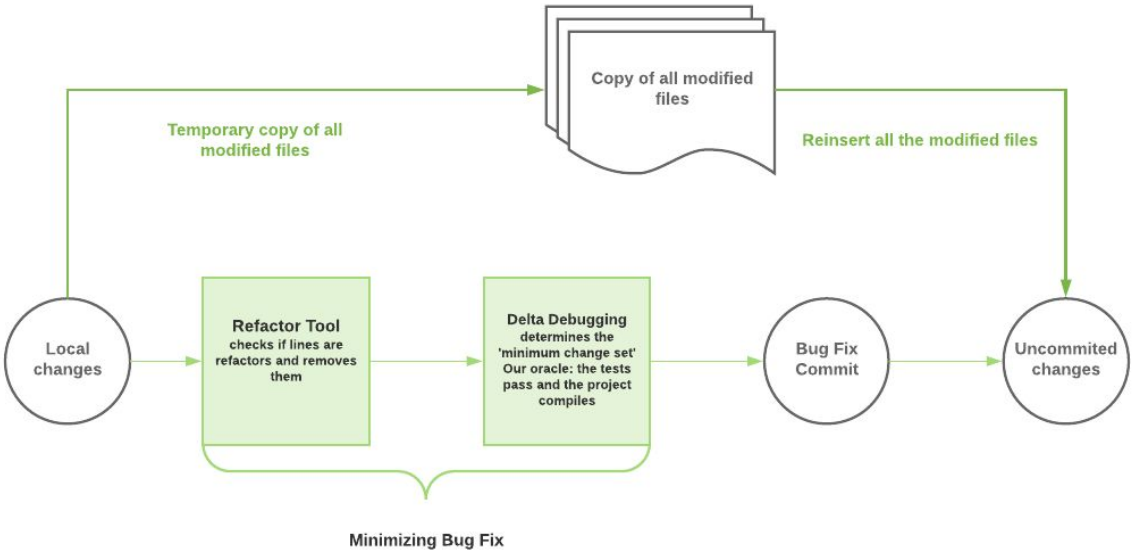


Figure 2: Architecture diagram

Challenges and Payoffs

Since our approach consists of removing lines and re-running tests to determine which lines of code, if any, can be removed without affecting functionality we see a scalability problem arising with large commits. With this type of checking, large commits would take exponentially longer to fully check and resolve. Another concern is that if the user introduces a method to fix a bug, the tool might incorrectly indicate that just the call to the method is the part of the commit that fixes the bug. One solution is to recursively look at each methods the user calls in that function and see if any changes have been made in each of those since the previous commit, this could be costly though. There are some simple cases that would be easier to handle, like for example if the user just created that method in that commit and

[5]Wilkerson, Daniel S, et al. "Minimizing Interesting Files with Delta." Delta: Minimizing Interesting Files with Delta, Dsw, 3 Aug. 2006, delta.stage.tigris.org/using_delta.html.

[6] https://github.com/benjsc/splitpatch

put it in the method we're concerned with. This is something that is possibly out of the scope of what our tool will be able to handle, but it is a case to be considered.

Designing a solution that runs efficiently on large commits will be the most difficult part of development. To reduce the problem space we plan to use an algorithm that does not try every single possible combination of line removal, such as delta debugging. It may rely on a heuristic to choose potential lines for removal. This approach may improve runtime but could introduce suboptimal minimizations because no heuristic is perfect and not every superfluous line will be removed. We will be able to manage our risk by removing any additional planned optimizations. An unexpected challenge we also faced was how long it would take to build all our files to retest for delta debugging, and as it turns out this is going to slow our tool down much more than we anticipated, causing even small changes to take several minutes. Because of this we were forced to instead do delta debugging based on chunks instead of individual lines, reducing our potential accuracy, but improving our runtime.

Successful implementation would result in high quality minimizations. Since running the tests with removed lines of code is a sure way to isolate the bug fix, if this process is time efficient, it will allow for narrowing down the scope of the bug fix further than code profiling and refactor detection could do alone.

Commit-min aims to automate the bug fix detection process in order to free up developer time before the commit, and after. If developers were previously spending time minimizing commits by hand, they will no longer need to do so. More dramatically, commit logs will be cleaner for future use. A developer will now spend next to no time in the future attempting to isolate code if they need to modify the bug fix, since it has automatically been split into its own commit message. This would result in hours of saved time, making a difference by freeing up developer time.

Cost and Testing Experiments

We have done multiple experiments for various code projects inside the defects4j repository, running our tool on their commits and testing across different sized commits. Some projects that we plan to evaluate in defects4j are Apache commons-lang, and Apache commons-math. To do this, we will use as many bug fix commits as we can find and run our tool against to test how accurate(percent of lines in expected that our tool found), precise(percent of lines that our tool found that were not in expected), and how fast our tool is.

We will specifically look at how many lines of code our tool missed that were a part of solving the bug. For example, if the real bug fix consisted of 4 lines, but our tool only found 3 of those lines that would be 75% recall - hopefully through our approach this number would be 100% for all sizes of commits. This would mean that our tool has been successful will be able to determine the exact number of lines that were required to solve the bug.

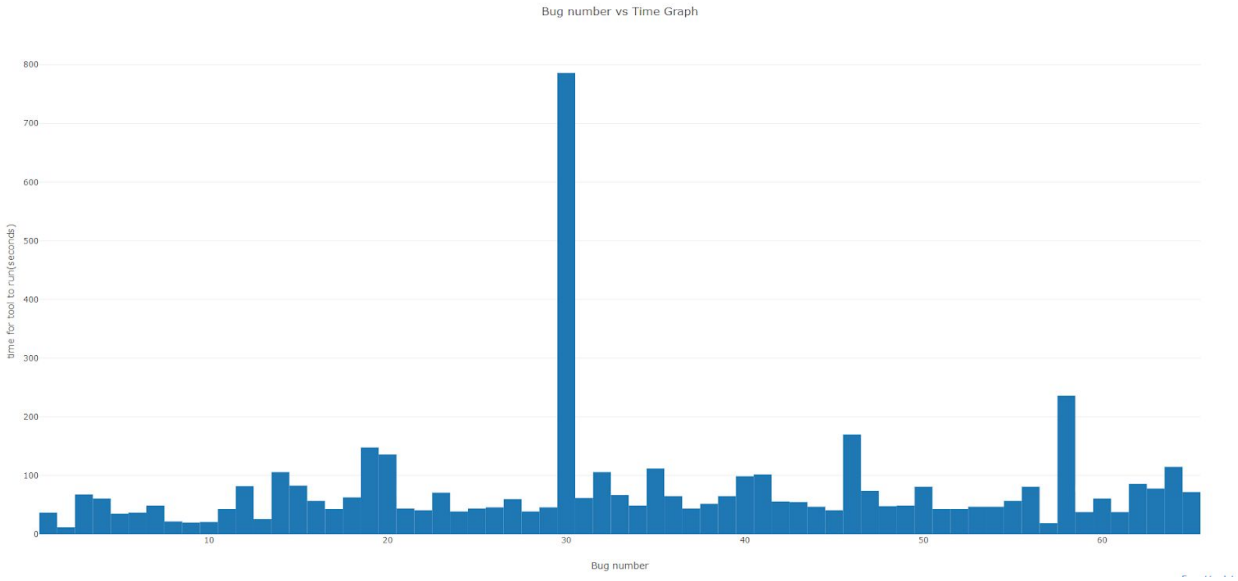
Additionally, we would test how specific our tool can get our prediction for the section of the code that solved the bug. For this, we would be testing the multiple of the number of lines it takes to solve the bug by the number of lines the tool predicts the bug is solved in. Say, for example, the bug was fixed in 4 lines, but our tool included those 4 lines plus a 5th line, we would consider that 20% over - this is something that we will probably work the most on trying to get low, seeing as it is the crux of our entire project essentially - the ideal/complete success would be to get this to 0%, but realistically we will want to balance the time it takes to run the tool versus getting as precise as possible in our estimation of what lines caused the bug fix.

The last test would show how the amount of time in seconds it takes us to figure out which lines of code were responsible for fixing the bug vs the size of the commit(in lines of code). This amount of time will probably be, depending on our strategy of narrowing the code that fixed the bug, dependent on the tests that the code base has. So, we will want to keep the repo we're testing on constant while changing

the number of lines of code in the commit. We would consider this a success if we see our tool take below 2 minutes for the majority of bug fixes.

Initial results

Time Graph for the Lang repo in defects4j

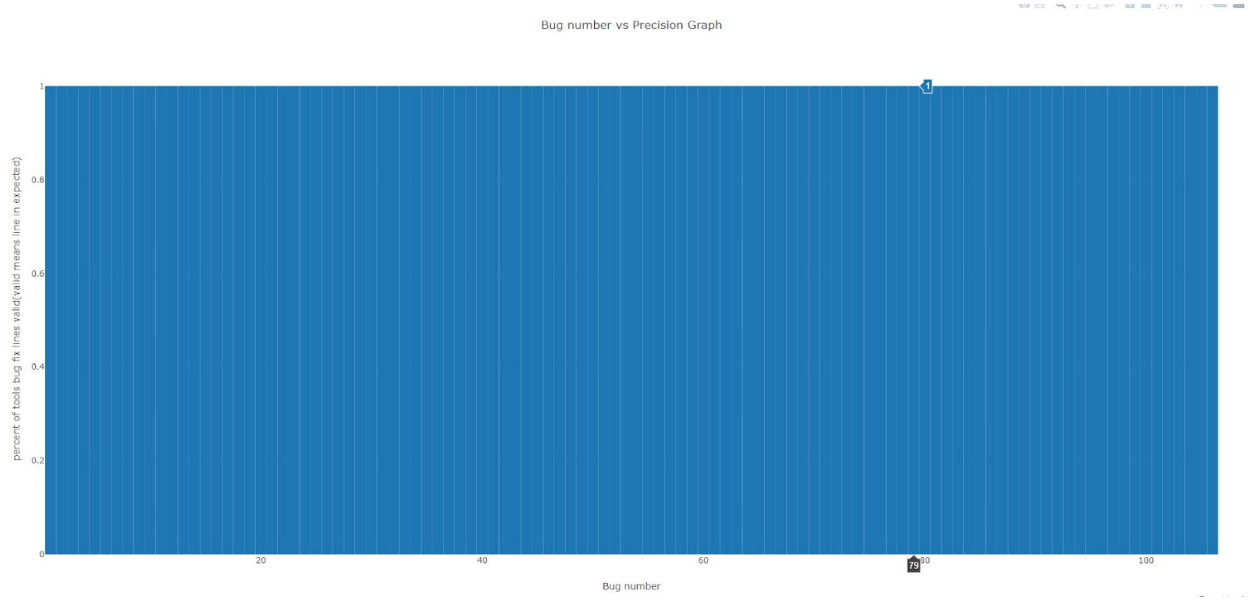


Just a section of the Accuracy Table for the Math repo in defects4j

Bug number vs accuracy

Bug number	percent of bug fix lines found
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1
11	1
12	1

Precision graph for the Math repo in defects4j



The rest of the graphs and tables can be found in our repo given below under the reports/results directory
Repo : https://github.com/Juliusc01/commit-min
Instructions to run our evaluation: README.md

With hunks we were able to run around 5 bug tests in around 10 minutes, which is a significant speedup. Our results from analyzing the Math repository in defects4j showed that our tool got 100% with respect to accuracy and precision. Which means our tool is unable to minimize the commits further, which wasn't necessarily expected as the defects4j repo contains small commits in the first place.

Midterm and Final Checks
Our midterm check entails finishing our early prototype, which we accomplished during week 5. The prototype uses delta to remove lines from the commit that are not affecting the test. At this point, our commits are not being reduced particularly far and we have yet to incorporate the refactor tool. The final check would involve running our tool with several examples as well as user testing to gain more information on how close our tool gets to minimum commit at a reasonable amount of time.

Implementation Details
Note we intend this to work just for java, using maven build tools since we needed a way to build, compile, and execute the tests while considering all the dependencies.

We've implemented our tool using git, and delta. We have been able to successfully modify delta to minimize several java files with respect to a given test. This modification included adding a name flag to pass in the name of the file to minimize, and changing the copying of files to include this name. It seems like it will work well with finding where bug fixes are located in a user's commit by minimizing the commit before and current commits minimized forms with respect to a test.

There is an optimization called Hierarchical Delta Debugging[7] which takes inputs in the form of a tree and recursively uses delta debugging on levels of the tree. There is an implementation of this called

[7] Misherghi, Ghassan, and Zhendong Su. "HDD: hierarchical delta debugging." *Proceedings of the 28th international conference on Software engineering*. ACM, 2006.

Picireny[8] which can take Java code as input, and parses it into a syntax tree for execution. Picireny could be used as a substitute for delta if we had time to continue development.

Hours spent on this report this week:
Jacob: 2 hours on report
Evan: 2 hours on report
Sarah: 2 hours on report
Conner: 4 hours on report
Julius: 2 hours on report

[8] https://github.com/renatahodovan/picireny