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

Smerge: Smarter Merge Conflict Resolutions

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.

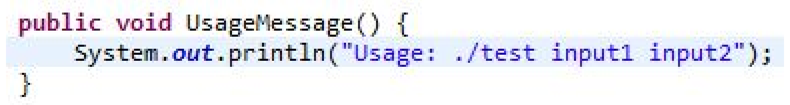
The typical VCS, such as Git, uses line-based analysis to detect merge conflicts. Each line of code is treated as an atomic unit, and a change anywhere in the line is seen as a change to the entire line. This approach often detects "false" conflicts that should not require a manual fix. Changes to the 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. Our goal with this project is to make resolving git merges easier and less time consuming. To do this, we will both automatically handle as many trivial merge conflicts as possible (like whitespace) and incorporate a GUI that accepts user input so that we can handle the less trivial conflicts. By finding this middleground, we believe resolving merge conflicts will become easier with our tool.

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.

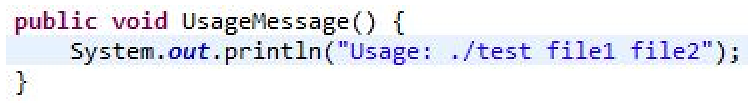
Another approach by previous CSE 403 students, Conflerge [4], handles merging by either parsing code into abstract syntax trees (ASTs) and merging the trees of the conflicting commits or tokenizing the input and diffing the tokens using the Wagner-Fischer algorithm [3]. However, with the lack of user 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. For example, consider the following two code bases that the program wants to merge:

User’s commit:



```
public void UsageMessage() {
    System.out.println("Usage: ./test input1 input2");
}
```

Other user’s commit:



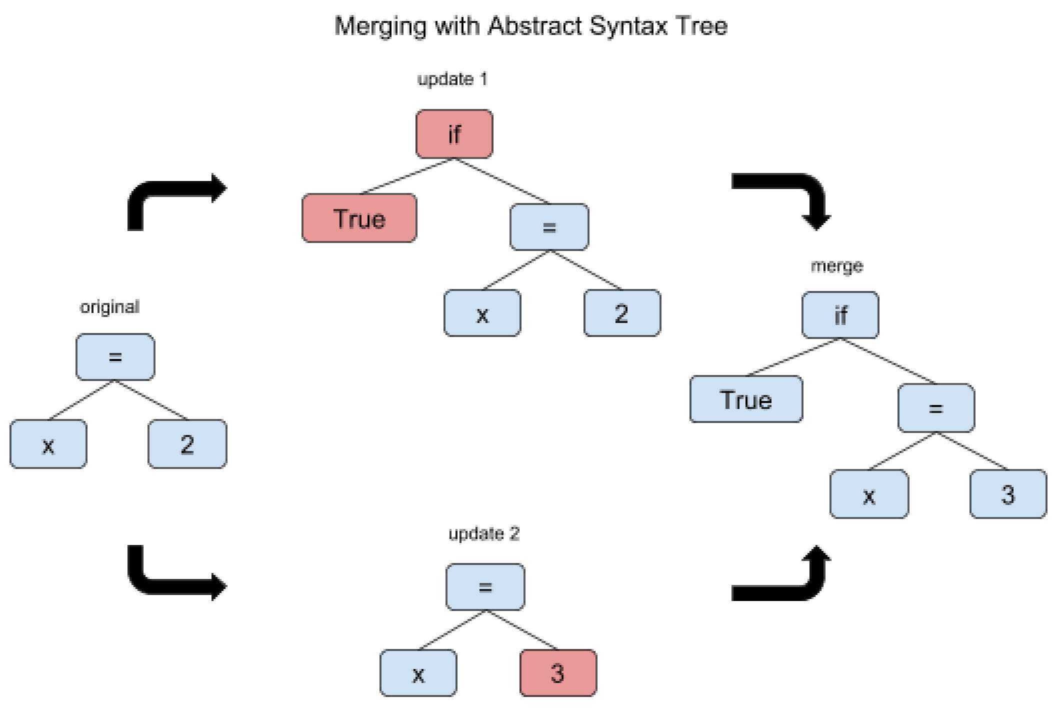
```
public void UsageMessage() {
    System.out.println("Usage: ./test file1 file2");
}
```

In this case, since the two lines highlighted in blue represent completely different outputs, Conflerge’s merge tool will not know which commit the user intends to keep. This results in the tool aborting the merge because otherwise it will produce a defective merge. Our tool supplements this by allowing the user to fix conflicts like these while also automating any trivial conflicts.

Our Approach

Our approach will attempt to combine previous approaches: non-conflicts will be categorized and handled accordingly, simple conflicts will be be auto-resolved by some algorithm (without necessarily relying on branch ordering like Bitbucket [2]), and complicated merges will be handled via user input in a GUI.

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. Our attempt involves creating abstract syntax trees (AST) from the code changes. If the syntax trees are the same, then the conflicts are superficial and we only need to decide which new variable name or extra whitespace to keep. If the changes are in different subtrees, then we can resolve the conflict by performing a tree merge on the ASTs. Shown below is an example of merging ASTs to solve a merge conflict.



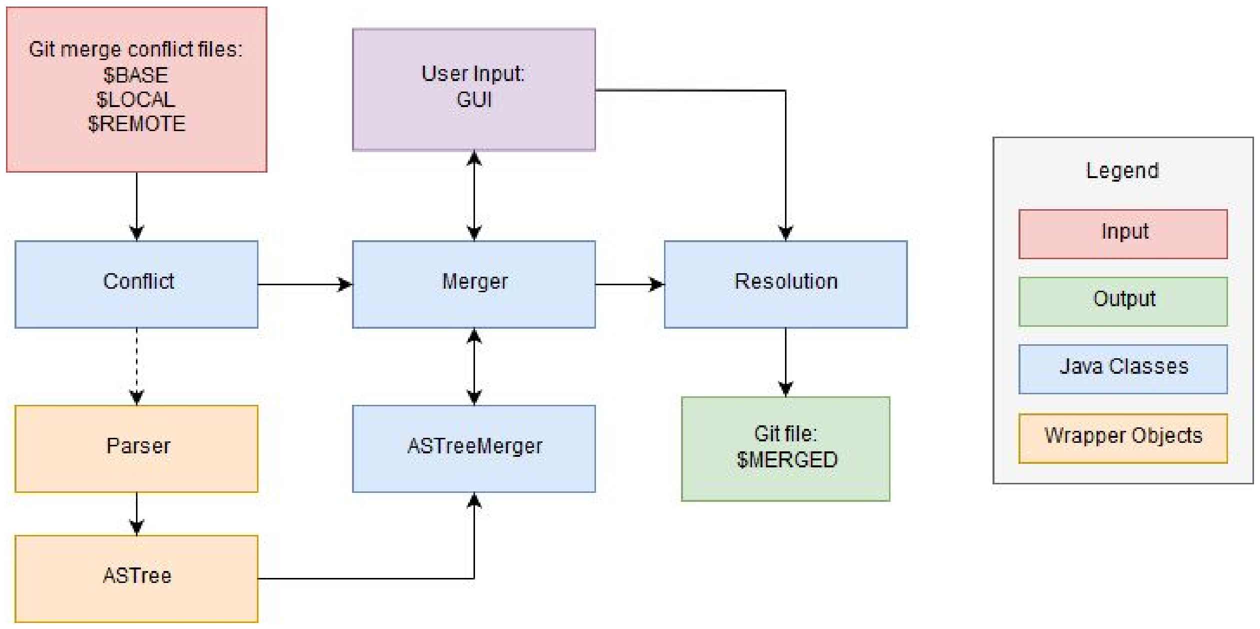
Otherwise, if the changes are in the same subtree, then the conflict involves overlapping structural differences and manual user input is required to resolve the conflict.

Conflerge is a tool that sits right in the middle of the git merge process. Similarly, we will utilize git merge to generate input files. Git merge produces three temporary files $BASE (the common ancestor), $LOCAL (our changes), and $REMOTE (their changes). Then, these files can be passed to Javaparser to create AST's so that we can edit and merge them. After that, we can unparse and hand back the results, completing the merge process.

Currently, Conflerge [4] either succeeds silently or fails completely. To aid this, we propose creating a GUI that helps visualize the merging 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. Our goal with this approach is to provide a middle ground between the two types of existing approaches previously mentioned.

Project Architecture

As a black box, Smerge takes in a merge conflict (a set of temporary git files) as an input, and ideally outputs a successful merge (another file) as an output. Additional user input will be acquired through a GUI.



The Conflict class will act as a container for the input files. The Merger class acts as the main program, taking user input from the GUI to produce a merge conflict resolution. A Resolution object stores potential resolutions (as a file), and will write the final merged file to disk.

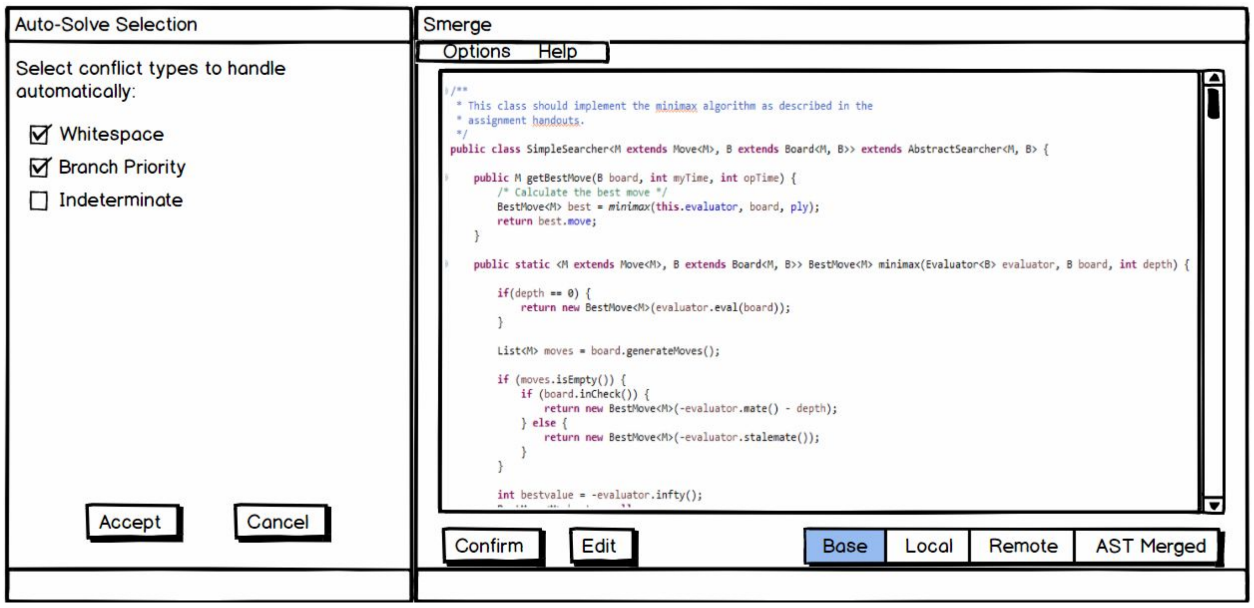
The Parser class acts as a wrapper around JavaParser, which will parse the input files into abstract syntax trees. An ASTree object is a wrapper for the trees JavaParser produces. These wrapper classes allow room for our tool to support other languages in the future. The ASTreeMerger will attempt to merge the parsed trees into a single tree as a potential resolution (after unparsing the tree) to the Merger.

Currently, like Conflerge, our tool can only be run as a git mergetool. Our architecture differs in the fact that our tool will use a GUI, using user input to shape the final resolution (even if an AST merge is impossible). Conflerge, on the other hand, tries to automatically produce a resolution with no user input.

Implementation Plan
The first step of our implementation is to be able to rewrite the backend of Conflerge as our own; this will allow better comprehension of our tool as well as room for additional features. If we can generalize abstract syntax trees for all languages, then supporting multiple languages only requires the ability to parse such languages into these general ASTs. This may be too idealistic for our time frame however, as Conflerge is heavily dependent on JavaParser's ASTs. Once we can merge conflicts through the use of abstract syntax trees, we can increase the functionality of our tool, primarily through the implementation of a GUI.

Graphical User Interface

GUI Mockup



```
/**
 * This class should implement the minimax algorithm as described in the
 * assignment handouts.
 */
public class SimpleSearcher<M extends Move<M>, B extends Board<M, B>> extends AbstractSearcher<M, B> {

    public M getBestMove(B board, int myTime, int opTime) {
        /* Calculate the best move */
        BestMove<M> best = minimax(this.evaluator, board, ply);
        return best.move;
    }

    public static <M extends Move<M>, B extends Board<M, B>> BestMove<M> minimax(Evaluator<B> evaluator, B board, int depth) {

        if(depth == 0) {
            return new BestMove<M>(evaluator.eval(board));
        }

        List<M> moves = board.generateMoves();

        if (moves.isEmpty()) {
            if (board.inCheck()) {
                return new BestMove<M>(-evaluator.mate() - depth);
            } else {
                return new BestMove<M>(-evaluator.stalemate());
            }
        }

        int bestvalue = -evaluator.infty();
```

The goal with our GUI is to provide a platform that makes communication with the user easier. This is done in two ways: the first is by providing a checkbox list with types of conflicts the user would like our tool to handle automatically (shown in Auto-Solve Selection window). For example, if the user checks ‘Whitespace’ then the tool will proceed to automatically handle conflicts caused by whitespace. This list is not complete yet and we may add more types as development progresses. The second is by allowing the user to “confirm” changes before we make them. To do this, whenever a conflict that doesn’t fall under the types selected in auto-solve is proposed, the user is prompted to confirm the changes through the ‘confirm’ button. This helps prevent the tool from making any changes that disturb the behavior of the intended code. Note that while the images shown are just placeholders for now, we do plan on highlighting the conflicts. Additionally, when a non-trivial conflict pops up, users will be given the opportunity to edit the code by clicking the ‘edit’ button. Also, there will be Base, Local, Remote, and AST Merged buttons that allow the user to see the current status of each of the files. The main window will also enable users to access a quick guide to using the tool under ‘Help’. It will also allow users to access the auto-solve menu through the ‘Options’ tab.

Existing Systems

As for existing systems, Smerge does not necessarily extend a larger existing system. However, we do plan on using several ideas presented in its predecessor, Conflerge [4]. This includes things like AST merging, git scraping for evaluation, and other general merging algorithms.

Tools

Javaparser is capable of parsing Java source code files into abstract syntax trees. Conflerge was centered around this tool, merging the ASTs generated by JavaParser to resolve merge conflicts. We currently plan to do the same, though this may bring trouble in generalizing AST merging for multiple languages.

Assessment and Experiments

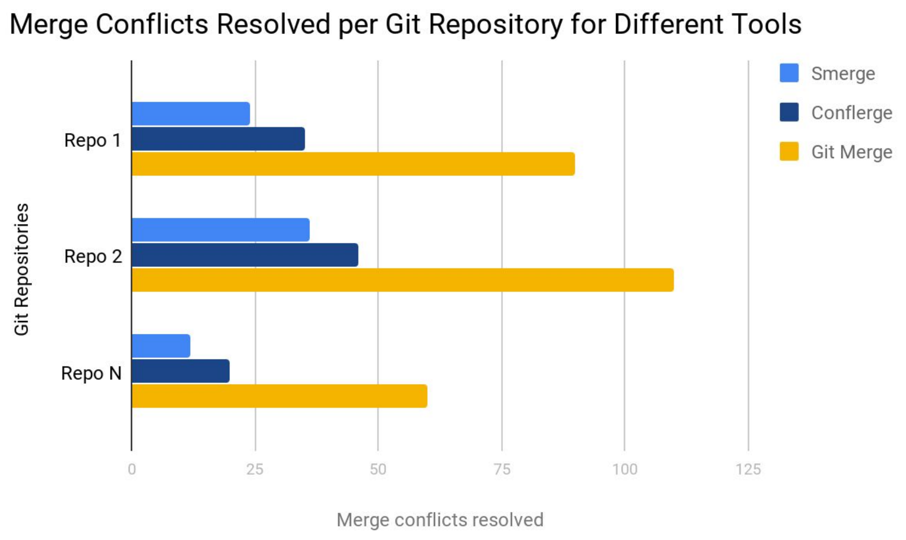
To assess the success of Smerge, we will conduct the following experiment:

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:



Currently, we plan on applying our tool to the following git repositories: androidannotations, elasticsearch, fastjson, glide, javaparser, libgdx, MPAndroidChart, netty, retrofit, and RxJava. We may add more if time permits.

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.

GUI Assessment: To assess the success of our GUI, we plan on distributing user surveys so that we can get user feedback on how usable it is. To do this, we would prompt the user to test our tool on a set of conflicting input files that we provide. Then, we would ask what worked well and what worked poorly when they tried to merge the files, followed by a request on what could be improved.

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.

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 like the GUI. If time permits, add additional conflict handling through algorithms like branch ordering, 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 by proofreading specification/documentation, cleaning up code, etc. Then continue testing, start and finish gathering results for evaluation, begin drafting final report
- Week 11:
  - Complete final report

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

We spent an additional 10 hours on this assignment.