Empirical

Merge-Tree: Visualizing the Integration of Commits into Linux

Evan Wilde* and Daniel M. German

Department of Computer Science, University of Victoria, British Columbia, Canada

Correspondence: *Evan Wilde. Email: etcwilde@uvic.ca

Received ; Revised ; Accepted

Summary

With an average of more than 900 merges into the Linux kernel per release, many containing hundreds of commits and some containing thousands, maintenance of older versions of the kernel becomes nearly impossible. Various commercial products, such as the Android platform, run older versions of the kernel; due to security, performance, and changing hardware needs, maintainers must understand what changes (commits) are added to the current version of the kernel since the last time they inspected it in order to make the necessary patches.

Current tools provide information about repositories through the directed acyclic graph (DAG) of the repository, which is helpful for smaller projects. However, with the scale and number of branches in the kernel the DAG becomes overwhelming very quickly. Furthermore, the DAG contains every parents of every commit, while maintainers are more interested in how and when a commit arrives to the official Linux repository.

This paper make three contributions; a conversion from DAG to Merge-Tree, an implementation of a tool built on the Merge-Tree model, and a user study to evaluate and validate the implementation and model.

Keywords: Linux, git, visualizations, Merge-Tree

1 Introduction

Between 50k and 70k commits are added to the Linux kernel per version, requiring maintainers of older versions of the kernel to sift through thousands of commits and merges with tools that are unable to filter and effectively visualize projects at the scale of the kernel. Older versions of the kernel are used in embedded systems and mobile phones; for security purposes, performance needs, and changing hardware requirements, maintainers must be able to understand the changes being made in the current version of the kernel in order to produce the necessary patches for the older versions of the kernel. Tools like Gitk use a directed acyclic graph (DAG) model of the repository, showing

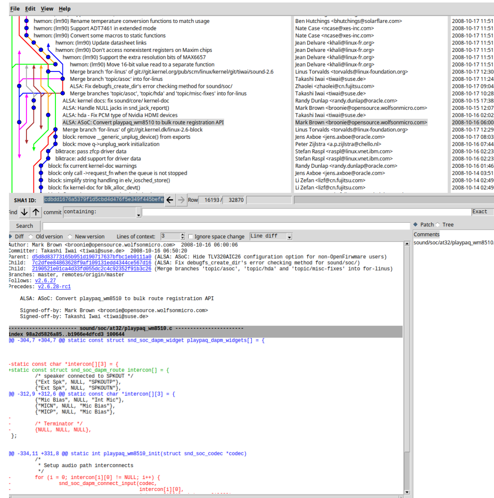

Figure 1: The Gitk interface centered on commit cdbdd1676a5379f1d5cbd4d476f5e349f445befe, *Commit 2* from the user study. The top-left pane shows the DAG and commit log preview, the top center pane shows the authors, and the top-right pane shows the commit dates. The bottom-left pane shows the full commit message, the parents and children of the commit, and the changes made to files in this commit. The bottom-right pane shows the list of the files modified by this commit.

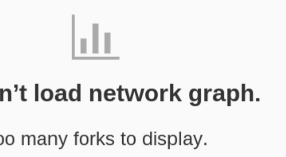

Figure 2: Github normally shows a visualization of the DAG, showing the commits, branches, and forks, but is unable to generate a visualization for projects at the size of the Linux repository.

all commits and merges in chronological order by when the commit was authored, not by when it arrived in the official Linux repository.

The DAG is able to provide a meaningful visualization in smaller projects; it enables users to see when changes are made, when these changes are merged, how each branch is interacting, and the point where a branch forks from another branch. In large modular projects, like the Linux kernel, the DAG becomes a mess of merges and commits (Figure 1) losing its visual meaning. In some cases, the Linux kernel is simply too large for the system to generate a visualization; Github provides a DAG view for a repository, but is unable to display the visualization for projects at the scale of the Linux of the kernel (Figure 2). Between 60k and 70k new commits are created for the Linux project every year; according to previous work(9), a commit takes a median of 30 days from the time it is authored until it arrives in the official repository. The snapshot of the kernel tomorrow may be different than the snapshot from today, containing new commits authored in the past; distinguishing these new commits from the commits in the snapshot from today is not trivial.

One major challenge with visualizing the arrival of commits to a repository is that git does not store the date that a commit was merged into another branch, including the master branch. To complicate the problem, the DAG only has references to the parents of a commit (a model necessary for the operation of git), but maintainers would prefer knowing the path a commit followed to reach the master repository. Tracing a path that any commit followed to the master repository would imply that for any given commit, we know the next merge in the direction of the master branch. A user could inspect the commits that arrived into the master branch within a given time-frame by checking which commits were merged during that time-frame.

This paper makes three contributions; first, we describe a method of converting the DAG of the Linux repository into the Merge-Tree of the repository, that represents the path used by a commit to reach the master branch; second, we present an implementation enabling the inspection and visualization of merges in the Linux project using the Merge-Tree model; third, we validate the Merge-Tree model and the implementation of the visualization through a controlled user study. We further discuss the issues of generalizing the model to other repositories, but present an updated version that improves the performance of the algorithm by pruning the DAG.

These methods and visualizations are implemented in a web-based tool called Linvis[1]. Our visualizations and tool provide information about the location of any given commit or merge in its respective Merge-Tree, the files edited, the modules edited, and the commit message. Linvis provides a search engine that filters commits by commit hash, terms from the commit log, the author, filenames, and other information. The results are then grouped together by the root of the corresponding Merge-Tree.

2 Background

Git is a distributed version control system that is used in many open-source and closed-source projects. It enables many developers to write code simultaneously, working on separate branches of the same project. Git uses a directed acyclic graph model (DAG) for storing the necessary information regarding the state of the code at any point in time. Git refers to nodes of the DAG as commits, not distinguishing between the code-carrying commits and the structural commits that bring two branches together. In this paper, we will refer to the individual code-carrying commits as commits, the structural merge commits as merges, and refer to both types without discrimination as repository events.

Each repository event contains an ordered list of one or more parents, except the initial commit which has no parents. The parent relationship points from the current event toward the initial commit. Commits have a single parent, while merges will have at least two parents, the parent on the current branch followed by the branches being merged into the current branch. Git allows for the merging of multiple branches simultaneously in octopus merges,

[1] Linvis is currently available at http://li.turingmachine.org

where the order of the parents represents the order that the branches were specified for being merged[2]. A *foxtrot*[3] merge occurs when the parent on the current branch is replaced by a parent from a branch being merged into the current branch, thus confounding the branch relationships.

The repository events in the DAG are immutable; once a repository event is created, it can't be changed. Git allows operations to alter repository events and re-order them, but this will create a new event with a new commit hash. This property makes a repository event unable to record the traversal of merges into the master branch of the repository. Introducing an additional field to track the path to the master branch would create a circular dependency between the parent and child. Any updates to the DAG would modify the commit hashes for all repository events in the repository. To alleviate this, git provides the command `git log --children` which traverses the DAG and prints the inverted edges relationship between events.

Under perfect conditions, visualizations of the DAG show how a repository changes over time; where logical branches are being made, where merges are happening, and who is making the changes. Most repositories are relatively simple, with relatively few, small, branches with a clear, well defined master branch. However, in large, active repositories it becomes very difficult for developers to use the DAG to identify the master branch, and as a consequence, where branches start and are merged on the path to the master branch.

3 Merge-Tree model

The Merge-Tree model abstracts the DAG of the repository into a set of Merge-Trees. Each Merge-Tree is rooted at a merge into the master branch. The leaves of this tree are the commits and the merges are the inner nodes. A Merge-Tree is built recursively with every merge in the Merge-Tree merging a sub-Merge-Tree. Any merges that a commit must pass through to reach the master branch become an inner node of the tree. Effectively, a Merge-Tree is a tree that shows the path that commits follow in their way to the master branch of a repository. The Merge-Trees also help understand how commits were grouped together in order to be integrated. In this model, not only has the DAG been inverted (and simplified), but the entire notion of parent-child relationship has been reversed. Due to this property, the terminology will be different depending on the model we are referring to; when we are referring to the Merge-Tree, the parent is the next node toward the merge into the master branch, or the root of the tree. When referring to the DAG, the parent relationship is in the opposite direction, from the root toward the branch-point. In addition to identifying the path that a commit took to being merged, we are able to aggregate the commit metadata at merges as we retain the parent-child and child-parent relationship for each event and know which commits belong to the merge.

[2] One octopus merge in Linux (2cde51fbd0f310c8a2c5f977e665c0ac3945b46d) has 66 parents.
[3] See http://bit-booster.blogspot.ca/2016/02/no-foxtrots-allowed.html for a full description of the issue.

To illustrate this model we will use a small example: assume the commits represented in Figure 3 show the sequence of events in a repository. The sequence starts with the initial commit in the master branch of the master repository at time t_0. Repository event 1 is a commit, which gets forked into a separate repository, *Repo A*, where another commit is made, event 2. Event 5 is a merge event, merging event 2, 3, and 4 into *Repo A*. Event 5 is branched from, commit 6 happens in the new branch, while commit 7 is added simultaneously to the original branch in *Repo A*. Events 11 and 12 are both merge events, merging changes made in *Repo A* into the master branch of the master repository. As every repository is a first-class repository, including local copies and forks, git does not distinguish between forked repositories and branches, and in neither case does it explicitly record where a commit was made. In this case, commits are performed in various repositories and branches. The DAG representation of these events is shown in Figure 4.

The DAG does not retain information about where a commit was originally created. The Merge-Tree form (Figure 5) does not completely rebuild the lost information, but is able to re-create the sequence of merges required to merge the event, and the depth, in the tree, of that event. The Merge-Tree does not include information about how the repository events that are the children of a merge are related. The order of the children does not matter.

Combining the DAG and the Merge-Tree results in a hybrid structure (Figure 6) that no only shows which merges a commit passes through, it rebuilds the order of those commits. In the example shown, this hybrid structure makes it clear that merge 9 merges the changes made in 6 and 8 into the commit 7. Finally, 11 merges merge 9 into the master branch.

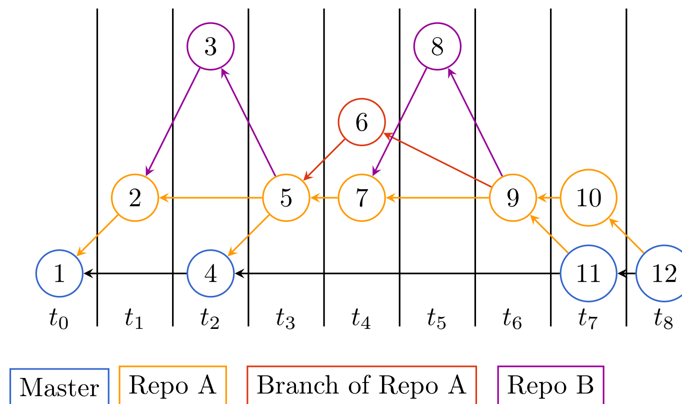

Figure 3: An example sequence of events performed in different repositories. The horizontal axis represents time. The branches and repositories are aligned horizontally, and color-coded. Each commit points to its parent. The initial commit is at time t_0, and the head is at t_8.

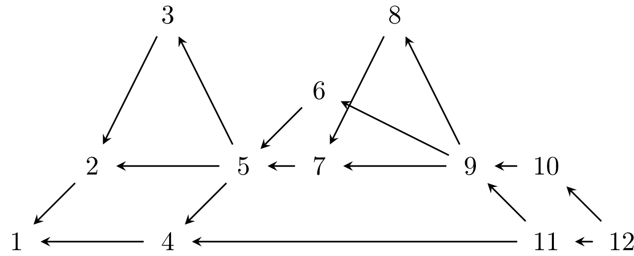

Figure 4: DAG representation of the commits represented in Figure 3. The DAG loses information about which repository the commit is performed in and through which merges it has passed on its way to the master branch. The DAG does not even distinguish the master branch from other branches.

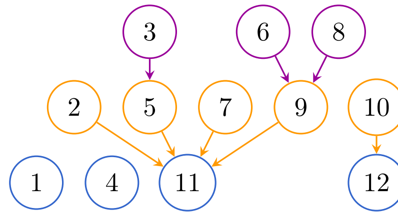

Figure 5: The Merge-Trees computed for each commit in Figure 4 showing the path that each commit takes to be merged into the master branch of the repository. This does not indicate how the events being merged are related. We retained the numerical order of the events, but the order can be arbitrary.

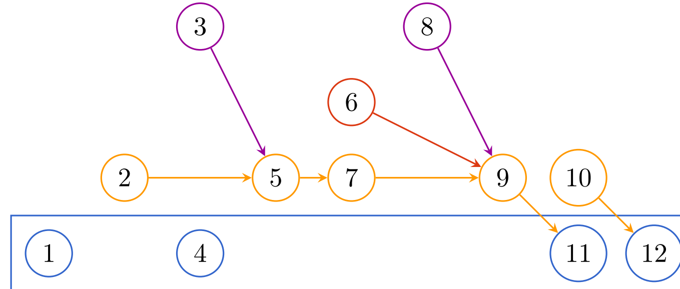

Figure 6: The Merge-Tree applied to the inverted DAG to show the order that commits are created, as well as the merge sequence for the events in Figure 3. This requires a post-processing step involving the Merge-Tree produced by Algorithm 1 and the DAG.

3.1 Computing the Merge-Tree of the DAG of Linux

Computing the Merge-Tree from a DAG for any repository may not be possible; however, certain features of the development process of Linux make it feasible to compute the Merge-Tree for the Linux repository. First, the master branch of Linux is maintained by Linus Torvalds, and only Linus has write access to it. We have verified this assertion in previous research (9). We have developed a heuristic that is presented in Algorithm 1. In short, the algorithm first identifies the commits made directly to the master branch, whereafter it recursively determines the shortest path, using the DAG, from each commit to the master branch using the inverted DAG.

The algorithm is broken into two phases. The first is determining which repository events are on the master branch. This is done by traversing the first parent from the master branch reference to the commit that has no parents. The second phase is encompassed by the function $MergeAtMaster$ which determines, for each commit, which merge the commit is merged at, the depth (as variable d in the algorithm), and the next merge on the path to the master branch. The function $MergeAtMaster$ has two parts, the first for determining the depth, from the master branch, that the repository event is at. The second phase determines the merge into the master branch, and the next merge on the way to the master branch. The distance is by shortest path, staying as close to the master branch as possible. If there is a tie between two paths, the path that merges into the master branch sooner is taken.

To demonstrate the behaviour of the algorithm, we use a short example, computing the merge at commit 5 in Figure 3. $MergeAtMaster$, recurses along the children of the nodes it visits. Eventually every child of every node along the path will be visited at least once. Without loss of generality, suppose that the path recursed along is from node 5 to 6, 9, 10, and finally 12.

The depth for each, except 12 (a merge into the master branch), is initialized to infinity, the merge into master is blank, and the next merge is blank. Merges into master trivially have a distance of 0 from the master branch, and it merges itself into the master branch. The recursion at 12 returns the triple $(0, \varnothing, 12)$ to the call from 10. 12 is a merge commit and 10 is not the first parent, so the temporary depth, d_c, is incremented to 1 and the temporary next merge, $next_c$, is changed to 12. 1 is less than infinity, so the depth is set to 1, the merge to 12, and the next to 12. This returns the triple $(1, 12, 12)$ to the call from 9. 9 is the first parent of 10, so no changes are made to the temporary variables.

The call to 9 recurses to the second child, 11. 11 is a merge into the master so it returns $(0, \varnothing, 11)$ to the call from 9. 9 is not the first parent of 11, so the d_c is incremented to 1 and $next_c$ is changed to 11. The distances d_c and d are the same, so to break the tie, we use the time. 12 was merged after 11, so 11 replaces 12 as the merge into the master branch for 9, as well as being the next merge. The call for 9 returns the triple $(1, 11, 11)$ to the call for 6. 6 is not the first parent of 9, so d_c is incremented and $next_c$ is changed to 9, as 9 merges 6. 2 is less than infinity, so the d is changed to 2, the merge to 11, and the next merge to 9. The call to 6 returns the triple $(2, 11, 9)$ to the call for 5.

The call for 5 recurses on the second child of 5, calling on 7, which calls 8, and then 9. 9 can continue, but if the implementation of the algorithm uses memoization, the call to 9 can immediately return the triple $(1, 11, 11)$ to the call for 8, and avoid an exponential runtime. 8 is not the first parent of 8, so d_c is incremented to 2 and $next_c$ is changed to 9. 2 is less than infinity, so d is changed to 2, $merge$ to 11, and $next$ to 9. The call to 8 returns the triple $(2, 11, 9)$ to the call for 7, which recurses on the second child of 7, 9. 9 returns the triple $(1, 11, 11)$. 7 is the first parent of 9, so the depth is not incremented. d_c is less than d, so d is changed to 1, m to 11, and $next$ to 11,

returning $(1, 11, 11)$ to the call for 5. d_c is less than d, so d is changed to 1, m to 11, and *next* to 11. There are no other children, so the function halts.

3.2 Evaluation

Merges that do not have conflicts provide information to verify this heuristic. If a merge does not contain a conflict, it records a summary of the commits that it merges. See Figure 7 for an example. This summary contains a list of the first 20 non-merge commits in the merge, including their one-line log description, and the total number of non-merge commits in the merge.

```
Merge: 8cbd84f fd8aa2c
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Tue Aug 10 15:38:19 2010 -0700

Merge branch 'for-linus' of git://neil.brown.name/md

* 'for-linus' of git://neil.brown.name/md: (24 commits)
md: clean up do_md_stop
[... edited for the sake of space]
md: split out md_rdev_init
md: be more careful setting MD_CHANGE_CLEAN
md/raid5: ensure we create a unique name for kmem_cache...
...
```

Figure 7: Example of how merges record a subset of commits being merged. The commit only shows the first 20 one-line summaries messages for the 24 non-merge commits it merged. The ending "..." is part of the log and represents that other commits were merged.

We used this information to evaluate the accuracy of the Merge-Tree model extracted from the DAG. The method we followed started with the extraction of the commit history up to July 20, 2016 form the Linux repository. We computed the Merge-Tree of every commit until that date. Since Linus Torvalds mostly does merging directly into the master branch, we assumed that every merge by Linus is the root of a Merge-Tree, later detecting merges that do not merge into the master branch and removing them from the set of root merges. As described above, the log of a merge-commit usually contains the number of commits in the merge the first 20 summaries of commits being merged. We extracted merges by Linus Torvalds using the command `git log --merges --author='Torvalds'` and compared the number of commits stated in the log message with the number of commits in the Merge-Tree. We also used the summaries of the commits found in the merge (not necessarily all—see above) to make sure those commits were in their corresponding Merge-Tree. For example, for the merge in Figure 7 we would expect that the Merge-Tree rooted at `8cbd84f` contains 24 commits, and the one-line summaries corresponds to commits in that Merge-Tree. We also inspected those with differences to make sure they were true errors. The results can be summarized as follows:

- Five merges were false-errors because their logs did not contain accurate information (were probably edited by hand). For example in `42a579a0f...` one commit summary was missing (the line was empty), in `c55d267...` the summaries were reordered.

- The heuristic correctly identified that 79 of Linus merges (between Jun 7, 2014 and Jun 2, 2014) were made to a branch (not master). This branch was merged at `3f17ea6d...` which contained 6809 commits.

- The heuristic worked perfectly until Sept 4, 2007, the earliest date that it could be verified. Before this date, and until Dec 12, 2006, merges did not include a summary of the commits they included, hence making it impossible to verify; during this period, however, we correctly identified the merges by Linus into the master branch.

- Before Dec. 12, 2006 (1542 merges) our heuristic breaks due to the presence of a *foxtrot* commit (`c436688...`), which confounded the true master branch.

In summary, of the merges after Sept. 4, 2007, our heuristic was correct in 100% of the 16,680 commits. It failed in 1,542 commits before Dec. 12, 2006 and in 836 it appears to be correct (Dec 7, 2006 to Sept 4, 2007).

4 Visualizing the Merge-Tree of Linux

The goal of Linvis is to simplify the navigation of the kernel commit information, specifically focusing on merges. This is done by leveraging the Merge-Tree to inspect how commits are merged on the path to the master branch of the repository.

4.1 Use cases

We designed Linvis with two use-cases in mind, though a user may switch between the cases as they work.

Use-case 1: top-to-bottom approach

These are users that are maintaining a section of the kernel and would like to pick a merge (including all commits that it merges) and merge it directly into their current repository. This is useful for reducing the amount of re-implementation work necessary for integration. For these users, it is important to have the ability to aggregate metadata about files and modules being effected by the merge. Also, it is important for these users to be able to navigate from the root of the Merge-Tree toward the leaves.

Use-case 2: bottom-to-top approach

These are users that start with a known merge or commit and would like to see what other changes are being made in commits that are in the same merge, including knowing the Merge-Tree they belong to. This is useful to see what other commits are related to the current commit and how they get joined into merges that eventually end in the

master branch. This is primarily for maintainers that need to perform some specific cherry picking of commits. We must provide these users a mechanism for navigating from a single commit toward the master branch, allowing them to see other commits that might be related to their original commit.

4.2 Data Model

In our visualizations, we leverage the Merge-Tree model described in Section 3. In this model, commits are either on the master branch, or part of a tree which is rooted at a merge into the master branch. Each commit, whether a merge or non-merge, has only one parent, with the exception of the root which has no parents. Non-merge commits contain the metadata for the changes made. This metadata includes the files changed, the lines added and removed from each file, the author, the date the commit was merged into the merge that led to being merged into the kernel, the date the commit was authored, the patch, and the commit log. Merges contain similar metadata, the author of the merge, the committer of the merge, if it has been rebased, the log, the commit date, and the author date, and potentially, changes necessary to address conflicts during the merge.

5 Design and Implementation

We constructed a web-based tool called Linvis in order to navigate and inspect the Merge-Tree visualizations of the kernel. Creating the web-based tool enables users to use the system without having to install additional software or store a large database, which makes it more accessible, easily maintainable, and platform independent. Linvis uses the following mechanisms to reach our goals of better navigation and better explanation of the selected changes.

- Filter by searching
- View aggregated summaries of authors, files, and modules
- Visualize Merge-Trees

5.1 Search

Linvis provides a search engine for navigating within the kernel, filtering commits that are not relevant. The search engine takes a plain text query from the user and returns the results that are relevant in the order of relevancy. When computing relevancy, the engine uses the commit log, the author name, the filenames, the commit hash, the date the commit was authored, and the date the commit was committed.

Before presentation, the results are grouped by Merge-Tree root. Each group has a link to the root-node of the tree at the top, followed by a table of commits and merges within the tree that match the query, shown in Figure 8. The table includes the relevancy rank of the entry, the commit log message preview, author, the date the repository

Figure 8: A single Merge-Tree tree in the search results of Linvis, showing the link to the root at the top and a table of the relevant commits in the bottom, sorted by relevance.

event was committed, and the date it was authored. The merge tree groups are ordered based on the mean of the relevancy scores of the relevant results in that tree.

5.2 Summarization

Linvis uses seven tabs to show the information and visualizations for a selected repository event. The summarization tabs are; messages, files, modules, authors; the visualization tabs are; list tree, pack tree, and Reingold-Tilford tree.

The message tab shows the full commit log message. This does not include the diff, but given the commit hash, this information can be found directly from the repository.

The files tab shows an aggregated table of all files that were modified in a merge. It includes metrics like the number of lines added, lines removed, total lines modified, and the delta, summed across all commits in the Merge-Tree tree that modify this file. A details drop-down button allows a user to see exactly which commit makes the changes, as shown in Figure 9. If the current repository event being viewed is a commit, the aggregate views will only show modifications made by the commit.

The modules tab shows the modules modified in the Merge-Tree. Like the files tab, the modules tab uses a table to show the name of the module, the number of commits that are in the Merge-Tree tree that work with the module, and a details button to provide the links to those commits.

Figure 9: Table showing the modified files in a merge, with the second entry expanded to show the commit that makes the changes.

The authorship tab is very similar to the files tab, but shows the authorship information. It shows the sum of the number of lines added, removed, modified, and the delta within the Merge-Tree. It also shows the number of files

that were modified by the author. The details tab is organized slightly differently. Instead of organizing the results by commit, the details are organized by file, showing which files were modified by the author in this Merge-Tree (Figure 10).

>	Takashi Iwai	1	1	0	1	1
∨	Randy Dunlap	2	32	40	72	-8

Show 10 entries Search:

Commit	Filename	Lines Added	Lines Removed	Total	Delta
1c85cc6445	sound/core/pcm_native.c	8	16	24	-8
1c85cc6445	sound/core/pcm_lib.c	24	24	48	0

Showing 1 to 2 of 2 entries Previous 1 Next

Figure 10: Table showing the authors who made changes in a merge. The entry for Randy Dunlap is expanded, showing the modifications to each file that were made by Randy.

5.3 Visualization

The ability to easily visualize the integration of commits into a project is what makes Linvis unique from other tools. Linvis implements three visualizations, the list tree, pack tree, and Reingold-Tilford tree.

5.3.1 List Tree

The list tree, depicted in Figure 11, is constructed as a series of nested lists. The nesting indicates the parent-child relationship. This visualization is text-based enabling fast navigation to commits using the built-in search in most web-browsers. The list tree is rooted at the current repository event; if the current event being inspected is a commit, only that commit will be shown. Conversely, if the selected event is the root, the entire Merge-Tree tree will be shown.

5.3.2 Reingold-Tilford Tree

The Reingold-Tilford tree(12), depicted for two trees in Figure 13, is the classic tree visualization, with the root at the top, leaves at the bottom, and edges between them showing the parent-child relationship. This tree visualization works very well in many cases, but does not easily visualize very wide trees, with many repository events on a single level.

- Merge branch 'for-linus' of git://git.kernel.org/pub/scm/linux/kernel/git/tiwai/sound-2.6
  - Merge branches 'topic/asoc', 'topic/hda' and 'topic/misc-fixes' into for-linus
    - ALSA: kernel docs: fix sound/core/ kernel-doc
    - ALSA: hda - Fix PCM type of Nvidia HDMI devices
    - ALSA: Handle NULL jacks in snd_jack_report()
  - Merge branch 'topic/asoc' into for-linus
    - ALSA: ASoC: Convert playpaq_wm8510 to bulk route registration API
    - ALSA: Fix debugfs_create_dir's error checking method for sound/soc/

Figure 11: The list tree visualization

When a user first looks at the visualization, the current repository event is placed in the center. Furthermore, it is highlighted in bright orange. Commits, the leaves, are shown in white, while the merges are colored shades of blue where lighter blue indicates fewer child events of that node, and darker blue indicates more children.

5.3.3 Pack Tree

Pack trees(13) are useful for displaying an overview of large data sets. The initial goal for the pack tree was for visualizing file systems, which are similar to git repositories in that they are relatively shallow, but very wide trees. The tree is represented by sets of nested circles. The largest circle, containing all of the other nodes, is the root, while the smallest circles are the leaves. In our representation, the root maps to the merge into the master branch, while the leaves are the commits.

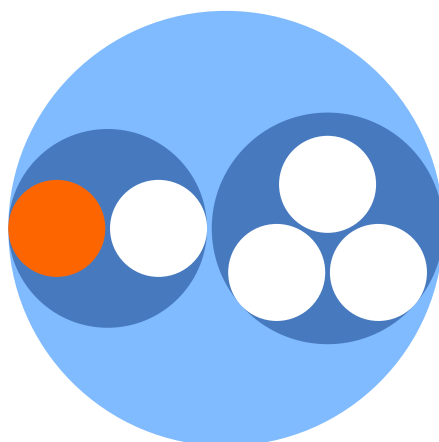

Figure 12: The pack tree visualization, with the root depicted as the outer-most circle, containing all other nodes, the leaves depicted as white circles containing no other nodes. The currently selected event is shown in orange.

We depict the pack tree in Figure 12. Like with the Reingold-Tilford tree, Linvis uses white to indicate the individual commits, and orange to indicate the currently selected node. The inner merges are colored with shades of blue to indicate the depth in the tree. Darker shades of blue indicate that the merge is deeper in the tree; the root will be the lightest shade.

6 Related Works

Version control systems monitor the development lifetime of software projects. This makes the version control system vital in providing information about how a software project is being developed. To do this, the user must be able to gain a clear understanding of how commits are being integrated into the project, along with understanding what changes these commits are bringing with them. These are the tasks that we set out to solve with the Merge-Tree

model. Ideally, we could compare our model against another that is designed with these goals. To our knowledge, we know of no git repository visualization tool that has these specific goals. This may be due to issues with finding the master branch of the repository which is a non-trivial task, or due to other factors. While we have found no tool with the express goal of showing how a commit is integrated, and providing a summarization of a merge, there has been a lot of work done in providing visualizations of various aspects of a repository.

Many tools work to address the issues in communication between developers in inter-team collaboration work. Codebook(1) uses a data mining technique to determine the developer of a piece of code, the program manager who wrote the specification for the code, and the program managers and developers on the team who were working together. Hipikat(5) is another tool with a focus on communication. Where Codebook focuses on developers working on a project, Hipikat is focused on enabling easier integration of new developers to a project by providing them with easily-searchable artifacts of the changes made. Codebook is useful for pairing a contributor with the original developer; however, the developer may not have worked with the piece of code in years. A Hipikat program may provide more information to the maintainer as it records the artifacts of why certain design decisions were made when they were made using other tools like Bugzilla and CVS. Neither tool is sufficient in meeting our goals to provide a summary of the topology of the kernel repository through a visual tree.

Most visualizers provide a visual presentation of a certain aspect of a repository. Fractal Figures(6) uses a unit square to represent a portion of a project, then partitions the square based on the proportion of an author's contributions to that portion of the project. EPOSee(2) and Evolution Radar(7) perform further analysis, determine which files are made together, and what changes are made over a sequence of commits, though the goals behind these projects is different.

Codebook, Hipikat, Fractal Figures, EPOSee, and Evolution Radar all work with data from CVS repositories. Our goal is to provide information about git repositories, specifically the Linux master repository. Fewer tools are available for generating visualizations and summaries of git repositories, potentially due to the DAG model used by git.

Gource is a tool for providing an interactive timelapse of the state of a repository(3). In the timelapse, it shows who contributes and what type of contribution a developer is making. These contribution types are one of, adding a file, removing a file, and changing a file. While the timelapse is interesting to watch, it does not provide any additional explanation of the changes actually being made, only the frequency that they are being made and who is making them. Codeswarm(11) is similar to Gource, using an organic approach to visualizing events in the repository. Unlike Gource, which focuses on the files, Codeswarm focuses on developers and the number of commits they are making.

Git itself is shipped with summarization and visualization tools; Gitk and `git log --graph` are both built-in tools for the purpose of visualizing and browsing the DAG of the repository. Many other tools are designed with a similar visual metaphore as what is presented in Gitk, including GitKraken, Github Desktop and web interfaces,

the GitLab web interface, the Bitbucket web interface, are a few examples. The Gitk interface is built around the central DAG viewer. The DAG displays the repository events on their respective branches, the author, and the date that the event was authored. A user can select an event to view addition information about it, including the full commit log, the parents and children of the event, and the diff generated if modifications were made. In the case of the Linux kernel, merges are not resolved as merge conflicts in the master repository, but by the developers prior to merging through rebase operations. With this development model, no merges will have modified any files, and will not show a diff. While the merge does not make changes directly, the effects of merging the commits persist, but Gitk is unable to provide an aggregated view of the files modified, the patches for those files, and the authors who contributed commits to that merge.

Our tool is primarily aimed at presenting the hierarchical structure of the Linux git repository. We use tables for presenting the summarized information of the commits and merges, but this information could also be presented in a graphical form. Various graphical forms for displaying file and authorship data exist, the principal forms being matrix views, city scapes, bar and pie charts, and networks (8). Any of these data visualization metaphors are applicable to our system.

7 Controlled User Study

We conducted a study to quantitatively and qualitatively evaluate the effectiveness of Linvis compared to the DAG-based visualizations found in Gitk or the git command line. The study is broken into three task-based sections; conceptual tasks, summarization tasks, and participant opinions. The study was performed during June of 2017 in a controlled environment running Ubuntu 14.04. To evaluate the DAG-based visualizations, participants were allowed to use both Gitk and the git command-line tools together, which we will refer to as Gitk. Linvis was used for evaluating Merge-Tree-based visualizations.

The primary goal of the study is two fold; first determining if the DAG-based visualization is sufficient for conceptual understanding; second comparing Linvis and Gitk to determine which is more capable of providing users with a summarization of various metrics involved with integrating a commit into the repository. The conceptual questions are to determine if the DAG-based visualizations (Gitk and git command line) provide enough information to determine how a commit, and any related commits, are merged into a project. Linvis displays this information directly in the form of a tree, so we will not be performing a comparison on this section, only evaluating the results from Gitk. The summarization tasks compare Linvis and Gitk, with the participants using one tool followed by the other on two commits from Merge-Trees of differing sizes. In both of these sections, the participant is given a task that they are to find an answer to; we evaluate the time it took, if the answer was correct, and how far from the right answer

they were. The user opinions provide us with a clear comparison between the tools from the point-of-view of the participant, providing us information about which aspects of each tool they preferred.

7.1 Tasks

Table 1: Conceptual Tasks

Task	Description
T1	Draw a diagram showing how this commit was merged into the master branch, along with any other related commits
T2	How many individual commits are related to this commit?
T3	How many merges are involved with merging this commit into the master branch?

Table 2: Summarization Tasks

Task Set	Task	Description
Merge	T4	What is the series of merges involved with merging this commit?
	T5	What other commits are merged?
Authorship	T6	How many authors are involved?
	T7	Who contributed the most changes?
Files	T8	How many files were modified?
	T9	Which file had the most changes?
Modules	T10	Which modules does this Merge-Tree involve?

The conceptual tasks are designed to gain better insight on the comprehension of the visualization in Gitk, and determine if it is sufficient for understanding what other commits are related to a given commit, and how those commits are merged. The tasks are outlined in Table 1. The tasks are performed in order using Gitk. T1 has the participant draw a diagram of the Merge-Tree, which will be used to answer the other two questions. This enables us to see issues with comprehending the DAG visualization. Participants are given 10 minutes to perform this task, which also is used to familiarize the participant with the Gitk interface, if they are not already, and with the set of commits they will be working with. T2 and T3 are drawn directly from the diagram from T1, giving quantitative metrics to measure the comprehension. T2 asks the participant to verify the number of commits, and T3 asks for the number of merges. The three tasks are performed on both commits before continuing to the summarization tasks.

The summarization tasks, listed in Table 2, are designed to compare the ability of the participants to summarize information about the two commits used in the study when using Gitk versus Linvis. The tasks are broken into four sets based on what is being summarized: merges, authorship, files, and modules. The order that the participants perform each task is randomized within each task set, and the order of the task sets are then randomized with the

exception of the merge task set which will always come first. The order that the tools are used is kept consistent for a participant, but randomized between participants. We measure and evaluate three metrics for each task: correctness, accuracy, and timing. Correctness measures whether the answer was correct or incorrect. Accuracy measures how far from being correct the provided answer was. Timing is the duration, in seconds, that the participant took to answer the task. We begin timing the task at the end of the question, and stop timing before the last modification to the answer. For this reason, it is possible for times to overlap between tasks if the participant changes their answer after answering another question. This will also modify the answer used for measuring the accuracy. We measure the timing from the screen capture.

While the merge task set is part of the summarization task group, it focuses less with summarization of the commits and more with comprehension of the model visualizations. The task is designed with two goals in mind; first we want to measure and define which commits and which merges the participant has defined to be in the Merge-Tree; second to help solidify which commits and merges will be used in the merge tree.

T4 provides a concrete series of merges that the participant believes to be integrating the commit into the repository. T5 provides a concrete set of commits that are related to the commit. We place these tasks first as the answers to these will be necessary for the participant to answer the tasks that follow, specifically finding the merge into the master branch. T6 and T7 are related to the authorship of a merge, determining how many authors are involved and who made the most changes. T8 and T9 are related to the files that were modified in the merge, determining how many files were modified and which file had the most changes. T10 is related to the modules that were modified in the merge. Modules are not explicitly defined in git, but are a property of the Linux repository that we noticed while looking at various commits. To find the module name, take the text up to the first colon in the git log preview. For the log preview *"ALSA: kernel docs: fix sound/core/ kernel-doc"* the module is *"ALSA"*, e.g.

The correctness metric is simply whether the answer was correct or not; an accuracy of zero indicates a correct answer. The accuracy metrics are measured slightly differently between tasks due to different requirements and structures in the answer. In task T4, both the order and the merges are important. We define the distance to be the edit distance with insertion, deletion, and transposition at unit cost. The accuracy is the minimum edit distance necessary to make the answer correct. T5, T9, and T10 are concerned with the set of elements in the response; the order is not important. We use the Levenshtein distance, with insertion, deletion, and substitution operations of unit cost, from the provided answer to the correct answer. The answers to T6 and T8 are single number values; the distance is measured as the absolute value of the difference between the provided answer and the correct answer. T7 has only a single correct answer, thus the distance would either be zero or one. For this reason, it is acceptable to omit the accuracy metric from this task.

Statistical significance testing is performed to verify that the results are meaningful. We use *alpha* = 0.05 or a 95% confidence level. The McNemar χ^2 test(10) with continuity correct is used to test the statistical significance of the hypothesis that Linvis improves correctness of the participants. The χ^2 threshold statistic at $\alpha = 0.05$ is 3.841. The Wilcoxon(14) test and Cliff's effect size(4) are applied to the accuracy and timing metrics to determine the significance of the difference between Linvis and Gitk. The Wilcoxon test is applied between commits to determine if the size of the Merge-Tree effects timing and accuracy. The null hypotheses and alternative hypotheses are stated in Table 3.

The user-opinion questions, listed in Table 4, are designed to determine which tool provides a better user experience for summarization and comprehension tasks, and what aspects of each assisted with their understanding. T11 asks for the preference of the participant, given that their goal is comprehension and summarization. We recognize that neither tool is perfect, and participants may have complaints about both tools, but are interested in what parts of each tool assisted them with understanding the events in the repository. T12 is meant to address this.

7.2 Commit Selection

We include two commits in the study to detect differences in the correctness, accuracy, and timing between Merge-Tree sizes. The order that the commits are presented to each participant is randomized. We analyzed the 15096 Merge-Trees from April 16th 2005 to October 14th 2014, which corresponds to the releases Linux 2.6.12-rc3 and Linux 3.17-rc1. 25% of the trees contain at most a single commit not including the root, while 50% of the trees contain at most seven non-root nodes. 75% of the trees contain at least 51 nodes, and the largest tree contains 7217 nodes.

From this information, we selected one tree with a single commit, and one tree with seven non-root nodes to represent a small tree and a medium-sized tree. A majority of the trees are flat, where all nodes merge directly into the root node. Of the 8031 trees containing at least seven non-root nodes, only 593 contained at least a single internal merge node. Trivially, trees with a single node cannot have any internal nodes, and of the 624 trees with seven non-root nodes, only 135 contained at least one inner node. To increase the complexity of the medium-sized tree, we randomly selected one of the 135 trees to represent the medium tree.

From the 2008 trees containing only a single node, we selected one at random, and trivially selected the only commit in the tree. From the trees containing seven non-root nodes, we selected one tree randomly, then randomly selected one of the commits in the tree. We selected commit a3c1239eb59c0a907f8be5587d42e950f44543f8 from the tree containing a single node, which we will refer to as *Commit 1*, and commit cdbdd1676a5379f1d5cbd4d476f5e349f445befe from the tree containing seven nodes, which we will refer to as *Commit 2*. A visual representation for these trees is shown in Figure 13.

Table 3: Null and Alternative Hypotheses for Summarization Tasks

Task	Metric	Null Hypothesis	Alternative Hypothesis
T4	Commit	There is no difference on accuracy or timing between *Commit 1* and *Commit 2*	There is a difference on accuracy or timing between *Commit 1* and *Commit 2*
T4	Correctness	Linvis does not effect the correctness of the participants	Linvis does effect the correctness of the participants
T4	Accuracy	There is no difference between Linvis and Gitk on accuracy	There is a difference between Linvis and Gitk on accuracy
T4	Timing	There is no difference between Linvis and Gitk on timing	There is a difference between Linvis and Gitk on timing
T5	Commit	There is no difference on accuracy or timing between *Commit 1* and *Commit 2*	There is a difference on accuracy or timing between *Commit 1* and *Commit 2*
T5	Correctness	Linvis does not effect the correctness of the participants	Linvis does effect the correctness of the participants
T5	Accuracy	There is no difference between Linvis and Gitk on accuracy	There is a difference between Linvis and Gitk on accuracy
T5	Timing	There is no difference between Linvis and Gitk on timing	There is a difference between Linvis and Gitk on timing
T6	Commit	There is no difference on accuracy or timing between *Commit 1* and *Commit 2*	There is a difference on accuracy or timing between *Commit 1* and *Commit 2*
T6	Correctness	Linvis does not effect the correctness of the participants	Linvis does effect the correctness of the participants
T6	Accuracy	There is no difference between Linvis and Gitk on accuracy	There is a difference between Linvis and Gitk on accuracy
T6	Timing	There is no difference between Linvis and Gitk on timing	There is a difference between Linvis and Gitk on timing
T7	Commit	There is no difference on accuracy or timing between *Commit 1* and *Commit 2*	There is a difference on accuracy or timing between *Commit 1* and *Commit 2*
T7	Correctness	Linvis does not effect the correctness of the participants	Linvis does effect the correctness of the participants
T7	Accuracy	There is no difference between Linvis and Gitk on accuracy	There is a difference between Linvis and Gitk on accuracy
T7	Timing	There is no difference between Linvis and Gitk on timing	There is a difference between Linvis and Gitk on timing
T8	Commit	There is no difference on accuracy or timing between *Commit 1* and *Commit 2*	There is a difference on accuracy or timing between *Commit 1* and *Commit 2*
T8	Correctness	Linvis does not effect the correctness of the participants	Linvis does effect the correctness of the participants
T8	Accuracy	There is no difference between Linvis and Gitk on accuracy	There is a difference between Linvis and Gitk on accuracy
T8	Timing	There is no difference between Linvis and Gitk on timing	There is a difference between Linvis and Gitk on timing
T9	Commit	There is no difference on accuracy or timing between *Commit 1* and *Commit 2*	There is a difference on accuracy or timing between *Commit 1* and *Commit 2*
T9	Correctness	Linvis does not effect the correctness of the participants	Linvis does effect the correctness of the participants
T9	Accuracy	There is no difference between Linvis and Gitk on accuracy	There is a difference between Linvis and Gitk on accuracy
T9	Timing	There is no difference between Linvis and Gitk on timing	There is a difference between Linvis and Gitk on timing
T10	Commit	There is no difference on accuracy or timing between *Commit 1* and *Commit 2*	There is a difference on accuracy or timing between *Commit 1* and *Commit 2*
T10	Correctness	Linvis does not effect the correctness of the participants	Linvis does effect the correctness of the participants
T10	Accuracy	There is no difference between Linvis and Gitk on accuracy	There is a difference between Linvis and Gitk on accuracy
T10	Timing	There is no difference between Linvis and Gitk on timing	There is a difference between Linvis and Gitk on timing

Table 4: User-Opinion Questions

Task	Description
T11	Given these tasks again, which tool would you prefer?
T12	Which aspects of each tool did you like and why?

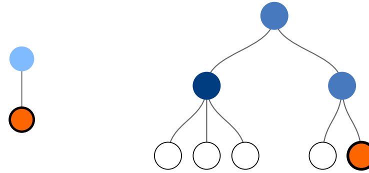

Figure 13: The Merge-Trees used in the user study, containing one and seven non-root nodes, respectively.

7.3 Participant Profile

The study was conducted with 12 participants, all of whom were masters, PhD, or post-doc researchers in the field of software engineering. The participants had between 6 months and 10 years experience with git, with the median being 3.5 years. Most participants had additional experience with SVN and CVS. One of the participants in the study worked as a release engineer, studying merge practices to determine the best way to merge branches while minimizing the number of merge conflicts, on SVN repositories. The participants worked with repositories ranging from around 10 commits up to 38000 commits, with the median being 350 commits. Two of the participants had never collaborated with anyone in a repository, while the rest had some experience with repositories being modified by multiple people, with the most being 219. The median number of collaborators was four. Participants had most experience with personal and academic repositories. Three of the twelve participants had experience with professional repositories.

All participants have had at least some experience with version control, branching in repositories, and git. The participants are from the same lab, and each participant worked with both tools in the study, thus, keeping the sample populations identical for both tools, with some variation between participants in experience with repositories.

7.4 Study Procedure

The study begins with a short introduction to the Merge-Tree model, followed by an introduction to Gitk and Linvis. We provided two examples of converting the DAG to the Merge-Tree. The first example DAG was a flat merge, so the corresponding Merge-Tree did not have any internal nodes. The second was not a flat merge, so the corresponding Merge-Tree contained a single internal node. Any questions about the model were answered at this time. Following the introduction to the Merge-Tree model, we introduce Gitk and Linvis, providing information about what each pane shows in Gitk, and what each tab does in Linvis.

The conceptual tasks follow the introduction, followed by the summarization, which are then followed by the user opinion. The study is concluded with three questions to enable us to learn more about the participants;

- How long have you used git?
- For what kind of projects have you used git?
- How many commits, files, and collaborators were involved with the largest repository you have worked with?

To mitigate the order bias cased by using one tool before the other or one commit before the other, we randomize both the order that the tools are used, and the order that the commits are used, between participants. The conceptual tasks are run twice, once with *Commit 1* and once with *Commit 2*, in the same order for all participants using the Gitk.

The summarization tasks are run four times, once with *Commit 1* and Linvis, *Commit 2* and Linvis, *Commit 1* and git tools, and *Commit 2* and Gitk. The participant will start with one tool, and complete the tasks for both commits in that tool. The order of the commits is the same as in the conceptual tasks. Once they complete the summarization tasks for both *Commit 1* and *Commit 2*, they will switch to the other tool and complete the tasks again on the commits in the same order. The order of the tasks is consistent over all four runs. The questions in the user opinions section do not directly involve the tools or commits.

We use a script written in python 3.6.1 in order to assure that we do not bias the randomization, and ensure that correct ordering is maintained between tools, tasks, and commits. The script produces the entire script for the study, so we only need to read it to correctly conduct the study. Re-running the script generates the study for the next participant. We use screen capture to record the audio and video. From the screen capture, we measure the time, as well as recording the responses in the study.

7.5 Results

7.5.1 Conceptual Tasks

Table 5 shows a summary of the results from conceptual tasks T2 and T3. The answer column contains the correct answer to the corresponding task, while the median, mean, and variance columns are with respect to the answers provided by the participants. The median(s), mean(s), and variance(s) columns are with respect to the time taken, in seconds, by the participants to respond to the tasks.

The results from T1 did not vary greatly between the participants. The drawings generally show something in the form of a list, starting from the provided commit and traversing the master branch, indicating that the participants were unable to determine which merge was in the master branch. Many of the participants identified the next tag, v2.6.29-rc6, to be “the master branch”, however, a tag is not a branch. One participant was able to draw the correct

diagram for *Commit 1* directly from the git command line, but rejected their response after inspecting the DAG visualization provided by Gitk and proceeded to draw a list of merges.

Table 5: Results from the conceptual questions

Question	Commit	Answer	Median	Mean	Variance	Median(s)	Mean(s)	Variance(s)
T2	*Commit 1*	1	4	19.11	753.11	10.0	49.92	5952.08
T3	*Commit 1*	1	5	8.27	53.62	7.5	24.67	884.42
T2	*Commit 2*	5	4	7.80	136.84	31.5	106.83	54123.42
T3	*Commit 2*	3	3.5	5.40	50.27	11.0	65.6	29798.82

Users were able to more closely estimate the number of commits and merges in the larger tree, but generally took longer than the smaller tree. The tree with a single node resulted in more variability in the estimate of number of commits. It should be noted that these questions were answered after spending roughly ten minutes attempting to draw a picture that held the answers to these questions, so the times indicate how quickly the participant was able to interpret their conceptual understanding.

7.5.2 Summarization Tasks

Table 6 shows whether the Merge-Tree size has an effect on the three studied metrics. An interesting observation from the figure is that the correctness for T5 and T9 are effected by the size of the Merge-Tree, but the accuracy is not. The correctness results for T5 and T9 are shown in Figure 15, and the timing results for T7 are shown in Figure 16. Figure 14 shows the combined results from both commits for the other tasks.

Summaries for correctness, accuracy, and timing are shown in tables 7. The full results are as follows:

- T4: What is the series of merges involved with merging this commit?

  Correctness: The χ^2 statistic is 13.07 with a p-value of 0.0003. $13.07 > 3.841$ and $0.0003 < 0.05$, therefore we reject the null hypothesis.

		Linvis	
		Correct	Incorrect
Gitk	Correct	6	0
	Incorrect	15	2

  Linvis improves the correctness when determine the series of merges involved with merging a commit.

  Accuracy: The p-value is 7.8×10^{-5}, which is less than 0.05; therefore we reject the null hypothesis. The Delta estimate is -0.61, which indicates a large effect size. Linvis improves accuracy when determining the series of merges involved with merging a commit.

Table 6: Cross-Commit Metrics: Showing that tree size make a difference in the correctness metric for T5 and T9, and the timing metric for task T7.

Task	Metric	p-value	Conclusion
T4	Correctness	0.22	Do not reject H_0
T5	Correctness	0.04	Reject H_0
T6	Correctness	0.13	Do not reject H_0
T7	Correctness	0.06	Do not reject H_0
T8	Correctness	0.07	Do not reject H_0
T9	Correctness	0.04	Reject H_0
T10	Correctness	0.62	Do not reject H_0
T4	Accuracy	0.94	Do not reject H_0
T5	Accuracy	0.09	Do not reject H_0
T6	Accuracy	0.19	Do not reject H_0
T8	Accuracy	0.16	Do not reject H_0
T9	Accuracy	0.08	Do not reject H_0
T10	Accuracy	0.37	Do not reject H_0
T4	Time	0.77	Do not reject H_0
T5	Time	0.97	Do not reject H_0
T6	Time	0.90	Do not reject H_0
T7	Time	0.01	Reject H_0
T8	Time	0.87	Do not reject H_0
T9	Time	0.99	Do not reject H_0
T10	Time	0.68	Do not reject H_0

Timing: The p-value is 0.0007, which is less than 0.05; therefore we reject the null hypothesis. The delta estimate is -0.58, which indicates a large effect size. Linvis decreases the time taken to determine the series of merges involved with merging a commit.

Overall, Linvis is able to help users determine the series of merges that a commit takes more quickly and more accurately than Gitk.

- T5: What other commits are merged with this commit?

  Correctness: The correctness results are split for this task. For *Commit 1*, the χ^2 statistic is 5.14 with a $p - value$ of 0.02334. 5.14 > 3.841 and 0.02334 < 0.05, therefore we reject the null hypothesis.

		Linvis	
		Correct	Incorrect
Gitk	Correct	4	0
	Incorrect	7	0

The χ^2 statistic for *Commit 2* is 5.14 with a p-value of 0.02334. 4.14 > 3.841 and 0.02334 < 0.05, therefore we reject the null hypothesis.

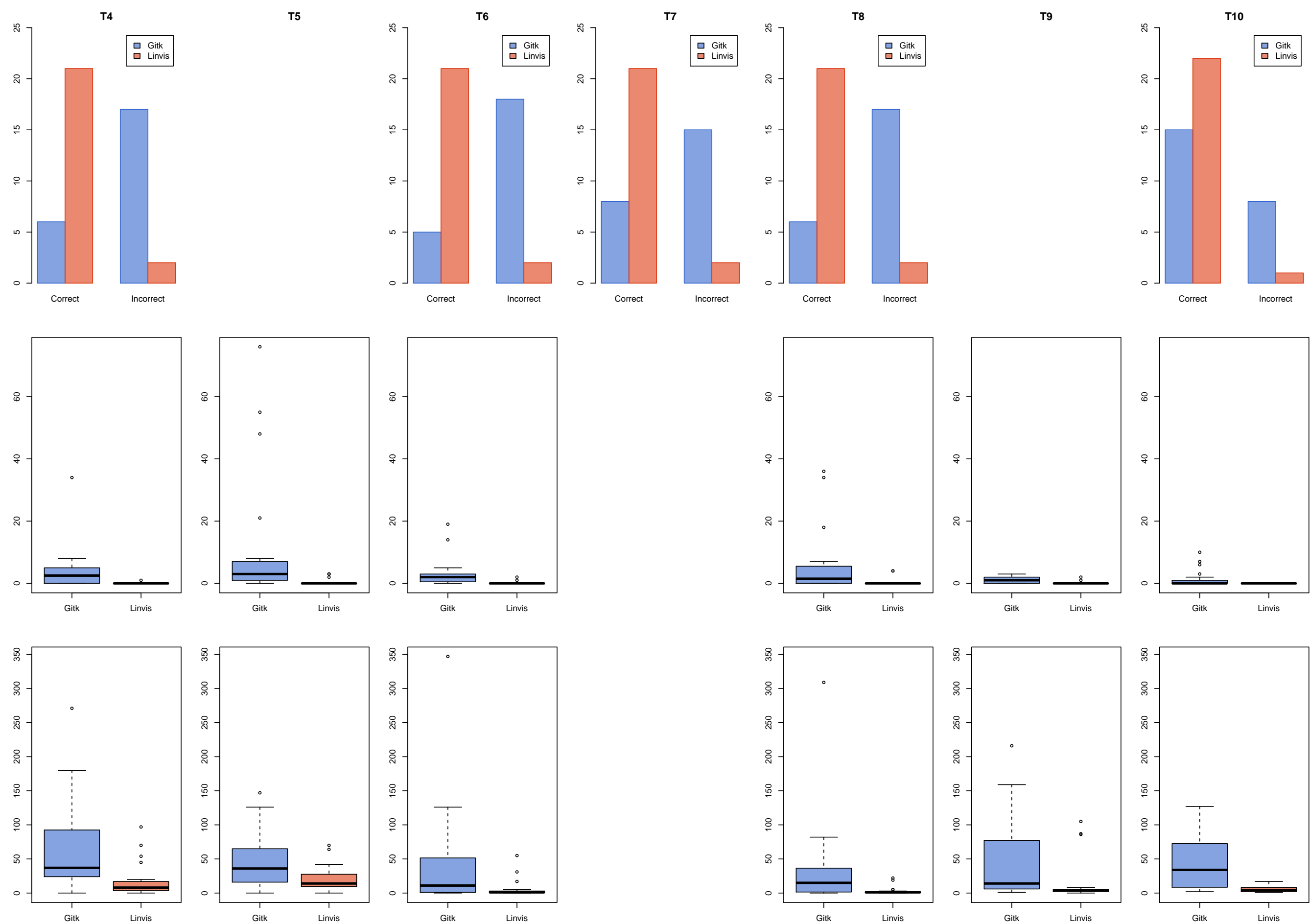

Figure 14: Aggregated results from the summarization tasks. The first row plots the correctness metric for each task. The second row plots the accuracy metric for each task. The third row plots the time metric for each task. The correctness results for T5, T9, and the timing results for T7 cannot be aggregated across commits, thus are omitted.

		Linvis	
		Correct	Incorrect
Gitk	Correct	1	0
	Incorrect	7	4

While there is a difference in correctness between different-sized Merge-Trees, Linvis improves correctness of determining the other commits related to a commit in trees of varying sizes. The difference in the two commits stem from the participants that did not change outcomes between the tools.

Accuracy: The p-value is 7.2×10^{-5}, which is less than 0.05; therefore we reject the null hypothesis. The Delta estimate is -0.63, which indicates a large effect size. Linvis improves accuracy when determining other commits related to a commit.

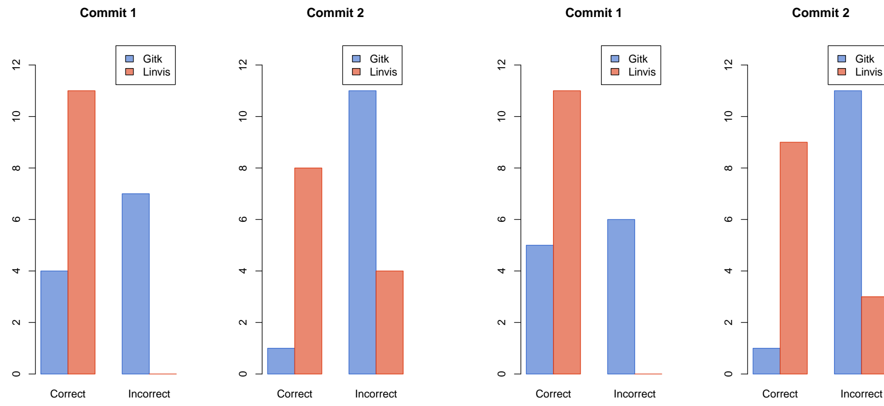

Figure 15: Unaggregated Correctness results for T5 and T9.

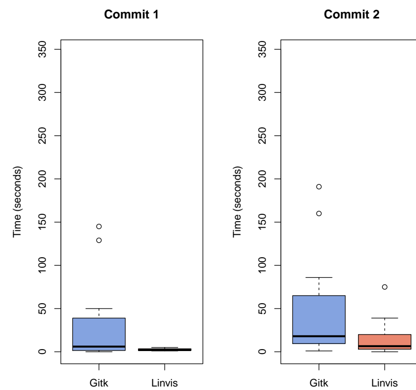

Figure 16: Unaggregated Timing results for T7

Timing: The p-value is 0.0142, which is less than 0.05; therefore we reject the null hypothesis. The delta estimate is -0.58, which indicates a large effect size. Linvis decreases the time taken to determine other commits related to a commit.

Overall, Linvis helps users determine other commits that are merged with another commit more quickly and more accurately.

- T6: How many authors are involved in this merge?

Correctness: The χ^2 statistic is 14.06 with a p-value of 0.0002. $14.06 > 3.841$ and $0.0002 < 0.05$, therefore we reject the null hypothesis.

Table 7: Summarized Correctness, Accuracy, and Timing results for the summarization tasks. In all cases except for the timing metric of T7, there was a difference between Linvis and Gitk.

Correctness

Task	χ^2	p-value	Conclusion
T4	13.07	0.0003	Reject H_0
T5	5.14	0.0233	Reject H_0
	5.14	0.0233	Reject H_0
T6	14.06	0.0002	Reject H_0
T7	11.07	0.0009	Reject H_0
T8	13.07	0.0003	Reject H_0
T9	4.17	0.0412	Reject H_0
	6.13	0.0133	Reject H_0
T10	4.00	0.0455	Reject H_0

Accuracy

Task	p-value	Delta Est.	Conclusion
T4	7.8×10^{-5}	-0.61	Reject H_0
T5	7.2×10^{-5}	-0.63	Reject H_0
T6	1.2×10^{-5}	-0.69	Reject H_0
T8	5.8×10^{-4}	-0.54	Reject H_0
T9	0.009	-0.42	Reject H_0
T10	0.002	-0.35	Reject H_0

Timing

Task	p-value	Delta Est.	Conclusion
T4	0.0007	-0.58	Reject H_0
T5	0.0142	-0.42	Reject H_0
T6	0.0168	-0.41	Reject H_0
T7	0.2586	-0.29	Do not reject H_0
	0.0780	-0.43	Do not reject H_0
T8	0.0009	-0.57	Reject H_0
T9	0.0002	-0.63	Reject H_0
T10	0.0001	-0.66	Reject H_0

		Linvis	
		Correct	Incorrect
Gitk	Correct	5	0
	Incorrect	16	2

Linvis improves correctness when determining how many authors are involved with a merge.

Accuracy: The p-value is 1.2×10^{-5}, which is less than 0.05; we reject the null hypothesis. The delta estimate is -0.69, which indicates a large effect size. Linvis improves accuracy when determining how many authors are involved with a merge.

Timing: The p-value is 0.0168, which is less that 0.05; we reject the null hypothesis. The delta estimate is -0.41, which indicates a medium effect size. Linvis decreases the time taken to determine the number of authors involved in a commit.

Overall, Linvis assists users with determining the number of authors involved in a merge more quickly and more accurately.

- T7: Who contributed the most changes to this merge?

  Correctness: The χ^2 statistic is 11.07 with a p-value of 0.0009. $11.07 > 3.841$ and $0.0009 < 0.05$, therefore we reject the null hypothesis.

		Linvis	
		Correct	Incorrect
Gitk	Correct	8	0
	Incorrect	13	2

  Linvis improves the correctness of participants when determining the author that contributed the most changes to a merge.

  Timing: The null hypothesis is not rejected in the case of *Commit 1*, but is rejected in the case of *Commit 2*. This indicates that Linvis is able to help users determine who made the most contributions to a merge in a large tree but not in a small tree.

  Overall, Linvis improves the correctness for trees of various sizes, but only has an effect on the length of time taken when the tree is large.

- T8: How many files were modified in this merge?

  Correctness: The χ^2 statistic is 13.07 with a p-value of 0.0003. $13.07 > 3.841$ and $0.0003 < 0.05$, therefore we reject the null hypothesis.

		Linvis	
		Correct	Incorrect
Gitk	Correct	6	0
	Incorrect	15	2

  Linvis improves the correctness of users when determining the number of files that were modified in a merge.

Accuracy: The p-value is 5.8×10^{-4}, which is less than 0.05; we reject the null hypothesis. The delta estimate is -0.54, indicating a large effect size. Linvis improves accuracy when determining how many files were modified at a merge.

Timing: The p-value is 0.0009, which is less than 0.05; we reject the null hypothesis. The delta estimate is -0.57, indicating a large effect size. Linvis decreases the time taken to determine the number files modified in a merge.

Overall, Linvis assists users to determine the number of files modified in a merge more quickly and accurately.

- T9: Which file(s) had the most changes in this merge?

  Correctness: The correctness results are split for this task. For *Commit 1*, the χ^2 statistic is 4.17 with a p-value of 0.0412. $4.17 > 3.841$ and $0.0412 < 0.05$, therefore we reject the null hypothesis.

		Linvis	
		Correct	Incorrect
Gitk	Correct	5	0
	Incorrect	6	0

  The χ^2 statistic for *Commit 2* is 6.125 with a p-value of 0.0133. $6.125 > 3.841$ and $0.0133 < 0.05$, therefore we reject the null hypothesis.

		Linvis	
		Correct	Incorrect
Gitk	Correct	1	0
	Incorrect	8	3

  While there is a different in correctness between different-sized Merge-Trees, Linvis improves the correctness when determining which files had the most changes in trees of varying sizes. The difference in the two commits stem from the participants that did not change outcomes between tools.

  Accuracy: The p-value is 0.009, which is less than 0.05; we reject the null hypothesis. The delta estimate is -0.42, indicating a medium effect size. Linvis improves accuracy when determining which file had the most changes.

  Timing: We reject the null hypothesis; the delta estimate is -0.63, indicating a large effect size. Linvis decreases the time taken to determine which file had the most changes.

  Overall, Linvis is able to assist users determine which file had the most changes more quickly and accurately.

- T10: Which modules does this Merge-Tree involve?

Correctness: The χ^2 statistic is 4 with a p-value of 0.0455. $4 > 3.841$ and $0.0455 < 0.05$, therefore we reject the null hypothesis. The p-value is very close to the α value in this case, so the conclusion is not as well defined.

		Linvis	
		Correct	Incorrect
Gitk	Correct	14	1
	Incorrect	8	0

Linvis improves correctness when determining the modules involved at a merge.

Accuracy: The null hypothesis is rejected; the delta estimate is -0.35, indicating a medium effect size. Linvis improves accuracy when determining which modules are involved with a merge.

Timing: The null hypothesis is rejected; the delta estimate is -0.66, indicating a large effect size. Linvis decreases the time taken to determine the modules involved with a merge.

Overall, Linvis assists users determine which modules are involved with a merge more quickly and accurately.

To summarize, Linvis is able to assist users with summarizing various aspects of a merge more quickly and accurately than Gitk. In two tasks, T5 and T9, correctness was affected by the size of a tree, but in both trees, Linvis was able to provide a statistically significant improvement to the correctness. Only in one task, T7, was timing affected by the size of a tree; Linvis did not significantly improve the time taken to answer T7 on the smaller tree, but did improve the time taken on the larger tree.

7.5.3 User Opinions

Among the 12 participants, there was nearly unanimous agreement that for conceptual understanding and summarization tasks, Linvis was easier to use than Gitk. The participants cited the ability to abstract information about the merge from the clean summarization tables and simple visualizations as the primary reasons for preferring Linvis to Gitk. Three participants suggested that someone with a professional understanding of Gitk and the git command-line may be able to extract a conceptual understanding from the DAG visualization and perform the summarization tasks. One of these three participants said that they would prefer to have both tools available, as they are able to complement each other.

8 Discussion

The results show conclusive results; the DAG visualization provided by Gitk is unable to provide an explanation of the changed being made in the repository, is not easily understood, and is unable to provide summarizations of the changes being made at a merge. Meanwhile, the Merge-Tree visualization of Linvis is better able to provide

an explanation of how a commit is integrated, along with better summarization. In this section, we discuss some observations, threats to the validity, and comments made by the participants during the study followed by discussion about modifications to the original algorithm to help generalize it for more repositories, concluded with future work.

8.1 Observations

During the course of the study, we noticed an issue that was consistent among the participants when using Gitk; the most difficult challenge was identifying the master branch of the repository. Many participants would identify the tag as being the "master branch", but a tag is not a branch (it simply points to a given commit). The visualization of the DAG does not provide any notion of which branch being shown is the master branch. The default assumption is that the first branch on the left is the master branch, but this is not always correct. Furthermore, the visualization in Gitk is not consistent, the ordering of the branches changes between runs, so identifying the branch once will not guarantee that it will be identifiable after restarting Gitk.

If users were able to easily identify the master branch using the visualizations of the DAG, the results of the study would likely be very different, making the differences between Gitk and Linvis less significant. This would help in the case of the single-commit trees, it is trivial to summarize information about a single item, but identifying that there was only a single commit in the tree was non-trivial. This information is important because more than a quarter of the trees in the Linux repository contain a single commit, and the structure of these trees is identical to the structure of flat trees. Flat trees are the most common branching structure in the repository, with the root being the parent to all of the other nodes in the tree; to improve summarization and comprehension of these trees, it would likely by sufficient to indicate the master branch in the DAG visualization.

While, in general, the participants of the study preferred using Linvis to Gitk, there were some noticeable usability issues in Linvis. The most prominent issue being the navigation within the tree visualizations. The participants expected that the tabs would update when they clicked on a node in the tree, instead of having to click the link to the commit after clicking the node. This was true in both the Reingold-Tilford tree and the Pack tree. The second major usability issue was the inconsistency between the list tree and the other tree visualizations. The list tree is rooted at the repository event that the user is current looking at, while the other tree visualizations show the entire tree, and use bright orange to highlight the current node.

8.2 Comments From a Release Manager

One participant in the study worked as a release manager for more than three years, working with both SVN and CVS repositories, but had little experience with git. This subsection contains improvements proposed by this participant during the study and afterward.

Contributors making merges need to understand more than just what merges a commit was collected into before reaching the repository of the contributor. It is also important to understand order that the related commits were made, as the order tells the story of what the developer was thinking as they were writing the changes. The visualization of the Merge-Tree in Linvis does not order the commits, randomly ordering them in each level. The Merge-Tree model does retain this information, so implementing this modification is trivial. This is the primary reason behind this participant's request to use both tools simultaneously. Linvis is able to help with the aggregation of the information, and provide a better understanding of the next merge involved in integrating this commit, but the DAG visualization in Gitk provides the full story of the commit instead of hiding it behind a layer of abstraction.

The comments from this participant were very insightful, and will assist in improving the model in the future.

8.3 Threats to Validity

Internal Threats:

The repository contains many commits, of which we only chose two. The commits chosen were randomly selected to be representative of those in the repository. Additional commits could be added, but this would include additional time costs during the study.

The answers provided by the participants to earlier tasks were not taken into account for determining correctness or accuracy in further questions. An incorrect answer may impact the results of the tasks that followed. If the participant was unable to determine the correct commits for the tree within the conceptual task group, the summarization results would be recorded as wrong, even if the summarization they provided was correct for the set of commits selected to be part of the tree. A further study could be done where the correct answer was provided to the participant after each task so that errors would not propagate between tasks.

To mitigate the effects of order-bias, the order the tasks were performed, the tools were used, the commits were studied was shuffled between participants. A single participant is affected by the effect, but this should be mitigated by the results from other participants.

We were only able to work with 12 participants. We apply various statistical tests to ensure that our findings are statistically significant.

External Threats:

While git includes Gitk and git log provides a visualization of the DAG with `git log --graph`, many of the participants in our study were unfamiliar with the DAG visualizations shown by these tools. Other tools may provide better summarizations and visualizations, but we are unaware of any. We investigated the gui tools for Linux on the git website[4], but none of the tools, with the exception of Gitk and git log, were able to produce a visualization of the

[4] https://git-scm.com/download/gui/linux

DAG for the Linux repository. Investigating the tools more deeply, it appears that GitKraken is the most popularly cited git client. The visualization provided by GitKraken was centered around the same DAG metaphor used by git log. GitKraken was unable to produce a visualization for the Linux repository on our system. Inspecting these tools on smaller repositories show that the visualizations of the DAG were very similar to the visualization in Gitk.

The participants in the study were students, some with industrial experience. Most of the participants had worked relatively few collaborators, so professional developers may have been able to better comprehend the DAG visualizations in Gitk.

9 Generalizing the Creation of the Merge-Tree

The visualizations of the Merge-Tree model assist users with understanding how commits are integrated. We continue by discussing possible methods for improving the algorithm to directly produce the combined DAG Merge-Tree without a post-processing step and pruning the DAG to improve the performance and make it feasible to compute over the internet. Furthermore, we look to generalize the algorithm to work with arbitrary repositories.

The problem of identifying the master branch is the most difficult, and is not possible without heuristic approaches. The master branch can be confounded by *foxtrot* merges, which change the order of the parents in the parent list of a merge. Other than the ordering of the parent list, git has no other way to determine which commits are on the master branch of the repository. For the purpose of this discussion, we will assume that we can rely on the order of the parent list to be correct, and that *foxtrot*s do not occur. Furthermore, merge information is lost in the event of fast-forward merges, which collapse the branch into the current branch.

The primary difference between the original algorithm and the generalization is the main data structure. Algorithm 1 is centered around the use of the stack, making it akin to a depth-first traversal of the DAG with dynamic programming. The generalization in Algorithm 2 is based on the queue, making it akin to the breadth-first traversal of the DAG. While the final results of the two traversals should not be different, the runtime properties are different.

We will begin with a simplified description of the original algorithm. The algorithm starts at the reference to the master branch. We will use the term master branch loosely here, meaning any branch that has been labeled in some way as being the main branch, this branch may have a different name, but standard naming conventions call it the master branch. Starting from the master branch label, we traverse the entire master branch to the initial commit, recording the commit hash of every repository event encountered.

The second phase iterates over every repository event, computing the shortest distance to the master branch. For this reason, the entire repository must be available to the algorithm. This is a dynamic programming solution, centered around the call stack. The next node on the shortest path to the master branch will become the parent of this node in the tree.

If this algorithm is to be generalized to work in a dynamic, web-based, tool it may not be feasible to download every every repository event in the repository in a reasonable period of time. We instead look toward a breadth-first approach at limiting the number of repository events that must be downloaded.

Focusing on the generalized algorithm (Algorithm 2), the algorithm is broken into two phases. The first phase is where the improvements take place. The first phase has three main goals; the first is to track the master branch; the second goal labels the depth of each repository event that is traverses, or how many merges the event is from being merged; the third goal is to record the children of each repository event. The first phase also begins the process of reversing the DAG, by providing the candidate events that could possibly be the real parent to this node. The first phase can terminate early, if it reaches a point where all branches that it is following terminate. The $openBranches$ metric on line 7 indicates how many branches are currently being traversed. In line 11 through 15, we decrement the $openBranches$, as this is the point where a branch occurs. The second branch closing clause is necessary to handle the case when the depth was incorrectly assigned, which can occur if there are more events on the current branch than on the side branch.

Table 8: Example traversal of the example DAG in Figure 4 starting at merge 11.

Current	Parents	Open Branches	Q	Visited	Children
-	-	0	11	11:0	
11	4,9	1	4,9	4:0	11, 5
4	1	1	9,1	9:1	11
9	7,6,8	3	1,7,6,8	7:1	9, 8
1		3	7,6,8	6:2	9
7	5	3	6,8,5	8:3	9
6	5	2	8,5	5:1	7, 6
8	7	1	5	2:1	5, 3
5	4,2,3	2	2,3	3:2	5
2	1	1	3		
3	2	0			

In Table 8, we follow the algorithm starting at merge 11 from the example in Figure 4, recording the current node, the parents of the current node, the open branches, and the queue structure after updating. The visited column is not dependant on the current step, but keeps track of the nodes that have been visited, the depth, and the children of the corresponding node. When we reverse the DAG, the children in this table will become the candidates for being the parent of the corresponding node. For example, the parent of node 7 may be resolved to either node 9 or node 8. In this case, the shortest path to the master branch is through node 9.

The terminology surrounding the parent/child relationship gets very confusing at the change between phase one and two because this is where the roles switch. In phase 1, we are working with the DAG; in phase 2, we are working with a structure that is neither a tree nor a DAG, but can only be described as a directed graph.

The goal of the second phase is to resolve the tree from the directed graph. We will refer to parents in the same sense as we would refer to them in the tree form of the repository for the remainder of this section. We have gathered the depths and possible parents (line 17), but we need to remove the extra parents. Extra parents are culled in line 42 of the algorithm. Of the possible parents, we take the parent that is closest to the root, but not beyond. This forces the real parent to either be deeper in the tree, or on the same branch as the current node. Line 30 to 37 are for bootstrapping the queue, ensuring that we are not starting on an empty queue and is nearly identical to lines 41 to 48.

The construction of the tree becomes trivial now that we are able to determine the parents and children for each node. Lines 44 and 45 perform this construction, adding the DAG parent to the tree children of the current node, and the DAG parent's parent becomes the current node, which was the DAG parent's child. We continue following the path until we hit the root node, merging the commit into the master branch.

This algorithm relies heavily on the order of parents in the parent list, but is able to trim the number of nodes that must be downloaded. Furthermore, the resulting tree is able to show the order of commits, telling the story of how a developer was working, before the set of commits was merged. We developed a small Bitbucket plugin to test the design and determine whether it was capable of producing Merge-Trees, shown in Figure 17.

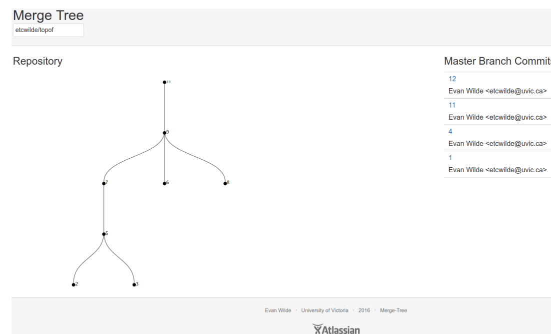

Figure 17: Screen shot of the Bitbucket plugin showing the online tree computation of the repository shown in 4.

10 Conclusion

In this paper, we describe a tree-based model, Merge-Tree, designed to provide git users with a better explanation of the events occurring in a git repository. The Merge-Tree model is rooted at the merge into the master branch of a repository, where the leaves are the individual commits.

We implement a tool, Linvis, using the Merge-Tree model, which allows a user to filter the commits with a simple search interface. Once the desired commit is found, Linvis provides tabs for simple aggregated tables showing the files that were modified, the authors that made the modifications, among other information. Linvis also provides three visualizations, list tree, pack tree, and a Reingold-Tilford tree view. The list tree enables easy searching for textual features, while the pack tree is good for visualizing the structure of large Merge-Trees, and the Reingold-Tilford tree is better at visualizing the structure of small Merge-Trees.

To ensure that our model and implementation are useful, we evaluated the tool with a three-part user study. We tested the conceptual understanding using the Gitk, ability to summarize merges into the master branch, and ask for the opinions of the participants. We found that the participants were unable to determine the commits and merges that are related to the commits we tested with and build a conceptual understanding from the DAG visualization. The correctness, accuracy, and timing of summarization tasks were improved using Linvis compared to Gitk. The participants generally enjoyed the clean summarizations and simple visualizations provided by the Merge-Tree-based visualizations in Linvis.

Given that visualization of the model provides users with a better understanding of events in the repository, we describe some issues with identifying the master branch of the repository and a more generalized approach to constructing Merge-Trees. We implemented the generalized approach as a Bitbucket plugin to verify that it is able to produce visualizations for arbitrary bitbucket repositories. We found that it was able to produce visualizations for many repositories, but was unable to produce the visualizations of others due to rate-limiting. An interesting side-effect of pruning the DAG is that some Merge-Trees in the repository can be visualized, while other trees are too large to download the commit information for. The original algorithm design would be unable to produce a visualization for any of the Merge-Trees.

Merge-Trees are a novel means of building a model which can be visualized and summarized in an effective way. Participants in our study found visualizations of the Merge-Tree to be more enjoyable for summarization tasks than the visualizations of the DAG. While there are some implementation issues with Linvis, Merge-Tree visualizations are an effective means of providing a clear conceptual understanding and simple summarization of the events being merged into the master branch of a repository.

References

1. Begel Andrew, Khoo Yit Phang, Zimmermann Thomas. (). *Codebook: discovering and exploiting relationships in software repositories*, Proceedings of the 32nd ACM/IEEE international conference on software engineering - volume 1, ICSE 2010, cape town, south africa, 1-8 may 2010, pp. 125–134.

2. Burch Michael, Diehl Stephan, Weißgerber Peter. (). *Eposee - A tool for visualizing software evolution*, Proceedings of the 3rd international workshop on visualizing software for understanding and analysis, VISSOFT 2005, budapest, hungary, september 25, 2005, pp. 127–128.

3. Caudwell Andrew H. (). *Gource: visualizing software version control history*, Companion to the 25th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, SPLASH/OOPSLA 2010, october 17-21, 2010, reno/tahoe, nevada, USA, pp. 73–74.

4. Cliff Norman. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin.* 1993;114(3):494–509.

5. Cubranic Davor, Murphy Gail C., Singer Janice, Booth Kellogg S. Hipikat: A project memory for software development. *IEEE Trans. Software Eng.* 2005;31(6):446–465.

6. D'Ambros M., Lanza M., Gall H. (). *Fractal figures: Visualizing development effort for cvs entities*, 3rd ieee international workshop on visualizing software for understanding and analysis, pp. 1–6.

7. D'Ambros M., Lanza M., Lungu M. Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering.* 2009Sept;35(5):720–735.

8. Eick S. G., Graves T. L., Karr A. F., Mockus A., Schuster P. Visualizing software changes. *IEEE Transactions on Software Engineering.* 2002Apr;28(4):396–412.

9. German Daniel M, Adams Bram, Hassan Ahmed E. Continuously mining distributed version control systems: an empirical study of how linux uses git. *Empirical Software Engineering.* 2015:1–40 (English).

10. McNemar Quinn. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika.* 1947Jun;12(2):153–157.

11. Ogawa M., Ma K. L. code_swarm: A design study in organic software visualization. *IEEE Transactions on Visualization and Computer Graphics.* 2009Nov;15(6):1097–1104.

12. Reingold Edward M., Tilford John S. Tidier drawings of trees. *IEEE Trans. Software Eng.* 1981;7(2):223–228.

13. Wang Weixin, Wang Hui, Dai Guozhong, Wang Hongan. (). *Visualization of large hierarchical data by circle packing*, Proceedings of the sigchi conference on human factors in computing systems, pp. 517–520.

14. Wilcoxon Frank. Individual comparisons by ranking methods. *Biometrics Bulletin.* 1945;1(6):80–83.

Algorithm 1 Computing the Merge-Tree of Linux from the DAG

```
1:  function ComputeMergeTree(DAG): tree
2:      head ← Head of master of git repository
3:      master ← traverse DAG from head using
4:              first parent until reaching root
5:      nodes(Tree) ← nodes(DAG)
6:
7:      function MergeAtMaster(cid)
8:          # Returns (depth, merge, next)
9:          # Helper function
10:         # Compute the closest merge into master,
11:         # setting the children on the way to master.
12:         if cid in master then
13:             return (0, cid, ∅)
14:         end if
15:         d ← ∞
16:         # Traverse the inverted DAG
17:         for c ∈ children(cid, DAG) do
18:             (d_c, merge_c, next_c) ← MergeAtMaster(c)
19:             if IsMerge(c) then
20:                 fp ← FindFirstParent(c)
21:                 if fp ≠ cid then
22:                     d_c ← d_c + 1
23:                     next_c ← c
24:                 end if
25:             end if
26:             # Find the shortest path
27:             if d_c < d then
28:                 (d, m, next) ← (d_c, merge_c, next_c)
29:             else if d_c = d then
30:                 # Use the time as a tie-breaker
31:                 if cTime(merge_c) < cTime(m) then
32:                     (m, next) ← (merge_c, next_c)
33:                 end if
34:             end if
35:         end for
36:         # c is the commit that follows cid
37:         # in its way to master
38:         add edge (cid, next) to Tree
39:         return (d, m, next)
40:     end function
41:     # Compute the distance for each commit
42:     # discarding result
43:     for c ∈ nodes(DAG) do
44:         MergeAtMaster(c)
45:     end for
46:     return Tree
47: end function
```

Algorithm 2 Computing the generalized Merge Tree

```
 1: function Phase 1(initial commit root) : updated tree
 2:     openBranches ← 0
 3:     Q ← root
 4:     do
 5:         cur ← Q.dequeue
 6:         parents ← cur.parents
 7:         openBranches ← openBranches + parents.length − 1
 8:         for index, parent ∈ parents do
 9:             if parent.depth is undefined then
10:                 parent.depth ← cur.depth + index
11:             else if cur.depth > parent.depth then
12:                 openBranches ← openBranches − 1
13:             else if cur.depth < parent.depth then
14:                 openBranches ← openBranches − 1
15:                 parent.depth ← cur.depth
16:             end if
17:             parent.children ← cur
18:             if parent has not been visited then
19:                 Q ← parent
20:                 parent.visited ← True
21:             end if
22:         end for
23:     while Q not empty and openBranches ≠ 0
24: end function
25: function Phase 2(initial commit root) : Merge tree
26:     tree.root ← root
27:     Q // Empty Queue
28:     parents ← root.parents
29:     parents ← tail parents
30:     for parent ∈ parents do
31:         realParent ← min(∀child ∈ parent.children, parent.depth >= child.depth)
32:         if realParent is root then
33:             root.children ← parent
34:             parent.parent ← root
35:             Q ← parent
36:         end if
37:     end for
38:     while Q not empty do
39:         cur ← Q.dequeue
40:         parents ← cur.parents
41:         for parent ∈ parents do
42:             realParent ← min(∀child ∈ parent.children, parent.depth >= child.depth)
43:             if realParent is cur then
44:                 cur.children ← parent
45:                 parent.parent ← cur
46:                 Q ← parent
47:             end if
48:         end for
49:     end while
50: end function
```