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/Related works
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 which is what we want to do.

We did not find an approach that works directly to minimize a bug fix commit. In practice it seems that developers are doing this minimization by hand. One example of this is in defects4j[3], a database of bug fixes and their commits. The paper on defects4j mentions that bug fixes were manually reviewed by the maintainers to remove any additional features or refactorings. Our goal is to automate this process.

Our Approach - High Level
Commit-min automates the bug minimization search process, freeing the developer from needing to manually decide which lines to include in a bug fix commit. Our tool determines that a bug has been fixed by running some test that was previously failing that is now succeeding. 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.

[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.
[3] "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs" René Just, Darioush Jalali, and Michael D. Ernst, ISSTA 2014

Our tool will be a script that is executed from a Git pre-commit hook that is 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.

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.

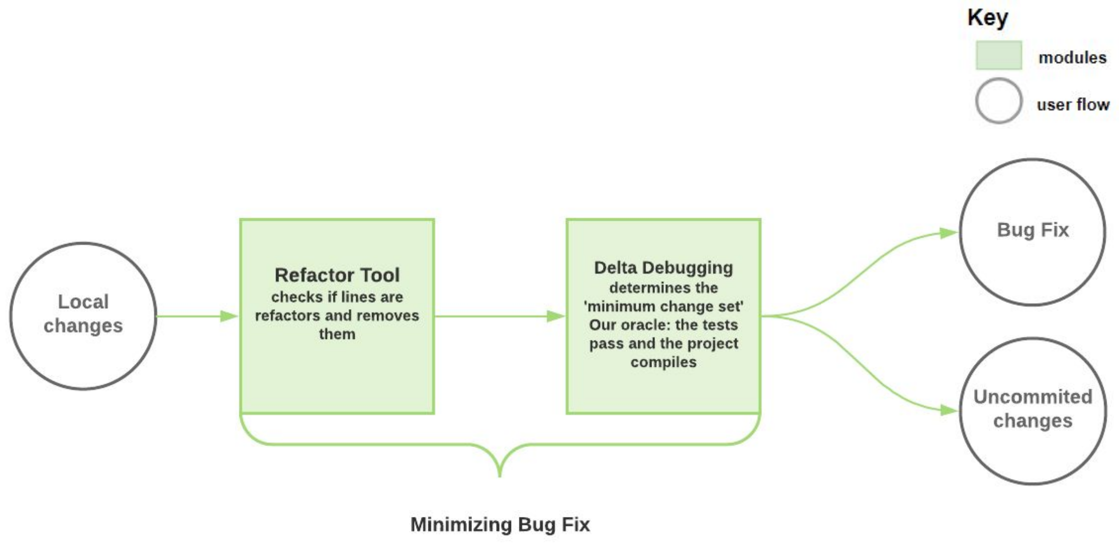
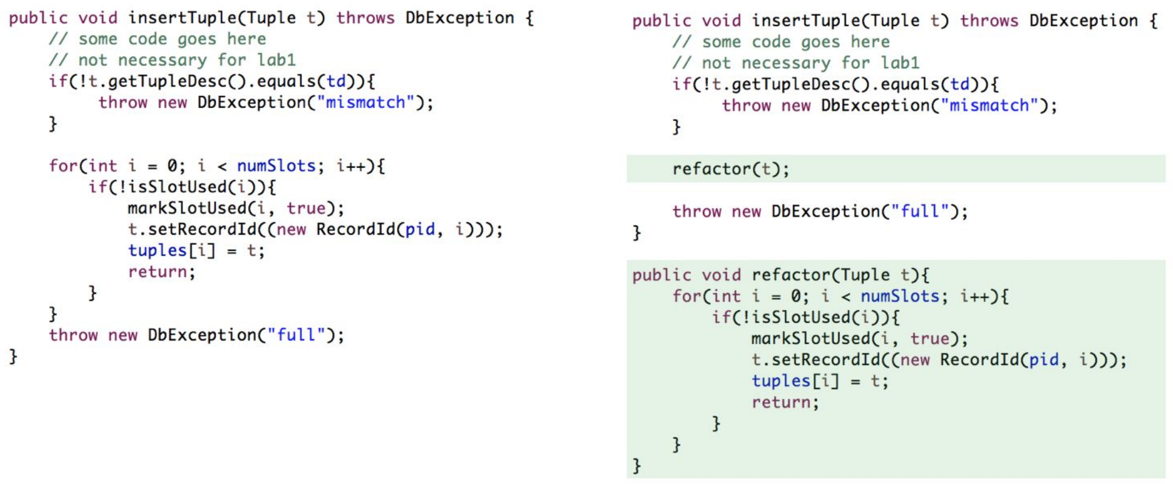

Key
modules
user flow
Local changes
Refactor Tool
checks if lines are refactors and removes them
Delta Debugging
determines the 'minimum change set' Our oracle: the tests pass and the project compiles
Bug Fix
Uncommited changes
Minimizing Bug Fix

Figure 1: Architecture diagram

Our Approach - Implementation

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 2 illustrates a type of refactoring. Through research, we found that many refactor tools were not open source or had unusable output. After discovering this, we went with the approach of using plagiarism detection tools to detect refactoring hoping these would be open to public use. In "*A comparison of plagiarism detection tools*"[4], we found and tested several tools to help us determine refactorings. We tried and tested a couple of open source tools that may be helpful such as SIM (supports C, Java, Modula-2, Lisp, Miranda) and Plaggie (supports Java), which detects similarities by source code tokenization. Unfortunately these did not work out well since SIM crashed when executing on Java, and Plaggie's output was unusable. We also explored the Moss plagiarism detector, but it did not run locally and required sending code to a remote server. We decided to use and 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.

[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



```java
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");
}
```

```java
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 2, 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 decided to use a tool called Delta[5]. This tool uses the 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. There is an optimization called Hierarchical Delta Debugging[6] which takes inputs in the form of a tree and recursively uses delta debugging on levels of the tree which should be considered for future work.

We modified Delta to work on the granularity of Git hunks instead of lines. 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 the tool's accuracy because a hunk might include some relevant and irrelevant changes, 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[7] 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 it has found the minimized commit, it will terminate and allow the user to commit the changes if they would like. The original files before the tool ran are stored in .bak files with the same name as the original.

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. Since delta is agnostic to language and test, future work could make this tool agnostic as well.

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

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, as delta debugging uses subsets of lines of code, causing an exponential runtime. This was fixed by having the tool run on hunks, which could decrease accuracy but greatly improved runtime. 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 those methods have themselves been changed since the previous commit, this would 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.

Designing a solution that runs efficiently on large commits was the most difficult part of development. To reduce the problem space we used to use an algorithm that does not try every single possible combination of line removal, by switching the delta debugging granularity to hunks. This approach improved runtime but introduced suboptimal minimizations because not every superfluous line is removed. 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. Changing our tool to hunks also helped with this problem, as the files would have to be rebuilt less often.

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 ran our experiments on the projects the defects4j[8] repository, specifically Apache commons-lang, and Apache commons-math. To do this, we ran our tool on 100 bug fix commits for both of those projects and tested for 3 things: accuracy (percent of lines in expected min bug fix commit that our tool also found), precision (percent of lines in our tools min bug fix commit that were also found in expected), and how long it takes our tool to terminate (seconds).

The combination of the accuracy and precision measurements determines whether our tool is able to correctly minimize a bug fix commit. Since when accuracy and precision are both 100%, that means all the lines in our bug fix commit are in the expected bug fix commit, and all the lines from the expected bug fix commit are in our bug fix commit, so our commit is the same as the expected which means we produce a correct minimum bug fix commit. The reason for looking at accuracy and precision separately is because they each reveal a different type of error produced from our tool. If the accuracy is below 100% that shows to us we are not finding all the lines that are essential for the bug fix, this is something we wanted to be at 100% at all times, since if it's not then our tool is not very useful, because it has not even found a correct bug fix, let alone a completely minimized one. If the precision is below 100% that shows to us we are not minimizing our bug fix commit as much as we can. We wanted to get this

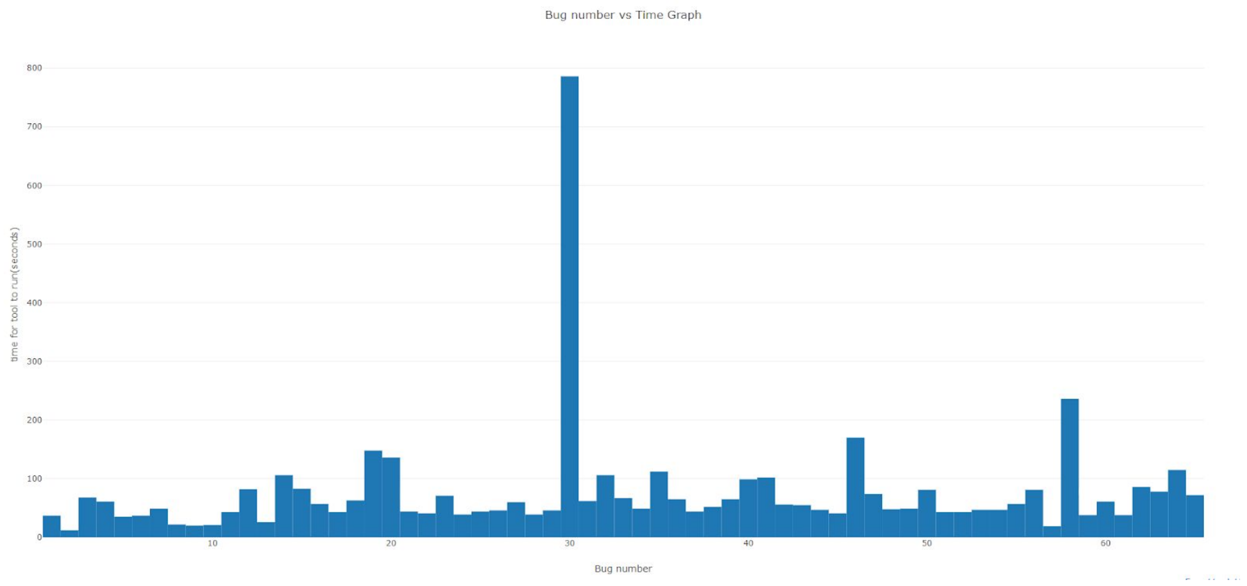
[8] https://github.com/rjust/defects4j

measurement as close to 100% as possible because it effectively determined how good of a job at minimizing we were doing, which is an important part of our tool.

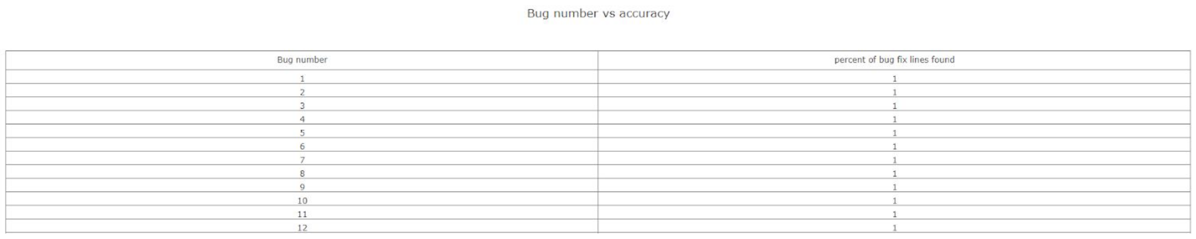
We also tested for the amount of time in seconds it takes for our tool to run. This amount of time is dependent on the size of the project we are working on as well as the time it takes to run the tests that the code base has. This is an important measurement because although it is important to get our tool to be correct, people will only want to use it if it is expedient enough to justify doing .

Initial results

Time Graph for the Lang repo in defects4j



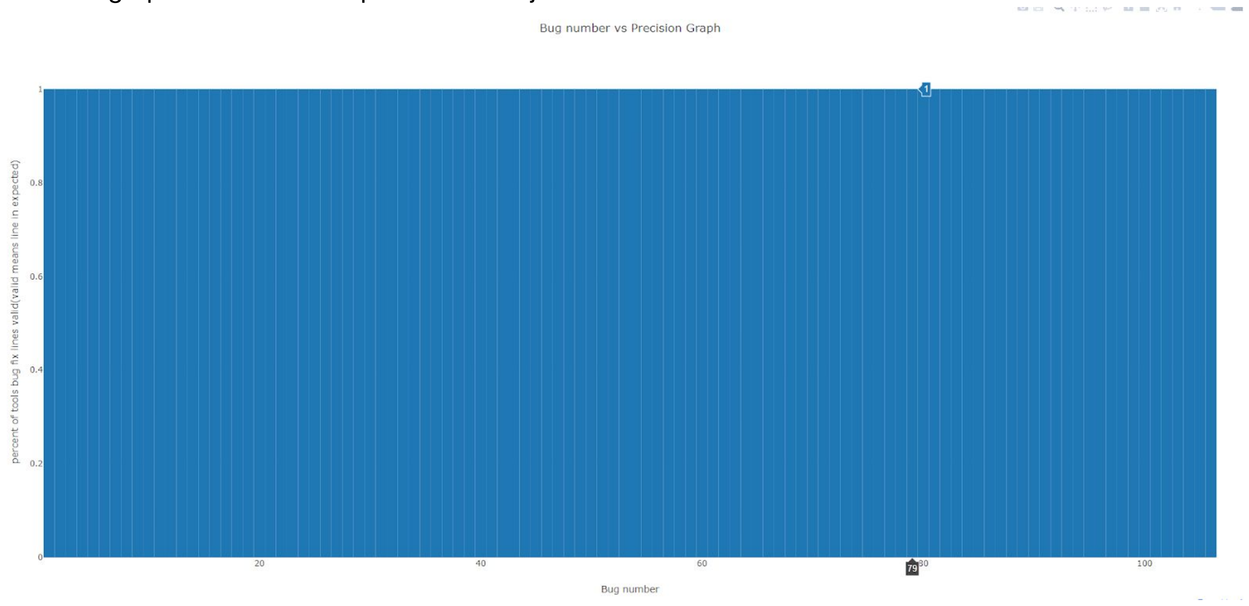
Just a section of the Accuracy Table for the Math repo in defects4j



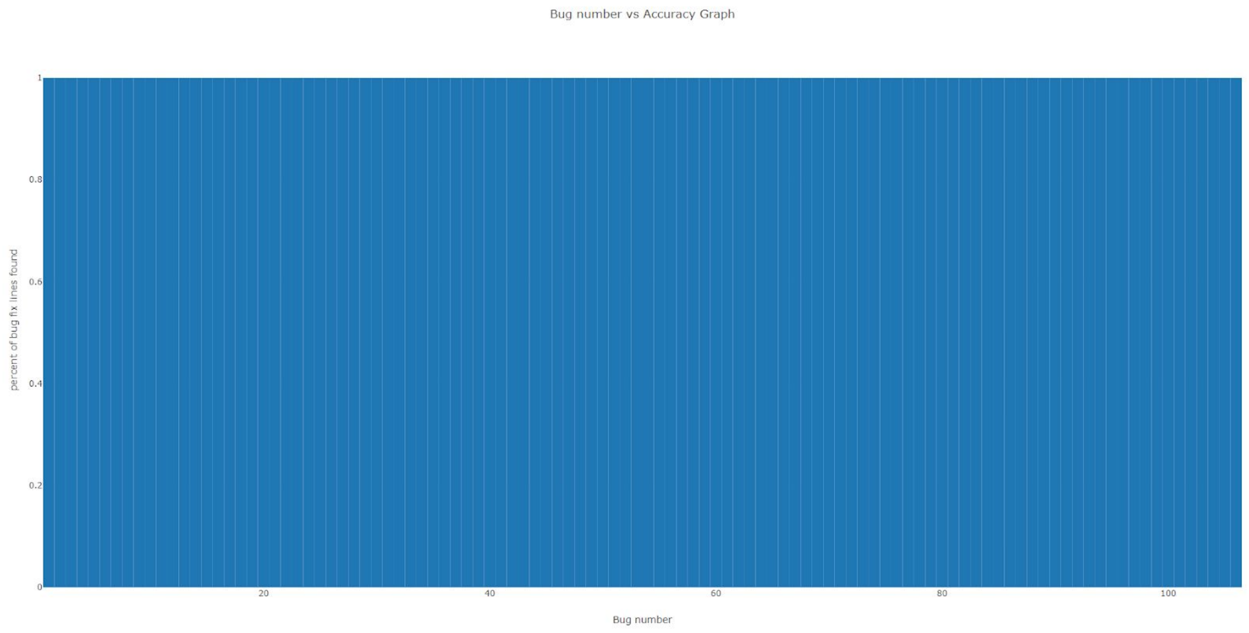
Bug number vs accuracy

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



Accuracy 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 unexpected as the defects4j repo contains small commits in the first place.

Hours spent on this report this week:
Jacob: 5
Evan:
Sarah:
Conner: 3
Julius: 3