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

1 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. The most common VCS used today is git [17]. 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. Merging is the process of taking two versions of a file, local (your version) and remote (the version everyone contributes to), and combining them into a single file. In order for the main goal of contributing and collaborating in the same project to be a success, VCSs such as git are conservative in that when a merge fails, they notify the user that the merge was unsuccessful instead of attempting to automatically solve it. This leads to developers having to spend time to manually edit the files to account for the changes made in the two parent files. In the best case scenario, one programmer makes a change to a single file that no one else is working on, and wants to merge their version of the file/project with the remote branch. Git can merge the two versions of the code in this case, and everything works as expected.

Git is not always able to automatically merge two files, however. We define a merge conflict as what happens when two contributors make changes to the same piece of code and one contributor merges their code with the remote version before the other. When the second person merges their code which interferes with the code that the first person wrote, there is a merge conflict. Consider the following simple example in python:

The base version that 2 users edit:

```
1  # Common ancestor (Base):
2  x = 0
3  def doSomething(modify)
4    self.x = 2;
5
```

Contributor 1's commit, note their change in x assignment on line 4

```
1  # Yours (Local):
2  x = 0
3  def doSomething(modify)
4    self.x = 3
5
```

Contributor 2's commit, note their addition of an if statement spanning from lines 4-5

```
# Theirs (Remote)
x = 0
def doSomething(modify)
  if modify
      self.x = 2;

```

In the case of an example like this, git is unable to automatically merge the two commits when both users push their changes. These competing changes could be as harmless as an extra line of whitespace or different variable name, but both will result in a merge conflict that requires manual resolution. We define the notion of “harmful merge conflict” as a conflict which affects the way the code works, such as a merge conflict where there are updates on separate branches on the same exact line. Harmful merge conflicts will require a manual resolution. Here is an example of a harmful merge conflict:

> Person 1 and person 2 pull from the same version of the remote project and begin writing tests for some similar source code. Say they are making changes in the same file, and person 1 removes a test from the test suite and adds a new one to replace it. Person 2 edits the same test that person 1 deleted, and pushes their code to the repository. When person 1 gets finished making changes and pushes their code, there is a conflict. This example is what we classify as harmful, and is something we would not be comfortable with tackling using our tool. This conflict would require manual input from the user to be resolved.

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. We define false conflicts as ones that are simplistic enough that resolving them should be taken care of automatically. Some examples of false conflicts include things like variable name changes, extra added white space, or putting an if statement around a variable assignment. Additionally, changes to the same line of code do not necessarily conflict if the changes are made in two separate regions within the line. For example, consider a method header with multiple parameters. If one person changes the name of a parameter and another person adds a parameter, those changes occur in the same line of code but shouldn’t necessarily cause a conflict. What we determine to be a “true” 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. Whether the conflict be either “true” or “false”, the user is required to manually edit the file that has both changes in it, until the final version is correct. Automatically resolving such “false” conflicts is just one way to reduce the work needed by programmers.

It is worth noting that Conflerge[4] is an existing tool that solves many of the above problems, barring merge conflicts that it believes to require manual resolution (edits to the same println statement, for example), but we believe it can be improved upon for the following reasons:

1. Conflerge's use of JavaParser [15] imposes limitations on Conflerge. The result of a merge is all formatted according to the default Javaparser formatting. Any custom whitespace, etc is NOT preserved. This is relevant to developers who use custom whitespace in their code for code clarity and readability purposes. For example, custom whitespace is important in industry in that it allows developers to break code into blocks so they may better understand each separate action made in a method, which in turn allows for better code readability. While the conflict may be solved, if a developer cares about custom whitespace, they must manually reinsert it.
2. Conflerge acts as a wrapper around JavaParser, meaning other languages are not supported.
3. When there is a failure to merge (meaning that the tool couldn't resolve the merge conflict automatically), no information is given to the user as to why it failed. Manually merging with git is required at this point. In the case of a failure, we plan to provide the user with a location in the file that caused the merge conflict instead of exiting silently if time permits. This is a pretty standard practice (can be found in Git), although Conflerge does not provide this feature.

Our goal is to reduce the number of conflicts presented to the users, similar to Conflerge, but also provide a merging module that can be used with several different language parsers (i.e. python, java, etc.). To do this, we will automatically handle as many trivial merge conflicts as possible and implement a generic abstract syntax tree that can work with different parsed languages (both of which are detailed later in the report.)

Throughout the report, we will use Conflerge[4]'s evaluation technique to evaluate our tool.

2 Current Approaches

Developers already have a few options to facilitate the merge conflict resolution process. Git has options to ignore whitespace changes[1] while attempting the initial merge, and there are a multitude of merge conflict resolution tools for three-way merging, visual representation of merges, viewing merge history, etc. Some of the more popular merge tools:

- kdiff3[2]
- P4Merge[3]
- diffmerge[4]
- Meld[5]

Most of these existing merge tools are comprehensive, all-in-one merge solutions. Not all of these tools are merge automators; most of them provide an interface for the user to try different methods of merge automation. For example, kdiff3 provides a GUI, a code editor, color-coded difference visualization, and an automatic merge facility with several options for automatic conflict resolution[6], to name just a few of their features. Smerge, on the other hand, provides one specific way to resolve a conflict, AST merging. In the future, we could see Smerge being integrated into an existing merge tool like kdiff3 as an additional option for automatic conflict resolution.

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, as mentioned above, we believe Conflerge is not an ideal tool, and our solution will attempt to improve on their work.

3 Our Approach

The two issues of Conflerge that our tool aims to address are language flexibility and formatting preservation. Both of these issues can be solved by a generic AST. A generic AST is not language dependent, meaning that any language should be able to be parsed into the AST (which should be able to be unparsed back into source code). That means, ideally, that our tool requires only a language specific parser for each supported language, as our AST diffing and merging algorithms work on generic ASTs.

Another benefit of implementing our own generic AST is that it allows us to retain more source code information. This allows us to unparse a generic AST as well as retain original source code formatting. This does make our generic AST less abstract, but it is necessary for being able to unparse a merged tree.

This kind of generic AST requires language parsers specialized for our generic AST. Most parsers parse source code into ASTs that are too abstract (such as JavaParser which ignores whitespace, and Eclipse's JDT parser which ignores comments). There may exist some parsers that produce an AST that can be converted into our generic AST, but that would require a lot of time.

Thus far, our tool supports only Python 3. We plan to support at least one other (still undecided) language to ensure that our generic AST is not language dependent, time permitting.

4 Architecture & Implementation

Smerge currently acts as a git mergetool, allowing developers to use our tool with ease. There are alternatives like Mercurial and Darcs, but we chose git on account of its wide popularity as a VCS. Most developers are accustomed to using git, so incorporating our tool into git leads to an even more convenient process. After a user runs a failing "git merge" command, they may invoke *smerge* through the "git mergetool" command:

```
git mergetool --tool=smerge <conflicting file>
```

This passes the following necessary file locations to our tool:

- $BASE: The original file modified into two conflicting versions, $LOCAL and $REMOTE
- $LOCAL: The conflicting file version the user has modified.
- $REMOTE: The conflicting file version of the branch the user is attempting to merge with.
- $MERGED: The output destination where the final merge is written.

The following diagram (Figure 1) illustrates the high level operations of our tool:

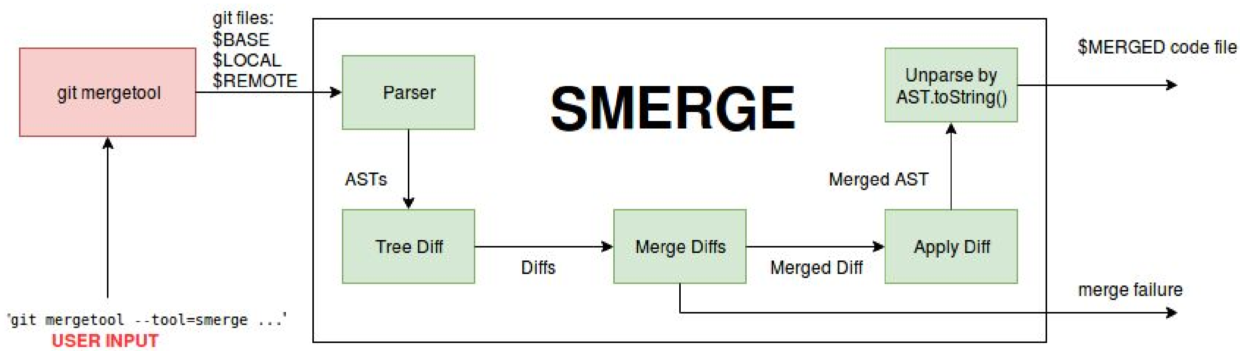

Figure 1

When a user runs the “git mergetool” command shown above, *smerge* will be run with the necessary files passed in as arguments (see Figure 1).These three conflicting file versions (base, remote, and local) are first parsed into their own abstract syntax tree (AST). Next, the base AST is compared to both the remote and local ASTs, producing two different tree diff objects. Each tree diff is defined as a list of actions, where each action represents a change from the source tree (base) to the destination tree (local or remote). These actions include inserting a node, removing a node, moving a node, and modifying/updating a node.

If the two diffs are non-conflicting (for example, if one tree moves a node and another one updates it), then they are merged into a single set of actions (otherwise our tool fails to produce a conflict resolution). These actions are then applied to the base tree, producing a “merged” AST. This final AST must be converted back into a source code file, and is written to the $MERGED file location.

If, however, the diffs are conflicting, then the merge fails and Smerge is unable to automatically resolve the conflict. In this case the user must manually resolve it.

4.1 Existing Tools

4.1.1 JavaParser & Conflerge

JavaParser [15] is an analytics focused tool that converts java source code into ASTs, and is the backbone of our predecessor Conflerge[4]. JavaParser’s AST’s are guaranteed to be unparsed into valid Java files, but are overly complex for our tool’s needs, and do not preserve the exact original source code when parsed such as code clarifying whitespace. As JavaParser works only with Java, we theoretically only need to import Conflerge into our own tool if we wish to use it at all.

One type of merge conflict that Conflerge claims to be successful with is import declarations. Conflerge handles these conflicts by combining the sets of import declarations from the local and remote files, and including the union within the final merge file. We included this feature in our implementation as it solved a lot of conflicts for Conflerge.

4.1.2 GumTreeDiff

GumTree[5] is described as "a complete framework to deal with source code as trees and compute differences between them," [5]. In other words, GumTree can parse source code files into ASTs, and then produce a diff between two such trees. As GumTree supports several languages (Java, C, Javascript, Ruby, and more in the future), this tool seems to have a lot of potential. GumTree is, however, heavily dependent on other parsers and does not support unparsing, making the tool extremely difficult to use directly for our purposes. For example, GumTree uses the Eclipse JDT parser for java code, which discards comments when generating an AST.

4.2 Implementation

Due to time constraints, the focus of our project is to create a foundational tool that can be easily built upon. As such, our implementation will be centered on a generic AST that the diffing and merging processes work with. This is so that each supported language only requires a parser that parses a source file into a self-unparsing tree (such that the tree's toString() method returns the original source content) compliant with the generic AST.

Our generic AST is made up of ASTNodes with the following properties:

- The content of the line of code it represents in a string (the "label"). This is used when unparsing the tree back into code.
- Its parent node, which is used for identifying if a node has been moved, for example.
- Its children, as a list of ASTNodes.
- A classification type which is one of the following: root node, import, whitespace, class name, method header, while loop, for loop, assignment, return, comment, or block comment.

Our implementation has 3 main components: a PythonParser, a Matcher, and a Differ.

- PythonParser, as described above, takes in a file and parses it line-by-line into an AST. Note that the PythonParser actually parses the file content into a PythonNode, which must satisfy the requirements of the generic AST nodes as described above. In addition, PythonNodes include information about indentation, which is relevant for Python but may not be applicable to other languages.
- Matcher takes the $BASE, $LOCAL, and $REMOTE AST trees generated by PythonParser and matches nodes by assigning unique IDs to each node. First, each node in the $BASE tree is assigned an ID. Then each node in $LOCAL is compared to each node in $BASE: matching nodes receive the same ID; unique nodes are given new IDs. The same is done with $REMOTE. Nodes are considered "matching" if (1) their types match, and (2) their labels are above a certain similarity threshold to each other. We calculate string similarity by using the Levenshtein distance [16] between two strings
- After Matcher matches the three trees, Differ iterates over the trees to produce two Action sets: one set of Actions that transforms $BASE into $LOCAL, and one that transforms $BASE into $REMOTE. Actions include the following:
    - Insertion: a node that exists in the other tree but not in $BASE
    - Deletion: a node that exists in $BASE but not in the other tree
    - Move: move a node to a different location in the tree
    - Update: update the node's label

  The two sets of Actions are then merged and performed on $BASE to obtain the final resolution. If merging the Action sets is not possible (e.g. a particular node is updated in one set and deleted in the other), Smerge currently throws an exception. In the future, we will look into ways to handle this differently if time permits.

We began with support for just one language, Python, due to it being a simpler language to parse. One important note: we are currently handling imports separately. If there are differing import statements, we simply put them all together. Since many conflicts are due to missing import statements, doing this solves those right off the bat without having to go through potentially large amounts of code.

4.3 Example

Let's run Smerge on some simple input files:

$BASE:	$LOCAL:	$REMOTE:

`import A`
 	
`x = 1`
	`import C`
	
	`x = 2`
		`import B`
		
		`if True:`
		`    x = 1`

4.3.1 Parsing and Matching

After parsing and matching, Smerge produces three line-based ASTs with IDs as follows:

```
$BASE:
(0) @root
   (1) import A
   (2)
   (3) x = 1
```

```
$LOCAL:
(0) @root
   (1) import C
   (2)
   (3) x = 2
```

```
$REMOTE:
(0) @root
   (1) import B
   (2)
   (4) if True:
      (3) x = 1
```

Note: “import A”, “import B”, and “import C”, and “x = 1” and “x = 2” receive matching IDs. This is because their similarity scores are above the threshold (which will be refined through testing). The “@root” node is just the root node of the AST tree (it does not hold any information about the source code) and is assigned an ID of 0.

4.3.2 Diffing

The Action set from $BASE to $LOCAL:
[Update 1, Update 3]

- Node (1), the import statement, is updated. The text “import A” is changed to “import C”.
- Node (3), the assignment of x, is also updated. “x = 1” becomes “x = 2”.

The Action set from $BASE to $REMOTE:
[Update 1, Move (Insert 3 under 4[0], Delete 3 from 0[2]), Insert 4 under 0[2]]

- (where x[y] represents the y-th child of the (x) node)
- Node (1), the import statement, is updated.
- Node (3) is moved under the new node (4), which is inserted as the second child of the root node.

The final merged Action set:
[Update 1, Update 3, Move (Insert 3 under 4[0], Delete 3 from 0[2]), Insert 4 under 0[2]]

- Update node (1), the import statement.
- Update the text of node (3), the assignment of x.
- Move node (3) under node (4) and insert node (4) as the second child of the root node.

Note: Even though (1) is updated differently in both trees, we are able to merge it because, as described above, we handle imports by putting them all together.

4.3.3 Merging

The actions are performed on $BASE to achieve a resolution. The resulting tree:

```
(0) @root
  (1) import C
 import A
 import B
  (2)
  (4) if True:
    (3) x = 2
```

4.3.4 Output

The final tree is unparsed by converting each node back to its label (original input files shown for comparison):

$MERGED:	$BASE:	$LOCAL:	$REMOTE:
`import C`	`import A`	`import C`	`import B`
`import A`			
`import B`	`x = 1`	`x = 2`	`if True:`
			`    x = 1`
`if True:`			
`    x = 2`			

This final merged file $MERGED is then outputted to the user.

5 Assessment and Experiments

5.1 Question

To what extent is Smerge successful at resolving conflicts? Does Smerge improve upon previous attempts at automatically resolving conflicts?

5.2 Hypothesis

Smerge will reduce the amount of merge conflicts experienced by the programmer.

5.3 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 to see how many conflicts arise as a baseline
4. Use Smerge's merging algorithm and record metric information
5. Compare the human resolution to the resolution presented by Smerge automatically. If they are the same, report the merge as correct, correct w/o comments, or incorrect.

5.4 Metrics

We begin by defining the true and false positives and negatives relevant to our evaluation:

- **True Positive:** A true positive results when a merge conflict is found, but it requires a manual user resolution to resolve. For example, if two developers change the same line of comments, there is no way to decide which one to keep and which one to discard. True positives also result when the tool correctly merges conflicts automatically.
- **False Positive:** A false positive results when there is actually no merge conflicts between the two parent files, but the tool still modified the code as if there was one.
- **True Negative:** A true negative results whenever the tool detects no merge conflicts and accordingly merges the two commits without error. This is the ideal merging case.
- **False Negative:** A false negative results whenever a merge conflict is found that the tool believes requires manual resolution, but the conflict itself is trivial enough that it does not. For example, if two developers commit the same file, but with different amounts of whitespace, some tools will flag this as a conflict when it should really be resolved automatically.

When following the above procedure, the following metrics will be recorded when analyzing a repository. The discussion above on true and false positives and negatives is contained within these metrics and detailed further below:

1. **Conflicts:** The number of merge conflicts (found in conflicting files, not commits) found in the repository's history with exactly two parents. If multiple conflicts are found between two parents, all of those conflicts are counted. This does not include conflicts that result from adding or deleting files in the repository.
2. **% Correct:** The percentage of conflicts that Smerge was able to resolve correctly. This number also is the percentage of false-positives encountered, as every conflict solved was one that the standard git mergetool flagged as a false positive.This means completely identical to the human resolution of the code and requires no manual merging.

3. **% Correct w/o Comments/Whitespace:** The percentage of conflicts that were resolved correctly with exception to cases where comments or custom whitespace were modified. This percentage also contributes to the false-positives category.
4. **% Unresolved:** The percentage of conflicts that Smerge aborted because attempting to merge would result in possibly undesired behavior. These conflicts would require manual resolution. This category includes both true positives and false positives, and requires manual checking to categorize the conflicts into the two.
5. **% Incorrect:** The percentage of conflicts that Smerge reported to have merged, but the solution it produced differed from the programmer's manual resolution. This category also contains false-negatives in the case where no changes were necessary but the tool still made some.

5.5 Results

As of now, we have only calculated the results for a few repositories, as our tool has not been fully optimized to work with some of the repos [6-14] listed here.

Repository	**# Conflicts**	**% Correct**	**% CorrectCW**	**% Unresolved**	**%Incorrect**
TensorFlow Models	7	0	0	100	0
Keras	33	0	0	100	0
flask	30	0	0	100	0
snallygaster					
django					
face_recognition					
ansible					
XX-net					
scikit-lean					
TOTAL:					

Our results can be reproduced by following the instructions described on the README file located in the version control repository here:
https://github.com/alvawei/smerge/tree/master/scripts

5.6 Analysis

Based on the three repositories we tested, we disappointingly noticed that our tool reported solving no conflicts. This is occurring mainly because the output of our tool is not yet precise enough to match the developer's manual resolution. We are currently working on a resolution for this, but right now the discrepancies are caused by actions made in our tool like the following:

- Whitespace is not exact. In some cases, our tool outputs extra lines of whitespace, causing the two files to be different.
- When merging two parents where the conflict is caused by conflicting variable names for the same variable, the tool chooses the wrong variable name. For example, if one branch named a variable x, and the other changed that variable name to be y, and the developers' manual resolution keeps the name x, our tool will sometimes choose y instead.
- Some developer manual resolutions are impossible to obtain because they add new code that wasn't included in the original two parents.
- Some of the merge conflicts analyzed actually required manual resolution: this is the case where our tool decided to categorize it as "Unresolved" correctly.

Currently, the tool works on the simple examples previously used. We are still working on extending it to more complicated examples from the tested repositories.

Unfortunately, at this stage in development, we do not have enough time to provide the below results because they require manual counting/analysis.

After gathering the data, we will analyze it by manually looking at the cases where the tool marked the merge as unresolved. When marked unresolved, this means that the the automation could not find a desired merge. Usually, this occurs when the merge is non-trivial and requires a manual resolution. However, there may be cases where the merge was trivial, but our tool failed to recognize it. From here, the team will analyze each "unresolved" conflict from the top three repos that had the most conflicts and categorize them into the above two categories, placing the results into a table as follows:

Repository	**# Unresolved**	**% T-Pos**	**% F-Pos**
Top Repo #1			
Top Repo #2			
Top Repo #3			

In addition to manually analyzing the unresolved category, we will look at the conflicts that were incorrectly merged to try and categorize some scenarios that caused the merge to fail.

5.7 Preliminary Conclusions

As of now, our data suggests that our tool is unable to handle any conflicts despite working on our simple test files. We cannot conclude that we have met our goals even partially yet.

However, if in the coming weeks our tool manages to resolve a significant percentage of conflicts automatically, then we will have some degree of confidence that our tool fulfilled our goal of handling more conflicts than git's standard merge tools. However, there are some challenges to this experiment in that it falls victim to both selection bias and undercoverage bias. We only apply our tool to a select few repositories, meaning that we can only infer that our tool was successful. For instance, they may exist some code bases where our tool was no better than git's standard merge tools because none of the merge conflicts were trivial enough for it to handle.

6 Risks and Challenges

The primary focus of our project will be to provide support for multiple languages. To achieve this goal, we have examined few different open-source parsers available on GitHub, including JavaParser and GumTreeDiff. These parsers have their own definitions of abstract syntax tree that are incompatible with our goals as they remove comments and destroy original source formatting. Lack of documentations in these project make it very difficult to understand their system and build our system around theirs. Consequently, we will build our system from scratch from end to end. Managing time properly will be crucial since we will also have to implement our own parser and AST on top of originally planned merger and unparser. Our hope is that building our own parser and AST will make it easier to build merger and unparser.

As we understand that we may not be able to complete multiple parsers for different languages due to time constraints, our implementation will focus on the generic AST that can represent different languages, yet can be merged and unparsed using the same algorithms. Our vision is that this generalization will allow our program to be flexible and easily extendable to multiple languages in the future. Different languages have different formatting; therefore they may require the AST to have different properties that are unique to each language. Since we do not have all the parsers already implemented, we need to foresee and envision what these requirements may be, at least for the languages we plan to support if not all languages.

We want to be able to resolve more merge conflicts automatically than other competitor tools. These tools use a well-defined differencing algorithm. To improve their performance, we would have to come up with a new algorithm or tweak the current algorithm to allow more merge conflicts to be resolved automatically. This process can be quite difficult in its own and could introduce more bugs as well. Additionally, we put ourselves at the risk of allowing undesirable merges when we add such flexibility. While we want to minimize the number of false-positive conflicts, it is more important that we do not allow true conflicts to be merged automatically. Automatically merging true conflicts will not only cause erroneous behaviors in the program, but also take away more time from the user to go back and fix the issue.

Current 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:
  - Continue basic implementation by completing Python parser/unparser, generic AST merging module/diffing algorithm.
  - Begin writing scripts for automatic evaluation
- Week 7:
  - Finish initial implementation of Python parser.
  - Finish initial implementation of AST matcher and differ.
  - Finish writing scripts for automatic evaluation and get some initial results. Debug tool if some repos cause bugs to occur.
- Week 8:
  - Improve matching and diffing algorithms to obtain better testing results.
  - Optimize algorithms to the point where we can run our tool on larger repos.
  - Begin using testing results to fine tune our matching thresholds and diffing preferences (which node to choose when multiple are available).
- Weeks 9 - 10:
  - Finalize project by proofreading specification/documentation, cleaning up code, etc. Tune implementation to work for all intended repositories during evaluation
  - Begin drafting final report
- Week 11:
  - Complete final report and presentation

Previous 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

Schedule Changes

In our initial schedule, we severely underestimated how long it would take to finish the basic implementation. We also ran into an issue around Week 4-5 where we scrapped our current plans for implementation in favor of other ones. Namely, we decided against implementing a GUI in favor of integrating our tool within git and providing a merging module that can take in parsers for other languages, thereby making the tool work for multiple languages. We also underestimated how long writing evaluation scripts would take. From this, we learned that coming up with a good idea is very difficult and requires time to consider the pros and cons of them. We also learned that it’s okay to change ideas midway as it’s much more valuable to pursue an actual improvement than to emulate a weaker version of existing tools. Additionally, we learned that time estimation tends to be severely underestimated--for example, we’ve been running into unexpected bugs left and right that require refactoring of the code base. To sum up the changes, we pushed back our basic implementation and updated it to be relevant to our goal of creating a tool that can handle different languages. This included details about writing a python parser and the generic AST model.

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
[5]Falleri Ean-R´emy, et al. "Fine-Grained and Accurate Source Code Differencing." *ACM/IEEE International Conference on Automated Software Engineering,* 14th ed., Vasteras, Sweden, pp. 313–324.
[6]TensorFlow Models - https://github.com/tensorflow/models
[7] Keras: Deep Learning for humans - https://github.com/keras-team/keras
[8] Flask - https://github.com/pallets/flask
[9] snallygaster - https://github.com/hannob/snallygaster
[10] Django - https://github.com/django/django
[11] Face Recognition - https://github.com/ageitgey/face_recognition
[12] Ansible - https://github.com/ansible/ansible
[13] XX-Net - https://github.com/XX-net/XX-Net
[14] scikit-lean - https://github.com/scikit-learn/scikit-learn
[15] JavaParser - https://github.com/javaparser/javaparser | https://www.javaparser.org
[16] Rosettacode.org. (2018). *Levenshtein distance* - Rosetta Code. [online] Available at: http://rosettacode.org/wiki/Levenshtein_distance [Accessed 11 May 2018].
[17] Git-scm.com. (2018). *Git*. [online] Available at: https://git-scm.com/ [Accessed 18 May 2018].
https://hal.archives-ouvertes.fr/hal-01054552/document

We spent an additional 15 hours on this assignment.