Alva Wei (alvawei)
Jediah Conachan (jediah6)
Kenji Nicholson (kenjilee)
Steven Miller (stevenm62)
Sungmin Rhee (srhee4)

Smerge: Resolving Merge Conflicts Automatically

https://github.com/alvawei/smerge

Motivation

Software development often involves collaboration between multiple programmers, resulting in different versions of their project's code throughout development. Version control systems (VCSs) exist to manage these different and potentially conflicting versions. A VCS allows multiple developers to make edits to the code independently, and then merge those edits back into the master version of the project. Merge conflicts occur anytime two revisions contain competing changes—this could be as harmless as an extra line of whitespace or different variable name. Less harmless merge conflicts will require a resolution that changes how the code works. Currently, nearly all merge conflicts are manually resolved by the programmer. Since merge conflicts are frequent and often take non-trivial amounts of time to resolve, automating merge conflict resolution will reduce the time and money spent on large projects.

The typical VCS, such as Git, uses line-based analysis to detect merge conflicts. Each line of code is treated as an atomic unit, and a change anywhere in the line is seen as a change to the entire line. This approach often detects "false" conflicts that should not require a manual fix. Changes to the same line of code do not necessarily conflict if the changes are made in two separate regions within the line. A real conflict occurs only when the changes overlap in the same region. Current VCSs are unable to make this distinction because they handle a single line as one unit. Automatically resolving such "false" conflicts is just one way to reduce the work needed by programmers.

Current Approaches

Developers already have a few options to mitigate this problem. Git has an "ignore whitespace" merging option that automatically resolves conflicts caused by unequal whitespace. Microsoft's Team Foundation Server can automatically resolve conflicts with line changes that do not directly conflict or line changes that exist in only one of the branches. Other tools, like Bitbucket [2], resolve all conflicts by prioritizing changes by branches given a higher rank. These tools fall short in that they either resolve only a few types of merge conflicts, or that they require a large amount of user input.

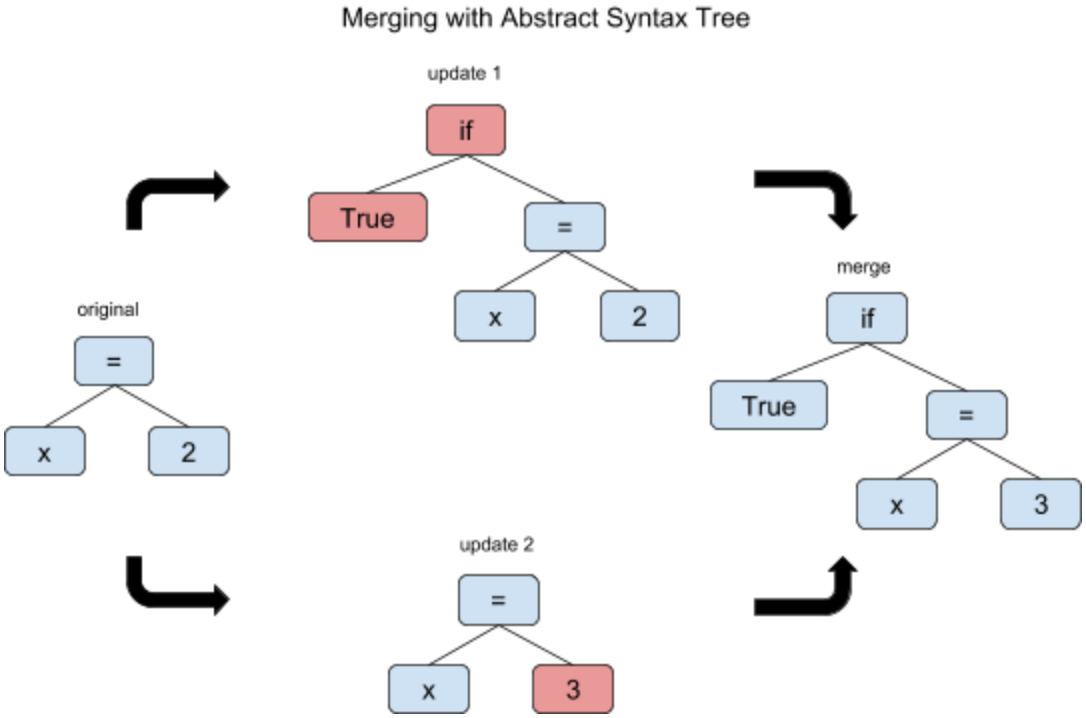
Another approach by previous CSE 403 students, Conflerge, handles merging by either parsing code into abstract syntax trees (ASTs) and merging the trees of the conflicting commits or tokenizing the input and diffing the tokens using the Wagner-Fischer algorithm [3]. However,

with the lack of user input, the tool tended to produce unintended results like defective merges and unnecessary modifications beyond the merge itself. It also faced issues where certain merges could be performed, but the syntax of the two commits resulted in an incompatible difference which ultimately led the tool to abort the merge. Additionally, Conflerge is only able to merge two branches at once—making a scenario where 2+ conflicting commits unfeasible.

Our Approach

Our approach will attempt to combine previous approaches: non-conflicts will be categorized and handled accordingly, and all other conflicts will be auto-resolved by some algorithm (without necessarily relying on branch ordering like Bitbucket). One algorithm we will consider while working on this project is the Wagner-Fischer algorithm, which computes the edit distance between two strings.

To resolve a merge conflict, we must first determine whether the changes truly conflict. It is a major issue if code is thrown away during the merge process that we intended to keep. This creates a headache potentially much more troublesome than just manually handling the merge conflict with the current git merge tools. One attempt involves creating abstract syntax trees (AST) from the code changes. Another approach could be to tokenize inputs. Examples of tokens are Java's keywords, variable names, fields, constants, etc. If the syntax trees are the same, or tokenizing the input does not produce any issues, then the conflicts are superficial and we only need to decide which new variable name or extra whitespace to keep. Shown below is an example of merging ASTs to solve a merge conflict.



Otherwise, the two branches have conflicts that are not easily resolvable and we must determine which changes to keep or how to combine the changes. This could involve some form of ordering branches (like Bitbucket), prioritizing certain types of changes, or looking at historical data to see how past conflicts have been resolved.

Similar to Conflerge, we would like to implement our tool so that it sits right in the middle of the git merge process. We could parse the git temporary files, $BASE (the common ancestor), $LOCAL (our changes), and $REMOTE (their changes) using something like Javaparser. Once the files are parsed, we could either use the approach of tokenizing or creating AST's, editing and merging them, then unparsing and handing back the results to complete the merge process.

Currently, Conflerge either succeeds silently or fails completely. A tree-based GUI might be helpful in performing/resolving merge conflicts. Since the AST approach is designed to find the differences between two subtrees, it might be helpful to visualize the process so that we can incorporate and streamline user input during the process. Although the perfect solution would handle all merge conflicts automatically and produce exactly what the user wants, this is probably a pipe dream with the variability that different code bases provide. The benefit of creating user input here with ASTs might be a more intuitive way to solve the problem, especially for new developers who do not quite understand git or how to resolve these conflicts manually. One thing that was discovered in the process of creating Conflerge is that tokenizing inputs in general was not as successful, so we may avoid creating a GUI for it.

Assessment and Experiments

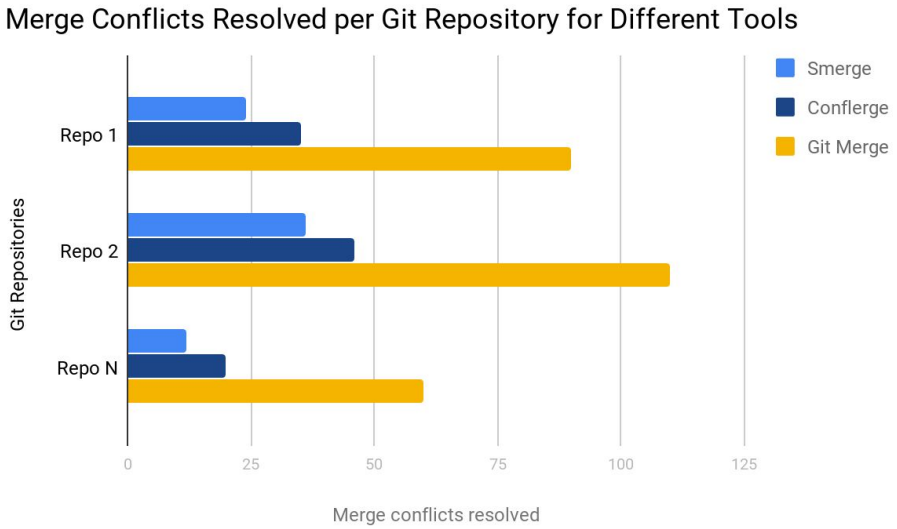
To assess the success of Smerge, we will conduct the following experiment:

Hypothesis: Smerge will reduce the number of merge conflicts experienced by the programmer

Procedure: To assess the viability of Smerge, we will follow the following steps:

1. Gather many GitHub repositories and their respective historical data
2. From the historical data, look for merge commits that have two or more parents
3. Use git's standard merge tools on the commits and record how many conflicts arise
4. Use other tools on the commits and record how many conflicts arise
5. Use Smerge's merging algorithm and record how many conflicts arise
6. Compare and contrast the human resolution to the conflict with Smerge's merge resolution

From there, we would record the data into a bar graph, similar to below. Note that the data shown does not represent any real values, it is just there for example purposes:



Conclusion: Based on our data collection, if Smerge results in effectively less conflicts than its predecessors, then we will have succeeded in our goal of creating a tool that auto-resolves more conflicts than existing tools.

Risks and Challenges

The primary function of this tool is to automatically resolve merge conflicts. The largest obstacle to achieving this goal is the quality of the merges produced. To minimize the work needed by the user to manually resolve a merge conflict, a satisfactory merge must be presented to the user. If an unsatisfactory merge is produced, the user must revert the merge and manually resolve the conflict, thus increasing the total amount of work needed by the user.

This leads to the difficult question of what constitutes a satisfactory or successful merge. In other words, how do we resolve a merge conflict in an acceptable way to the user? Our best answer is that it depends on the conflict; simple whitespace conflicts can easily be resolved and have no effect on the actual codebase, whereas some more complicated conflicts may require user input. A greater challenge may lie in minimizing this user input while still producing successful merges.

References

[1] Asenov, D., Guenot, B., Müller, P., & Otth, M. (2017, April 29). *Precise Version Control of Trees with Line-based Version Control Systems*[Scholarly project]. Retrieved April 5, 2018, from https://pdfs.semanticscholar.org/cc44/ca01cf8e82437f08374c03d2ef7a63651ec1.pdf
[2] The Git solution for professional teams. (n.d.). Retrieved April 05, 2018, from https://bitbucket.org/
[3] Navarro, G. (2001, March). *A Guided Tour to Approximate String Matching*[Scholarly project]. Retrieved April 5, 2018, from http://repositorio.uchile.cl/bitstream/handle/2250/126168/Navarro_Gonzalo_Guided_tour.pdf
[4] Hanawalt, G., Harrison, J., & Saksena, I. (2017, March 10). *Conflerge: Automatically Resolving Merge Conflicts*[Scholarly project]. Retrieved April 5, 2018, from https://github.com/ishansaksena/Conflerge

Week-By-Week Schedule

- Weeks 1-2
    - Choose project, complete project proposal, and begin planning implementation
- Week 3
    - Complete Architecture and Implementation plan and begin implementation
- Weeks 4 - 5:
    - Basic implementation (Parsing code into ASTs, functionality to merge trees, etc.)
- Weeks 5 - 6:
    - Additional implementation (add more features such as a GUI or additional conflict handling, introduce something new and helpful in handling conflicts)
- Week 7:
    - Begin writing tests and testing our tool, gather data for evaluation by pulling merge history from several popular Git repositories
- Weeks 8 - 10:
    - Finalize project, continue testing, start and finish gathering results for evaluation, begin drafting final report
- Week 11:
    - Complete final report

We spent 20 hours on this assignment.