Commit Graphs

Maximilian Steff and Barbara Russo
Faculty of Computer Science
Free University of Bozen-Bolzano
Bozen, Italy
{first.last}@unibz.it

Abstract—**We present commit graphs, a graph representation of the commit history in version control systems. The graph is structured by commonly changed files between commits. We derive two analysis patterns relating to bug-fixing commits and system modularity.**

I. Problem

One of the basic tenets of software engineering is modularization to break down complexity and enable many developers to work in parallel on the same system without having much overlap in terms of concurrent modifications of files. The manner in which these activities are carried out is reflected in the structure of the history of (sets of) files. Consequently, we can observe a lack of modularization from such histories as well as identify particularly problematic parts of the code base with respect to breakdowns in modularity and the incidence of bug fixes in them. In turn, we can use such information to focus manager and developer attention to improve product quality.

II. Solution

A. Defining Commit Graphs

Commit Graphs (CG) [1] offer a structured representation of commit histories. Each commit is a node in the graph. We add an edge between two nodes when they modify at least one common file without any other commit in between these two commits modifying that file as well. Formally [1], a commit is a tuple (t,$\mathcal{F}$) where t is a timestamp and $\mathcal{F}$ a set of files. A commit graph CG is a directed graph that consists of

- a set of commits {(t,$\mathcal{F}$)},
- a set of links between commits, $\{\rightarrow_{i,j}\}$, such that $(t_1, \mathcal{F}_1) \rightarrow_{1,2} (t_2, \mathcal{F}_2)$ iff $t_1 < t_2, \mathcal{F}_1 \cap \mathcal{F}_2 \neq \emptyset$, and $\forall (t, \mathcal{F})$ with $t_1 < t < t_2 : \mathcal{F}_1 \cap \mathcal{F}_2 \cap \mathcal{F} = \emptyset$.

Figure 1 shows an example over five commits (7,8,12,14,17) and four files (a,b,c,d).

On commit graphs we define root nodes as nodes with outgoing edges, i.e. nodes that have no predecessors. Such commits change no existing files, they exclusively add new files. End nodes are nodes without incoming edges, i.e. nodes without successors. Such commits change files that have not been changed since in the commit histories.

B. Analyzing Commit Graphs

We retrieve commit histories from commit graphs and use correlation to determine properties of these histories. The

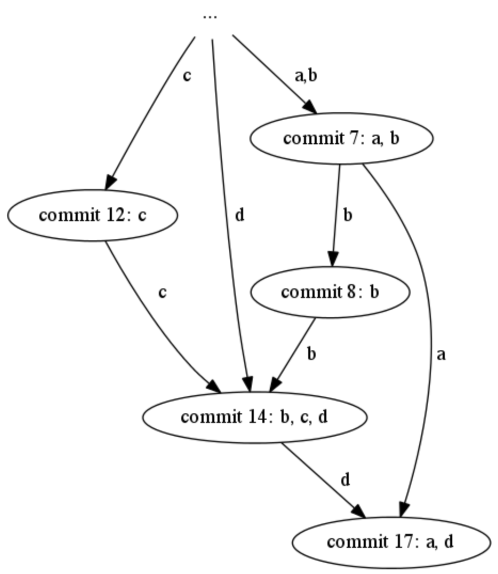

Fig. 1. Commit history example.

retrieval of commit histories is essentially a search in the commit graph, extracting a subgraph according to predefined search rules. Search rules include the selection of nodes at which to start the search, a search scheme, i.e. which edges to follow and which to avoid, and a rule for ending the search, e.g. when there are no edges left to visit. After the search has finished and returned a (set of) path(s) or tree(s), we can augment the returned data structure(s) with additional information, e.g. weights determined by the age of a commit or authorship information. This step may also include the addition of information from secondary sources, e.g. the labeling of nodes from an issue tracker into feature-introducing or bug-fixing commits.

We analyze retrieved histories by correlating two of their attributes to determine certain properties. We obtain attributes by counting their occurrences over the histories and correlate these counts over all retrieved histories. For instance, we may correlate the number of bug fixes collected over the histories of the files of a given namespace or package against time-weighted intervals of these histories to determine the occurrence of bug fixes over the number of commits.

Focusing on topological properties of nodes, we can identify commits that connect different strains of development. A commit that has many preceding commits in the commit graph ties together files that have previously been changed independent of each other. Such changes may indicate breakdowns of modularity.

III. Discussion

Commit graphs are essentially a structured way to examine logical couplings [2] over time. Additionally, they allow for combining logical couplings with process metrics. Logical couplings are dependencies between code entities inferred from the commit history in a version control system (VCS). They may complement code dependencies but not necessarily so. Process metrics are measurements of the data in a VCS, e.g. how many different authors a file has (had) or how many times a given file has been changed.

As an extension to logical couplings, the same caveats apply as for logical couplings. Co-change of files does not necessarily indicate an actual 'coupling', i.e. a dependency between two files. A developer might just have submitted two unrelated changes in a single commit. As we consider single commits, time is an important factor as files are changed differently over time, i.e. files are not changed with the same frequency over time. In turn, if considering long histories, appropriate steps have to be taken to account for changes in frequency if necessary. For instance, depending on the properties of interest in the analysis, two-year old commits may be of less importance than the commits of the past week. Weighting histories helps solve this issue.

Commit graphs provide a natural way for structuring process metrics. Process metrics, i.e. process information gathered from a VCS, can be added to a commit graph's nodes and edges. Foremost, we can augment nodes with the metadata of the corresponding commit, e.g. its author. We can assign edge weights, e.g., based on the time between the two commits or on the number of files the commits have in common.

Commit graphs offer additional value over plain search in unstructured commit history, as they allow for easy identification of higher-order logical couplings. Higher-order logical couplings are couplings between two files that are mediated by a third file. In the example of Figure 1, files *a* and *c* are never changed together. They do, however, share a common file in commits 14 and 17, *d*. If, for example, *d* is an interface that both *a* and *c* implement, we can infer this indirect relationship between them through their higher-order logical coupling.

IV. Examples

We used commit graphs in [1] and [3]. In [1], we examined the incidence of bug-fixing commits in commit histories and the topological properties of commits in the commit graph. Counting the number of files and commits in all commit histories of a system and identifying the number of bug-fixing commits in these histories, we obtained very high correlations slightly below .9 (significant) between the length and size, respectively, of histories and bug-fixing commits therein. In other words, the longer or larger a history is, the more bug fixes it will contain in that system. We interpret this result to indicate that low(er) correlations mean uneven distributions of bug fixes in the commit history. Such an uneven distribution may point to a disparity in development, i.e. where different parts of the system exhibit highly varying quantities of bug fixes. In examining topological properties of commits, we found that commits with a high node degree, i.e. an above-average count of preceding and succeeding commits, are more likely to incur bug fixes in subsequent commits.

A visual analysis of large commit graphs can reveal the structure of the development history. We found for several commit graphs, that the development of non-core modules was typically situated in isolated parts of the overall graphs. Each graph also had a dense core whose nodes were connected by the files changed most often in the development history - usually core infrastructure classes. We were also able to identify regions of development supposedly independent from the core parts of the system that were nevertheless closely tied to these parts in the commit graphs. Inspection showed that an initial lack in clear definition of interfaces and APIs required subsequent changes on both ends to settle the interactions. We are currently exploring automated, graph-theoretical means of analyzing such structural properties.

An interpretation of a high node degree is that the commit is connecting a number of files that are changed separately before and after that commit. Previous work on logical couplings has shown that they are valuable for suggesting possible or necessary refactorings. A high degree in the commit graphs identifies specific commits to recommend refactorings.

In [3] we examined the relationship between commits that introduce features, improvements and fix bugs. We found that feature commits that have other feature commits in their history are much more likely to be followed by improvement and bug-fixing commits as well as overall code churn in subsequent commits than feature commits that are not preceded by other feature commits.

References

[1] M. Steff and B. Russo, "Co-evolution of logical couplings and commits for defect estimation," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, june 2012, pp. 213 –216.

[2] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Software Maintenance, 1998. Proceedings. International Conference on*, nov 1998, pp. 190 –198.

[3] M. Steff, B. Russo, and G. Ruhe, "Evolution of features and their dependencies-an explorative study in oss," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2012, pp. 111–114.