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

Commit-min: Minimize Bug Fix Commits

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.

Current Approach
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 copy all modified files and our tool will then run a three 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

---

[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.

commit procedure described in phase 3 below. Otherwise, the tool aborts and just commits the users original commit.

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.
The first phase of our tool will utilize change impact analysis[3]. This works by determining if a line of code in the commit our tool is considering could have impacted the test that fixed the bug. If a line of code could not have impacted the test, then commit-min can safely to remove it from the commit. This helps us narrow down the set of lines in the commit before it continues, making the tool run faster overall.

The second phase looks for lines that are refactored and removes them. Our tool will determine the simple cases of logically equivalent code, such as new comments, whitespace, variable name changes and new lines. A goal we can potentially work on in the is the case of method refactoring. This would be if a block of code is pulled out into a method that is then called in replacement of the code. Figure 1 illustrates this type of refactoring. Plagiarism detection tools may help in finding if a commit is a refactor. The technical report, *A comparison of plagiarism detection tools*[4], we found and tested a helpful tool JPlag, which detects similarities by source code tokenization.

```
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 2: Refactoring that would be caught and removed

The third phase will remove subsets of changed lines until it finds the largest subset of lines it can remove without causing the test to fail. An approach to determine the 'minimum change set' could be 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[5], but unfortunately we couldn't find the actual tool anywhere, only research papers, so we decided to use another tool called Delta[6]. This

---

[3] Ren, Xiaoxia, et al. "Chianti: a tool for change impact analysis of java programs." *ACM Sigplan Notices*. Vol. 39. No. 10. ACM, 2004.
[4] 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
[5] 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.
[6]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.

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.
By examining the minimal files from the standpoint of a certain test before and after the commit our tool will be able to determine what code was responsible for fixing a bug to get a certain test working. Once this is done our tool can commit the minimized change set, add the removed lines back, and commit again under the name of bug fix for a certain test. This dual commit is illustrated in Figure 2, which showcases how the the changes will be temporary duplicated so that they can be reapplied after the first commit.

Commit-min aims to be successful by combining three 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 1 informs the developer where to begin the search for the bug fix, phase 2 eliminates changes that are basic refactorings, and step 3 experimentally determines which lines are unnecessary. Combining three relevant approaches into one systematic tool should increase the speed of the delta debugging since less lines are under consideration and improve the overall accuracy of the minimized commit.

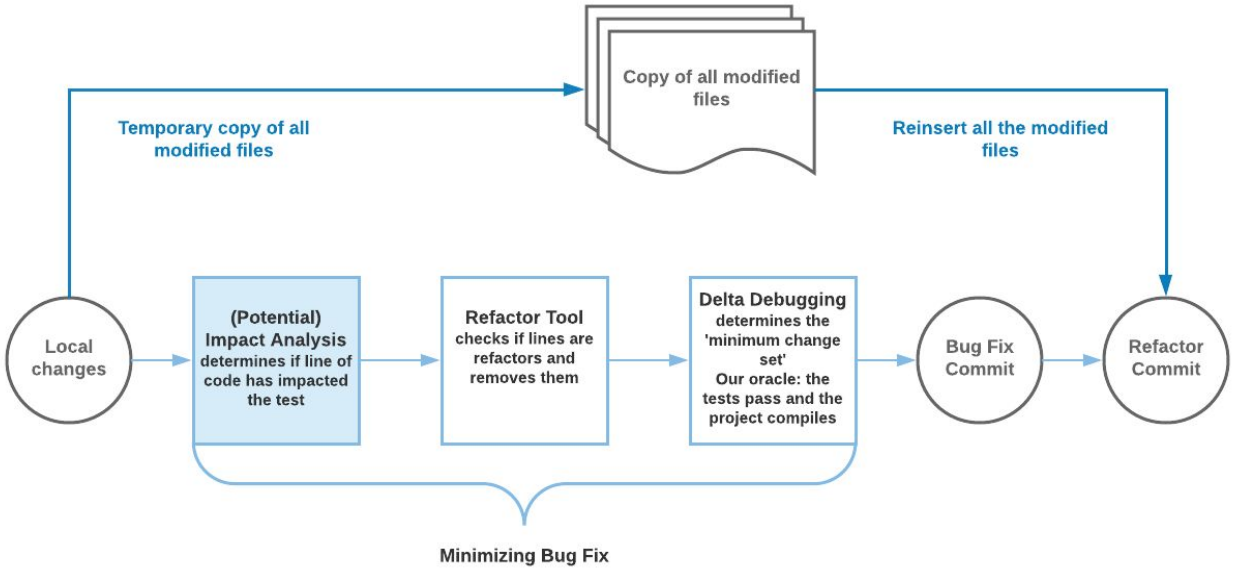


Figure 2: Architecture diagram

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. For n lines there are potentially 2^n combinations of line removal that would still allow for the tests to pass. For example, removing line number 3, 4 & 5, vs 3 & 4, vs 4 & 5, vs 3 & 5. With this type of checking, large commits would take exponentially longer to fully check and resolve. Another concern of our tool 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 that we came up with 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 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 we will try to consider.

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 difficult 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.

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.

Development Schedule

Week	Goals
3	-Design data structures and algorithms for our modules and how they will interact with each other. **(Completed - we have researched algorithms and tools)** -Setup the basic framework for our code such as necessary files, function headers, and comments. (**Completed)**
4	-Build and test git diff parser **(Completed)** -Implement test coverage tool to reduce lines to consider **(Decided to not put in our project)** -Implement basic refactoring reverts (whitespace, renames, comments) **(Completed)**
5	-Complete early prototype, will not have subset removal of lines yet **(Completed)** -Create test suite for test coverage and refactoring tools **(Completed)**
6	-Have others use our tool on their commits and get feedback **(Deferred)** -Create structured evaluation procedures **(Completed)** -Implement basic subset line reverting **(Completed)**
7	-Build more complex refactoring algorithms (method refactor, lines moved around) -Add heuristics to speed up subset line reverting and test
8	-Add functionality for user to control tool execution (run longer, choose whether to split commit) -Create high level how to use documentation
9	-Have others use our tool with new user controls and give feedback -Implement feasible feedback from users
10	-Evaluate tool over many examples to determine how effectively in minimizes commits -Build tool for deployment

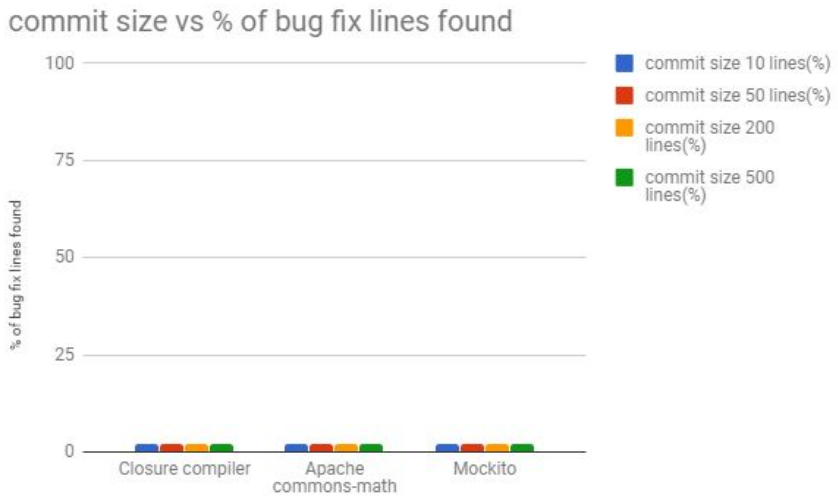
Cost and Testing Experiments
What we indicated in our timeline is how much this is going to cost in terms of effort and time from our group. We will also need the use of other tools like git pre-commit hooks, as well as possibly some sort of UI for our project. The time that this will take is going to hopefully be around 8 weeks to finish the project, then polishing up and testing for around 2 weeks. So 10 weeks in total.

We plan to do 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 Closure compiler, Apache commons-math, and Mockito. To do this, we will analyze 5 random bug fix commits of different sizes from these projects and run our tool against them in hopes to test how accurate(percent of lines that fixed bug found), precise(percent of lines that didn't fix bug added), and fast our tool is. We hope to do this by looking at the fixes, which we will consider minimal bug fixes and compare that to what our tool considers the minimum bug fix.

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% accuracy - hopefully through our approach this number would be 100% for all sizes of commits. 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 number of tests run constant while changing the number of lines of code in the commit. We would consider this a success if we see a linear speedup of our program as our commit size(in lines of code) increases, with a lower constant factor, so for a commit of around 50 lines we would shoot for around 5 minutes.

% of bug fix lines found	commit size 10 lines(%)	commit size 50 lines(%)	commit size 200 lines(%)	commit size 500 lines(%)
Closure compiler	2	2	2	2
Apache commons-math	2	2	2	2
Mockito	2	2	2	2



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 begun working on the implementation on two parts of our project, JGit, and delta. JGit is normally meant to be run from the command line, so we had a little trouble getting code that would work from the API, specifically the repository builder. It is not currently in a working or uploaded state, but JGit still looks to be to the tool to use in order to get what lines were changed from the previous commit in order to start minimizing. It will need more work to get to a usable state.

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 some details about minimizing code with respect to multiple tests that we need to figure out, this would require adding multiple tests into a script.

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 Picireny[8] which can take Java code as input, and parses it into a syntax tree for execution. We will try this in place of delta to see if it improves efficiency.

---

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