VisGi: Visualizing Git Branches

Stefan Elsen
Computer Science Department
University of Trier
54286 Trier, Germany
elsen@syssoft.uni-trier.de

***Abstract*—Git repositories quickly become highly complex structures that do not reveal much human-readable information beyond files and folders of active branches.**

This paper introduces VisGi, a tool to abstract and visualize the branch structure of Git repositories, as well as their folder trees. By interpreting branches as groups of aggregated commits, their dependencies are condensed into a directed acyclic graph, and displayed using graph layout strategies. Additionally, Sunburst diagrams are used to display the current content of these branches, the differences between each two branches, as well as the evolution along any singular selected path through the repository.

***Index Terms*—git; versioning; branching; visualization; CVS;**

I. Introduction

Typically used to synchronize concurrent file changes in the context of software development, revision control systems are complex file version archives, that allow to revert to any previously recorded state. Additionally, they enable branching, the creation of a temporarily or permanently isolated state, in order to develop independently of the processes in other branches. Merging, finally allows to reintegrate such branches.

Introduced in 2005, Git [1] has since established itself as one of the major revision control systems [2]. Internally based on a hash-based directed acyclic graph, Git allows sophisticated branching and merging.

While simple in their basic design, the composition of Git commits and branches allows repositories to quickly grow into large and complex aggregates. With the exception of the file and folder contents of a small number of well-managed and frequently updated branches, the history of these repositories often degenerates into write-only archives with no macroscopic, human-readable composition and no or little value to the current development process.

If extracted and visualized properly, however, it can turn into a wealth of information, and a useful tool for comprehending the dynamics of any old and rich source code repository. This becomes especially meaningful whenever repositories house more than one distinct version of a single product, or even multiple products, as each will leave different traces and references in the repository. Beyond the scope of concurrent development, these traces are often diminishing in their visibility, and past branches may simply be forgotten among the increasing amount of commits.

The visualization of a repository's history can be crucial to the integration process of new developers, company management, but also repository maintenance. Generally speaking, by increasing the comprehensibility of the multiple current states of a repository, it allows for more informed decisions.

Existing research has contributed various techniques to visualize linear repository evolution, focusing on structure (e.g. [9]), classes (e.g. [10]), code-lines (e.g. [12]), authors (e.g. [13], [18]), programs (e.g. [14]), or files (e.g. [13], [15]). Other techniques have been developed to also incorporate branching, such as VRCS ([16]), or Visugit ([17]) but scale poorly as repositories grow in size and complexity.

Thus a new visualization paradigm for large repositories with complex branching is proposed and implemented in the VisGi tool prototype. It attempts to extract a branch-based, coarse representation of the repository history instead of its inherent commit graph. By selectively reducing the granularity of repositories in this way, traditional layout and display mechanisms again become applicable and meaningful for most repositories.

The usefulness of this abstraction is demonstrated by visualizing a number of large-scale and openly accessible Git repositories. The benefits of integrating interactive visualizations of the current folder structures are also discussed, as well as those of deltas between any two branches and time slices along paths through the repository.

II. Graph Construction

VisGi extracts a visual Graph from a complex Git repository by aggregating the raw commit graph into branches. This is done by extracting the Git branch pointers, and walking backward through the commit graph from each branch entry point. Branch influence is determined by flagging all commits that can be reached from a certain branch entry point. Commit areas that are flagged by multiple branches are considered branch intersections, whose actual branch parentage cannot easily be determined, and thus treated as separate nodes in the branch graph. All nodes in the graph are called *groups* from this point on.

By definition, the maximum possible number of groups determined this way equals the number of subsets of all branches. Thus $2^N - 1$, where N is the number of branches (excluding the empty set). In practice, however, the number of groups often ranges around twice the number of branches.

978-1-4799-1457-9/13/$31.00

A. Layout

In order to visualize the resulting group graph, a simple layer-based two-dimensional layout was chosen, using Sugiyama-style graph drawing [11]. This particular drawing style was chosen for its simple layered architecture, which allows information to be encoded in the layer that an item is placed on. While these layers are typically arranged top-down, a 90 degrees rotation is applied later to maximize the use of the available space.

Typically, when placing nodes of an arbitrary graph in a Sugiyama-style graph, the length of the longest path to a given node is used as its layer. In the case of Git repositories, however, a time-based layout is more meaningful, since the full concurrency of branches and their history becomes much clearer.

In order to place branch nodes, they are first distributed across an initial set of layers, whose size is based on the number of nodes in the graph. This layer set spans the entire time window of the repository, with the time stamp of the newest commit in each group being used to determine its layer. A correction pass then makes sure that no two connected nodes are placed on the same layer. During this pass, additional layers may be added.

The temporal placement of a node is thus an approximation and not a precise representation of time. Typically the correction pass will create wider layouts, the more dependent branches exist in close temporal proximity.

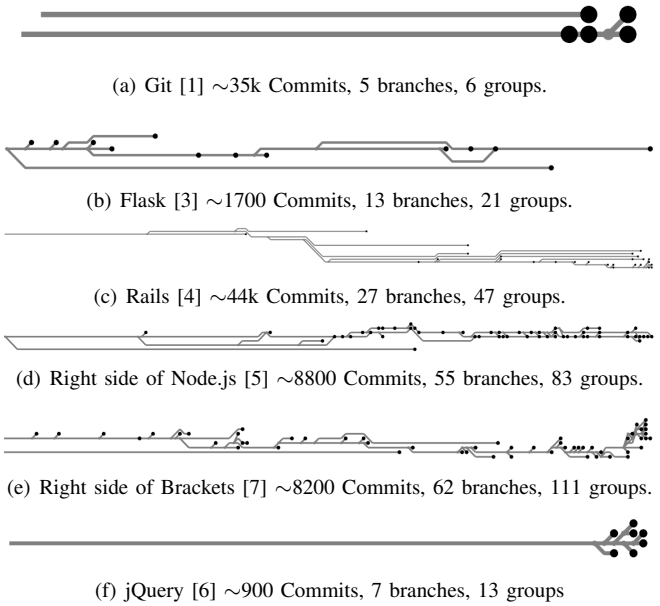
Finally, a layout optimizer attempts to minimize edge crossings as well as jagged lines. This is done by applying a greedy algorithm to a global error value that is sensitive to both non-straight lines and edge crossings.

B. Display

The final graph is rendered using a horizontal layout, with time extending from left to right. This helps maximize the usage of the generally higher horizontal screen resolution of modern screens. During rendering, branches are displayed as solid black dots, while groups resulting from branch intersections are displayed as dark gray dots, to make them visually distinctive. Additional lines are extended to the left from all group nodes that do not have parents but a significant age difference between their oldest and their newest commit. The left end of these lines represents the time of the very first commit.

III. Comparison of Repositories

In order to visualize their data, multiple large-scale and openly accessible Git repositories were mined. The visual output reveals considerable differences regarding the utilization of branches. While some of the repositories (e.g. Git itself) use very few branches, others (such as Brackets) place many, thus considerably increasing the complexity of the visualization. The analysis reveals that there is seemingly no direct correlation between the number of branches and the number of commits.



(a) Git [1] ∼35k Commits, 5 branches, 6 groups.

(b) Flask [3] ∼1700 Commits, 13 branches, 21 groups.

(c) Rails [4] ∼44k Commits, 27 branches, 47 groups.

(d) Right side of Node.js [5] ∼8800 Commits, 55 branches, 83 groups.

(e) Right side of Brackets [7] ∼8200 Commits, 62 branches, 111 groups.

(f) jQuery [6] ∼900 Commits, 7 branches, 13 groups

Fig. 1. Visualization of Git-repositories of various sizes.

IV. Visualizing Group Contents

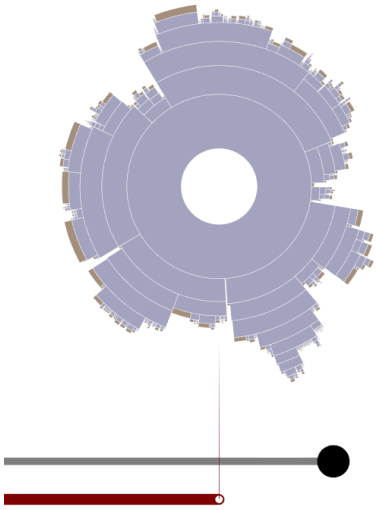


Fig. 2. Sunburst diagram showing the folder tree of the selected branch.

In addition to the pure graph layout, it is useful to also display the content of the displayed groups. Without actively parsing file data, however, the visualizable content is limited to the active folder-tree of any one commit (or all commits) in a group.

Showing all groups at once overloads the available display space, making any two dimensional visualization cluttered and uninformative. The use of an interactive model is important for clean and focused visualizations.

VisGi renders a Sunburst Tree Layout [8] of a group's most up-to-date folder tree just above any selected group node,

as shown in Fig. 2 . Selecting a different node replaces the previously visible sunburst diagram.

The sunburst diagram construction implements the original design for the most part. Some modifications include:

- Angular and radial gaps have been added between folders.
- The same angular and radial size is assigned to all files. Diverging sizes result in distortion and fragmentation of the diagram outline.
- Files in the same folder are not separated by angular gaps.

Rather than trying to keep files separate and easily distinguishable, the gap between files was removed for a cleaner overall appearance. Again, an interactive model is chosen to enable differentiation, making individual files and folders selectable. When clicking on a specific file or folder, their

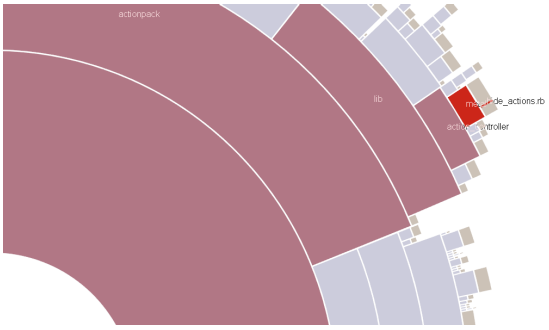


Fig. 3. Highlighted path to the selected file.

paths are highlighted in red, and their respective names printed out on screen as shown in Fig. 3. Where usually the large number of files would make it impossible to display all filenames, doing so for only a few selected files/folders is both clean and intuitive.

V. Visualizing Group Deltas

Using a similar sunburst diagram, deltas between two groups can be visualized by dragging the cursor from one group to another.

A file or folder is inserted into the sunburst diagram if it exists in at least one of the two groups in question. Files with identical hash values are considered unchanged and rendered with lower opacity.

Fig. 4 shows a delta between two groups that have significantly diverged. The intensity of the color of folders reflects the density of changes in this directory. Additionally, files are drawn larger or smaller than the average file, depending on how much they increased or decreased in size.

The visualization of deltas in the described way allows for a quick recognition of significant changes. Renames or larger numbers of changed files in certain folders are easily spotted.

VI. Paths

In addition to the selection of individual groups, VisGi implements a mechanism to select any singular path of connected groups through the active repository.

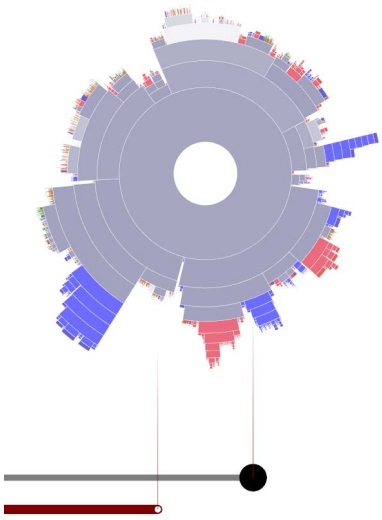


Fig. 4. Delta between two distant groups.

For displaying the final result, the following color code is chosen due to its high contrast and intuitive comprehensibility:

- Red and blue indicate the deletion (red) or creation (blue) of a file/folder.
- Green yellow indicate a significant increase (green) or decrease (yellow) of a file's size.

A. Path Tracing

Whenever the user actively selects a group, as much as possible of the previously selected path (if any) is preserved, while rerouting the path to pass through the newly selected group. Open ends or beginnings are completed arbitrarily such that the path cannot be extended further without changing its route. Coupled with the previously described adaptation mechanism, this allows the iterative selection of any possible path through the repository (see Fig. 5).

B. Path Rendering

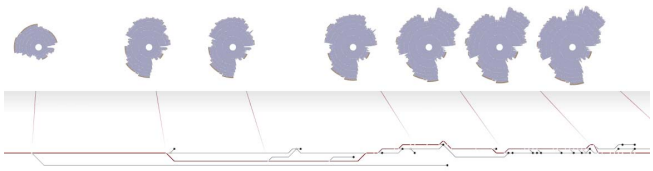


Fig. 6. Sunburst slices along the selected path.

With a sequential subset of connected groups selected, it is now possible to display any history portion of the repository in a linear way. Since folders trees are currently only extracted for the last commit in a group, the level of detail of a selection largely depends on the number of groups along the path.

A subset of all selected groups with a certain minimum temporal distance is chosen to serve as representative frames,

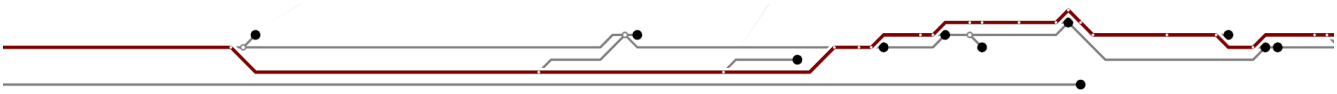


Fig. 5. One of multiple possible paths through the repository.

displayed in a time bar above the main view as shown in Fig. 6. Reference lines indicate the positions of the respective group nodes in the full graph. This allows to intuitively identify the groups that are actually represented in the time bar.

C. Path Frame Interpolation

Since the coarse time bar hides groups in areas of high temporal density, an additional mechanism is needed to restore the removed detail on demand. This is done by adding a time slider that displays the closest known state for the current location of the mouse cursor along the time bar (Fig. 7).

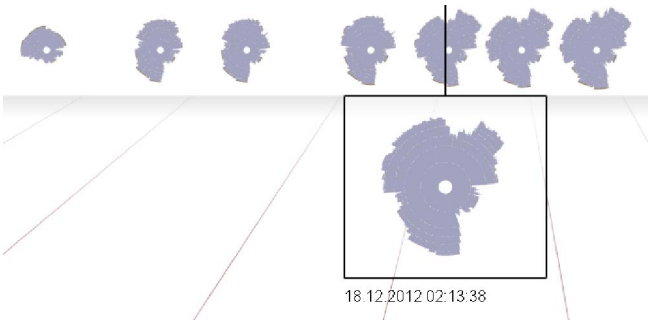


Fig. 7. The selected time window shows a close representation of what the active path looked like at that precise moment.

The two closest groups (the next closest newer, and the next closest older group) are interpolated to produce a smoother representation of the selected moment in time. The precision of this representation currently depends on the temporal resolution of the selected path, thus the general frequency of groups.

VII. Conclusion

This paper introduces the tool prototype VisGi together with an implemented set of strategies for mining and displaying Git repositories in a meaningful way.

Tagging mechanisms are used to aggregate commits into compact nodes of a coarse group graph. Edges in the resulting graph are currently not weighted. Doing so (e.g. by the number of parent relations from one commit group to another) may help improve the quality of the visualization.

Sugiyama-style graph drawing is used to create a clear layout of most repositories. Low numbers of branches (10 or less) do not reveal much about a repository, while large numbers (100+) cause the layout to become cluttered and dysfunctional. Additional filtering techniques may be necessary to resolve this issue.

Sunburst diagrams are used to visualize group contents. While applicable to all types of repositories, they do not allow much insight into their inner structure. Analyzing the contents of contained files may enable better visualizations.

An interactive tracing mechanism is used to allow the user the selection of any path through the group graph. This mechanism is sufficiently intuitive and clean for any of the experimental repository graphs.

A time bar is used to display the currently selected path through the repository. Representative groups are chosen and displayed along the bar. A slider allows to select and display any intermediate state along the time bar. Currently, extraction is limited to the last commit of each group, thus creating holes in older sections of most repositories. Extracting additional trees should remove the holes and create a more representative display of a repository's evolution.

Future work includes extracting file contents and author influences in order to achieve improvements over the current state of the visualization.

References

[1] Git: *http://git-scm.com/*, visited June 3. 2013
[2] Ohloh repository comparison: *https://www.ohloh.net/repositories/compare*, visited June 3. 2013
[3] Flask: *http://flask.pocoo.org/*, visited June 3. 2013
[4] Ruby on Rails: *http://rubyonrails.org/*, visited June 3. 2013
[5] node.js: *http://nodejs.org/*, visited June 3. 2013
[6] jQuery: *http://jquery.com/*, visited June 3. 2013
[7] Brackets: *http://brackets.io/*, visited June 3. 2013
[8] Werner Randelshofer. Tree Visualization: *http://www.randelshofer.ch/treeviz/*, visted June 3. 3013
[9] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A System for Graph-Based Visualization of the Evolution of Software. In *Proceedings of the ACM Symposium on Software Visualization (SoftVis 2003)*, New York, NY, 2003. ACM Press.
[10] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of International Workshop on Principles of Software Evolution IWPSE 2001*, pages 37-42, 2001.
[11] K. Sugiyama, S. Tagawa, M. Toda, Methods for Visual Understanding of Hierarchical System Structures. In *Systems, Man and Cybernetics, IEEE Transactions on (Volume:11 , Issue: 2)*, pages 109-125, 1981
[12] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. CVSScan: Visualization of code evolution. In *Proceedings of ACM Symposium on Software Visualization (SOFTVIS '05)*, pages 47-56, New York, NY, 2005.
[13] Eric Gilbert and Karrie Karahalios. LifeSource: two CVS visualizations. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, pages 791-796, New York, NY, 2006.
[14] Christopher M.B. Taylor and Malcolm Munro. Revision towers. In *Proceedings of the Workshop on Visualizing Software for Understanding and Analysis VISSOFT 2002*, pages 43-50, Washington DC, 2002.
[15] Lucian Voinea and Alexandru Telea. An open framework for CVS repository querying, analysis and visualization. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 33-39, New York, NY, 2006.
[16] Hideki Koike and Hui-Chu Chu. VRCS: Integrating version control and module management using interactive three-dimensional graphics. In *Proceedings of IEEE Symposium on Visual Languages VL'97*, pages 168-173, Washington, DC, 1997.
[17] Visugit: *https://github.com/hozumi/visugit/*, visited August 19. 2013
[18] Gource: *https://code.google.com/p/gource/*, visited August 19. 2013