Determining the Intrinsic Structure of Public Software Development History: an Exploratory Study

Antoine Pietri, Guillaume Rousseau, Stefano Zacchiroli

Determining the Intrinsic Structure of Public Software Development History: an Exploratory Study

Antoine Pietri[1*], Guillaume Rousseau[2*] and Stefano Zacchiroli[3*]

[1]Inria, Paris, France.
[2] Laboratoire Matières et Systèmes Complexes (MSC),UMR 7057, CNRS & Université Paris Cité, 10 rue Alice Domon et Léonie Duquet, F-75013, Paris Cedex 13, France .
[3]LTCI, Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France.

*Corresponding author(s). E-mail(s): antoine.pietri@softwareheritage.org; guillaume.rousseau@u-paris.fr; stefano.zacchiroli@telecom-paris.fr;

Abstract

Collaborative software development has produced a wealth of software source code artifacts (source files and directories, commits, releases, etc.) that have been studied for decades by researchers in empirical software engineering. Due to code reuse and the fork-based development model, those artifacts form a globally interconnected graph of a size comparable to the graph of the Web. Little is known yet about the network structure of this graph; such knowledge is useful to determine the best practical approaches to efficiently analyze very large subsets of it (if not *all* of it) in a methodologically sound manner.

In this paper we determine the most salient network topology properties of the global public software development history as captured by state-of-the-art version control systems (VCS). As our corpus we use Software Heritage, one of the largest and most diverse publicly available archives of VCS data—encompassing 9 billion unique source code files and 2 billion unique commits coming from about 150 million projects or, as a graph, 19 billion nodes and 221 billion edges.

We explore topology characteristics such as: degree distributions; distribution of connected component sizes; and distribution of shortest path lengths. We characterize these topology aspects for both the entire graph and relevant subgraphs.

Keywords: source code, open source, version control system, graph structure, complex network, statistical mechanics

1 Introduction

1.1 The global graph of public software development

The rise in popularity of Free/Open Source Software (FOSS) [1] and collaborative development platforms [2] over the past decades has made publicly available a wealth of software source code artifacts (source code files, commits with all associated metadata, tagged and annotated releases, etc.), which have in turn benefited empirical software engineering (ESE) and mining software repository (MSR) research [3]. Version control systems (VCS) [4] in particular have been extensively analyzed [5] due to the rich view they provide on software evolution [6–8] and their ease of exploitation with the advent of distributed version control systems.

Since the early days of empirical research on FOSS, initiatives [9–12] have been established to gather VCS data in a single logical place, easing access to software artifacts and mitigating selection bias. Over time, the scale at which such collections have been attempted has increased [13–15], with the goal of building an infrastructure that enables research on the global properties of the FOSS ecosystem.

The entire body of public code is not a collection of distant islands of software artifacts, but rather a single very entangled object. This is because modern software products are built by reusing third-party components from other projects, rather than being completely independent pieces of work. This organic code reuse happens through different means, either by simply copying source code across projects (also known as "software vendoring") or by explicitly forking [16, 17] an existing project and then building upon its preexisting development history to evolve it in a novel direction. Modern VCSs also allow one to explicitly reference external repositories (e.g., via Git submodules or Subversion externals) to be fetched as dependencies, which further binds separate software projects together.

Gathering as much publicly available source code artifacts together and representing them in some canonical and deduplicated data model is a way to represent the *global graph of public software development history*: an immense interconnected network of artifacts of various kinds (files, directories, commits, repositories), linking together all derivative works, shared code bases and common development histories, down to the level of the individual source code file. As of today, little is known about the *network structure* of this entangled graph at this massive scale. This is in contrast with what is known about other large graphs that are obtained as byproducts of technology-related human activities, such as the graph of the Web [18] or social network graphs [19, 20].

To fill this gap, in this paper we conduct the first systematic exploratory study on the intrinsic structure of public software development history as an interconnected graph, and its most salient topological properties [21]. To conduct this study we use Software Heritage, one of the largest and most diverse publicly available archives of VCS data—encompassing, in its snapshot analyzed for this paper, 9 billion unique source code files and 2 billion unique commits coming from about 150 million repositories. A dataset of such a scale can be used to approximate the entire body of public software development history.

1.2 Motivations and relevance

Understanding the graph structure of public software development is important for a number of reasons, which we detail below.

Improving our understanding of daily objects of study

Empirical Software Engineering (ESE) focuses on using empirical evidence, such as experiments, observations, and measurements, to understand and improve the software development process. The daily object of ESE studies can vary based on research goals and questions, but it generally involves studying artifacts that are by-products of collaborative software engineering.

While it is widely acknowledged that the raw data of artifacts capture information needed for ESE studies, the tools and platforms employed within developer communities, particularly various source code versioning tools, use different models that lead to data with various degrees of heterogeneity. To build comprehensive datasets that accurately reflect the variety of practices and situations, it is necessary to define ad-hoc representations. This raises the question of which of the representations are the most meaningful (cf. [22] for a similar discussion in a different context).

Many initiatives in aggregating data and artifacts from public software development history result in graphs, but these graphs and their representations exhibit variations and their topological properties may differ wildly. As an illustration, a common practice when studying complex networks is to assess whether degree distributions, at least in part, follow a power law [23, 24]. This characteristic relies on the local properties of the graph and is highly contingent on the chosen graph representation. A key question then becomes: what properties of a given graph representation of public software development history can be abstracted to the level behind the graph and therefore considered *intrinsic*? Beyond the efficiency of a specific representation for a given study, attributing and discussing the intrinsic properties of these ad-hoc graphs as properly reflecting the system's properties requires a careful examination. The present study represents a significant step in this direction.

In the near future, it is likely that we will be able to fully access and routinely work with several representations from different platforms and easily go back and forth between them. This will facilitate the comparison of results, the identification of possible biases induced by these representations, and then the discussion of potential threats to validity.

Avoiding methodological pitfalls

A better understanding of the intrinsic structure of public software development history is needed to avoid making overly strong assumptions on what constitutes "typical" VCS data. These pitfalls have been warned against since the early days of GitHub mining [25], but they have been neither quantified nor described at the scale we consider in this paper yet.

The extent to which repositories on popular forges correspond to "well-behaved" development repositories, as opposed to being outliers that are not used for software development or are built just to test the limits of hosting platforms or VCS technology,

remains unknown. In our experience GitHub alone contains repositories with very weird artifacts: commits with one million parents[1] or artificially built to mimic bitcoin mining in their identifiers,[2] the longest possible paths, bogus timestamps [26, 27], etc.

Outliers are common and of diverse nature. They have already been observed while studying topological properties of public software development history in the context of a clone detection study applied to file clone detection in the FreeBSD Ports Collection [28]. They usually refer to parts of the data that are not relevant to a given study, that are likely to negatively impact its results, and that one ideally seeks to eliminate during data preprocessing. However, whether or not a piece of data is considered relevant is study-dependant. Since our exploratory study aims to produce general-purpose results, we will use the term "outlier" in a narrowly-defined way from now on. Specifically, we assume that the topological properties we measure are well described for the bulk of the data by a "regular" (though discrete in most cases) distribution, and we call *outliers* data points that differ so much from the distribution that their occurrence in a dataset of this size would be statistically very unlikely.

How many outliers of this kind exist is unknown and needs to be documented as reference knowledge to help researchers in the interpretation of their empirical findings.

Determining the most appropriate large-scale analysis approach

Most "large-scale" studies of VCS fall short of the full body of publicly available source code artifacts and either resort to random sampling from existing collections or focus on popular repositories. This is understandable for practical reasons, but remains a potential source of experiment bias.

To enable studies on larger samples of public software development history (up to its fullest extent), in addition to suitable archival and compute platforms [13, 15, 29], we also need an understanding of its intrinsic structure to choose the most appropriate large-scale analysis approach depending on the study needs. For instance, if the graph turns out to be easy to partition into disconnected or at least loosely-connected components, then a *scale-out* approach with several compute nodes each holding graph (quasi-)partitions would be best. Conversely, if the graph is highly connected then a *scale-up* approach based on large-scale graph databases, ad-hoc derived graphs [26] or in-memory graph compression [30] would be preferable. Similarly, knowing that most nodes are part of a single giant connected component (CC) would help in avoiding algorithmic approaches with high complexity on the size of the largest CC.

Note that these topological properties partly depend on implementation choices such as deduplication granularity or the decomposition of the graph into different layers. In the data model used here and for the purpose of this work, deduplication pertains to *all layers* and is carried out up to the level of source code files, a choice that is consistent with what modern version control systems do and facilitates impact and traceability analyses. It should be noted that this deduplication granularity would not necessarily be the most relevant to study development practices such as snippet reuse or minor code changes that do not impact code semantics (e.g., in spacing) but

[1] https://github.com/cirosantilli/test-commit-many-parents-1m, accessed 2021-09-29.
[2] https://github.com/pushrax/round660, accessed 2022-10-21.

still results in different files. To study those one would need, for example, to hash individual code tokens or snippets to capture (or ignore) fine-grained changes in spacing, comments, and ordering [31]. And to conduct studies that need *both* representations one would need to maintain them both and bear the extra costs.

This is ultimately a trade-off between data completeness and genericity. For this study we rely upon the data model of Software Heritage (described in Section 3) and study the topological properties of the graph it engenders. Studying the topological properties of the same corpus (public code) using different large-scale analysis platforms might lead to different results. Conducting a comparative benchmark of these results is outside the scope of this work. Nevertheless we detail in Section 2 some of the consequences of the choices made by Software Heritage and other platforms.

With this work, we pave the way to further network studies of the corpus of software artifacts that (open source) software developers produce and augment daily as a result of their work.

1.3 Study protocol and research questions

In this paper, we conduct the **first comprehensive exploratory study on the intrinsic structure of source code artifacts stored in publicly available version control systems**.

Study protocol

To avoid methodological pitfalls such as publication bias, hypothesizing after the results, and data dredging, we have preregistered a study protocol [32] (phase 1) that this paper implements (phase 2). The consistency between the two phases is documented in Section 6.2. We briefly recall below the research questions and other main aspects of the study protocol; we refer the reader to [32] for full details.

As a corpus we use Software Heritage and its dataset [29, 33], which is one of the largest and most diverse collections of source code artifacts. We analyze a snapshot of the Software Heritage archive consisting of 9 billion unique source code files and 2 billion unique commits archived from about 150 million projects (see Section 4.1 for exact figures). We assess the most salient network topology properties [21] of the Software Heritage corpus represented as a graph consisting of 19 billion nodes of different types (source code files and directories, commits, releases and repository snapshots) and 221 billion edges (or arcs) connecting them.

Software Heritage has coverage from a wide array of sources, among which a full copy of software forges such as GitHub, Bitbucket or GitLab.com, language package managers such as PyPI and NPM, and packages from GNU/Linux distributions such as Debian and NixOS. This variety in coverage allows us to identify heterogeneous properties that appear when doing analyses across different version control systems and different hosting platforms, even though data from GitHub remains highly predominant in the studied dataset.

The extensive deduplication within Software Heritage (SWH) datasets and their associated graph representations, as detailed in Section 3, facilitates the study of properties related to the reuse and cloning of software artifacts. Furthermore, it enables

measurements such as the shortest path length across a software development project's file system or its history. However, for these non-local measurements, it is imperative to address potential biases and, if needed, consider the availability of more suitable representations, as we do in Section 6.

Research Questions

On this corpus we perform an exploratory study, with no predetermined hypotheses, and answer the following research questions:

RQ1 What is the distribution of indegrees, outdegrees and local clustering of the public VCS history graph? Which laws do they fit?
How do such distributions vary across the different graph layers—file system layer (source code files and directories) vs. development history layer (commits and releases) vs. software origin layer?

RQ2 What is the distribution of connected component sizes for the public VCS history graph? How does it vary across graph layers?

RQ3 What is the distribution of shortest path lengths from roots to leaves in the recursive layers (file system and development history layers) of the graph of public VCS history?

Relevance and exhaustiveness of the metrics

This exploratory study is a first step, based on a limited number of metrics, of the intrinsic structure of public software development history as captured by Software Heritage. Other metrics and properties are interesting and relevant, but fall outside the scope of this study. In particular, the betweenness distribution, the degree-degree correlation and degree-grouping correlation have been studied in other works that analyze large-scale systems using complex network theory (cf. Section 2). This study is not intended to be a study of the graph structure of public software development history from the perspective of complex systems theory, although some of the metrics shown are particularly important in this framework.

Because of the uncertainty at the end of phase 1 about the feasibility of analyzing at this scale the proposed metrics, we prioritized metrics associated with local properties for which regular distributions can be observed (RQ1), and those whose analysis could provide useful insights into efficient ways to traverse the different layers of this graph (RQ2) and (RQ3), independently of the measurements performed on them.

Further work will be needed to explore the implications of the results presented here, in particular to understand how they impact dynamic evolution and maintenance (including refactoring, reuse and optimization).

1.4 Paper structure

We discuss related work in Section 2. We detail the data model of the graph that is our main object of study in Section 3. Our analysis methodology is presented in Section 4. The main findings about the graph are in Section 5. Before concluding we discuss them, including threats to validity and deviations from the phase 1 protocol, in Section 6.

2 Related work

Two classes of related work are most relevant for this article. First, there is an extensive body of work in empirical software engineering and mining software repositories about large-scale analyses of source code artifacts. Second, there are comparable studies of the structure of large graphs and complex networks both in computer science and other fields, which provides important tools and methodological considerations which are applicable to the domain of source code artifacts.

2.1 Large-scale analyses of source code artifacts

A few different approaches have been proposed in the past to gather and study large bodies of software and associated development history in a single logical place. Boa [13] does not offer a unified graph view of all stored artifacts, but on the other hand reaches down to the level of abstract syntax trees, allowing fine-grained source code analyses. The graph we study in this paper stops at the granularity of individual files, but is fully deduplicated. All the byte-identical files, directories and commits are deduplicated into a single object. However, similar files (e.g., two files with a difference of a single whitespace) are never deduplicated and considered as separate entities, contrary to Boa or similar approaches such as DejaVu [31]. The dataset we consider in this paper is also about 20 times larger, in terms of the number of contained repositories, than what is available in the largest Boa dataset.

The first large-scale analysis of the Software Heritage archive graph has been conducted in [26, 34]. The transposed version of the graph has been studied in that work, characterizing its various layers showing the existence of a heterogeneous structure within the full graph. The analysis of efficient models for tracking the provenance of source code artifacts at a very large scale was a key aspect of this work. For this use case, queries exclusively involve the transposed version of the graph (obtained by reversing the direction of all edges in the original graph – see Section 3). This earlier work identified potential bottlenecks associated with defining technically transposable graph representations, at the (very) large scale of the public software development history. On one hand, it demonstrated that the entire graph of the SWH project was transposable without loss using finite adjacency lists compatible with most current production databases. On the other hand, it showed that the primary limitation when traversing the transposed graph stemmed from the time required to execute join queries, for 'directory' and 'file' nodes in particular (these nodes being duplicated in the core database of the SWH project).

Followup works focused on specific layers [17], different approaches [30], and/or different datasets [35]. This study is performed on a dataset twice as large as the initial study in [34] and is more exhaustive in terms of measured variables and analyzed layers.

World of Code (WoC) [15] is another recent attempt at creating a mutualized infrastructure of VCS artifacts to enable large-scale analysis. The WoC dataset, reachable through public API, is structured differently than Software Heritage: while WoC does contain information needed to build a graph view of all retrieved source code artifacts, they are "flattened" in a set of *mappings* between object types, e.g., to

quickly find all commits and files belonging to a project without having to perform graph traversals. This approach shares similarities with some previous implementations [28], and WoC could be employed to conduct analyses akin to those presented in this article since all Git objects (commits, tree, blob, and associated metadata) are archived there too. Indeed, with WoC, mapping precomputations accelerate certain analyses, provided they are based solely on available relationships. However, disparity in dataset coverage, which have no reason to be strictly identical, may yield distinct results and differences in design decisions may impede feasibility of some analyses. For example, constructing indegree and outdegree distributions requires access to git tree objects (i.e., source code directories and files) and thus to the git object archive. A comprehensive comparison of WoC and SWH is beyond the scope of this exploratory study.

𝕃ISA [36] is a framework to analyze VCS-stored source code at a fine-grained level, reaching down to abstract syntax tree nodes. 𝕃ISA also deduplicates software artifacts as a way to reduce redundancy and speed up analyses albeit only at the granularity of files and not across all artifact types as in our study. But 𝕃ISA introduces an additional deduplication technique based on vertex compression using commit ranges. This leads to a reduction of the memory and computational resources required of 50% to 99% depending significantly on language and project size[3]. These measurements are however based on a small set of 4,000 projects and warrant further investigation on a larger scale to assess 𝕃ISA's vertex compression efficiency across corpus like Software Heritage and WoC. The graph representation employed by 𝕃ISA would also need to be compared in detail to earlier models like the compact model [26] which involves vertex compression of the path vertex between commits and files. Yet these works give an indication that vertex compression, when applied to graphs representing the public software development history, may be efficient at larger scales. In practice, graph representations based on vertex compression may prove a worthy alternative to the model employed in this study. For example, it could offer enhanced effectiveness when measuring the distribution of the shortest paths in the filesystem layer, where, in our case, sampling became necessary.

A recent work [37] by Trujillo et al. highlighted representativeness issues when only using popular platforms like GitHub as a convenience sample, by showing large disparities in the characteristics of projects stored in these centralized platforms and those existing outside them. The representativeness problem outlined in this study is one of the main motivations for characterizing topological structure at the scale of the entire graph, including taking into account and analyzing outliers [28].

2.2 Complex network analyses of software artifacts

The study of complex networks [21, 38, 39] is dedicated to the analysis and characterization of graphs that exhibit non-trivial topological properties. Large graphs that emerge naturally as byproducts of human activities have been common objects of study in the field for several decades now. One of the best examples and most studied complex network is the graph of the World Wide Web [40]. The understanding of it

[3]For example, one of the instances provided, representing 7,947 revisions of the *Reddit* project using vertex compression, requires only about 0.3% of the space needed to represent all revisions individually.

obtained from complex network studies has given valuable practical insights to design crawling engines, searching and ranking methods, compact representation techniques, as well as sociological insights into its growth, social structure, and communities.

Studying the topology of the graph of the Web using large corpuses has been done as early as 2000 in [41] at a large scale: 200 million nodes (Web pages), 1.5 billion edges (hyperlinks between them). In this paper we apply a similar analysis approach, including the study of indegree and outdegree distributions, as well as the distribution of connected component sizes.

Further analyses of the graph of the Web have been pursued more recently in [18], which extends previous work characterizing different aggregation levels (pages, hosts, and domains). This study – which uses the WebGraph compression framework [42, 43] as we do – is particularly relevant in the context of this work, because the size of the analyzed corpus is comparable to ours (3.5 billion nodes, 128 billion edges). Among the notable findings are 1. the challenging of previous characterizations of various distributions as matching power laws, and 2. the realization that the "bow-tie" structure, previously described for this network, is strongly dependent on the crawling process and consequently cannot be seen as an *intrinsic* property of the underlying real-world system. This underscores the importance of empirically challenging *a priori* assumptions about properties of large-scale graphs. And in turn it explains the more conservative methodology used in this paper (Section 4) regarding the study of the tails of distributions and why we favor a more explicit multiscale analysis of connected component aggregation mechanisms across layers, which was previously introduced [17].

Other computer-related graphs that have been widely studied as complex networks are social network graphs, in which nodes correspond to users and edges to "friendship" or "follower" relationships. All major commercial social networks have been studied under these optics, including the social graphs of Facebook [19] and Twitter [20].

In the realm of interconnected software-related artifacts, software dependency graphs have also been studied and characterized as complex networks. [44] looked at the basic properties of the dependency graphs of Debian and BSD packages, again with a methodology similar to the web graph studies, albeit on a much smaller graph. [45] is a different take on characterizing the graph of software dependencies in Debian, considering semantic rather than merely semantic dependencies. On the same graph, [46] adds the time dimension to the analysis by studying its dynamic growth and trying to fit it to a Zipf law model. The Software Heritage graph is known to grown exponentially over time [26], with different growth rates for different layers. We do not look into dynamic growth of the graph in the present article; it hence remains a relevant research direction for future work.

The designs of specific software systems have also been studied as complex networks, mainly by looking at the relationships between related software modules and/or classes in object-oriented systems [47–49].

Several works have highlighted the importance of topological properties, in particular to build predictors of bug severity, high-maintenance parts of software, and failure-prone versions [50], and have studied different approaches combining graphs

at several scales [51] to characterize the evolution of systems producing software artifacts [52, 53]. That includes collaboration between software developers [54] and software engineering researchers [55] has also been analyzed using similar techniques. In this paper we do not look at collaboration graphs (which would be graphs *derivable* from, rather than natively represented in, our data model) and neither dig archived software projects to establish their dependencies. Both remain interesting areas for further exploration, but are out of the scope for this study.

This paper provides a fundamental stepping stone for those further analyses, providing a topological characterization of one of the largest available representations of the global VCS graph of public code.

3 Data model

Before jumping into the analysis methodology in the next section, we detail in this section the object of our study. The public software development history is here represented by a *directed acyclic graph* with *typed* nodes, where different node types correspond to distinct source code artifacts in the real world. The acyclic nature is guaranteed by construction, relying on Merkle graphs and the use of intrinsic identifiers. Additionally, both nodes and edges can carry named *properties* associated to a value. Formally the graph can be interpreted as a *property graph* [56, 57], a generic graph data model that is popular in the field of (graph) databases.

In the following we describe the various kinds of nodes and edges that compose the graph, what real-world version control system (VCS) data they stand for, and the rules that govern the construction of the graph. Those rules impose topological constraints on the graph (e.g., a file content node has no outgoing edge), a fact to keep in mind to avoid pursuing implausible research questions.

3.1 Nodes and edges

The graph of public software development contains nodes of different types, depicted in Figure 1. Each type of node corresponds to a different kind of software artifacts that can be found in real-world VCS.

Blobs

File *blobs* (or "file contents") represent the raw content of source code files, as recorded in modern VCS. A blob contains only the data stored in a file as a raw sequence of bytes. File names and other properties normally associated to the more abstract notion of "file" are not associated to blob nodes. Other types of nodes, and most notably directories, attach such directory-dependent information to blobs.

Blobs are identified by a cryptographic hash computed from the full binary data they contain. An illustration of a blob object is shown in Figure 2.

Directories

Directories represent source code trees. Each directory is a list of named directory entries, each entry pointing (with an outgoing edge) to either blob nodes ("file

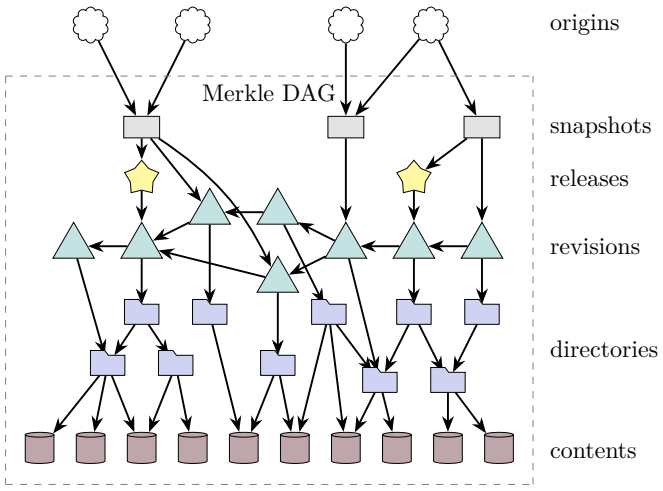


Fig. 1: Data model for the graph of public software development: a directed acyclic graph (DAG) linking together deduplicated software artifacts shared across the entire body of (archived) public code.

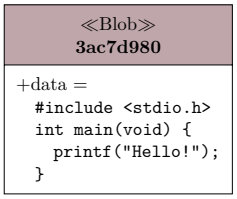


Fig. 2: Example of a blob object.

entries"), directory nodes ("directory entries"), or revision nodes ("revision entries"). Each outgoing edge from a directory is associated to a local name (i.e., a *relative* path without any path separator) and permission metadata (i.e., a Unix permission mode like `0o755` for an executable file). While file and directory entries are the most common to form nested source code trees, *revision entries* also exist and are used to represent sub-directories that reference specific revisions from external repositories, as it is permitted by VCS like Git (to reference so called "git submodules") and SVN (with "subversion externals"). Permission metadata is also used to recognize symbolic links from regular files.

A directory node is identified by a cryptographic hash of a canonical textual representation of all outgoing edges, which includes the identifier of target nodes. An illustration of a directory node is shown in Figure 3.

Commits

Commits (or "revisions") are point-in-time captures of the state of the entire source code tree of a project. Each commit has an outgoing edge to the directory node that

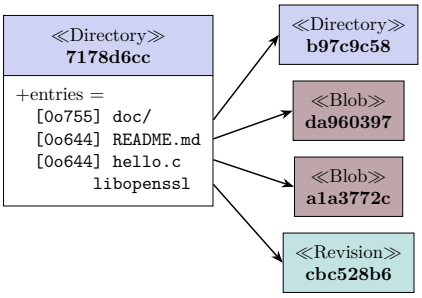


Fig. 3: Example of a directory object.

represents the "root" directory of the versioned project at the time the commit is/was recorded. The following properties are associated to commit nodes:

- *commit message*: a descriptive, human-targeted message explaining the reasons for the change;
- *author*: the name and e-mail of the person who authored the commit;
- *date*: the timestamp at which the commit was authored, including timezone information;
- *committer*/*committer date*: two properties analogous to author/date, but capturing the person who actually committed the change (who is not always the person who *authored* it, in particular in development workflows that rely on code reviews) and when the commit happened.

Finally, each revision points via outgoing edges to an ordered list of parents: zero parent for the first commit in a given development history (e.g., first commit in a VCS repository); one parent for non-merge commits; two or more parents for commits that merge together several development branches.

Commit nodes are identified by an intrinsic hash of a canonical textual manifest containing all their metadata, their parent identifiers, and the identifier of the directory node denoting the root of the source tree at the time of the commit. An illustration of a commit node is shown in Figure 4.

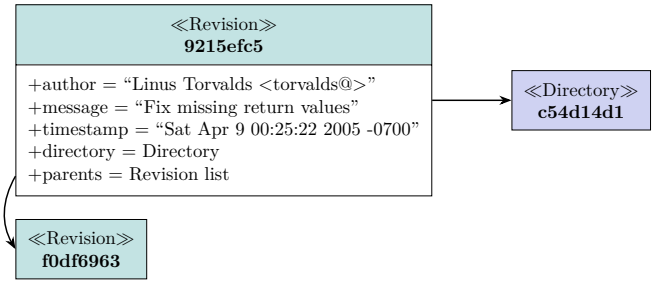


Fig. 4: Example of a revision object.

Releases

Releases (or "tags") denote marker objects that label specific commit nodes as relevant project milestones, e.g., commits that were distributed to the user base, as well as other steps in a release cycle ("alpha", "beta", etc.). Releases are marked with a specific and usually mnemonic short name (e.g., `2.0`). Aside from this name and an outgoing edge to the target commit node, releases are associated to additional properties:

- *message*: analogous to commit messages, generally used to include release changelogs;
- *author*/*date*: analogous to the homonym properties for commit nodes, but used here to identify the developer making a release.

Releases are identified by a cryptographic hash taken on a canonical text manifest containing release name, release properties, and the identifier of the commit node they reference. An illustration of a release node is shown in Figure 5.

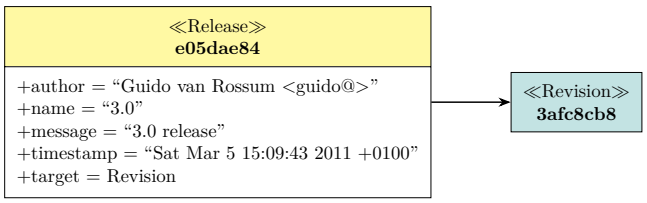


Fig. 5: Example of a release object.

Snapshots

Snapshots are point-in-time captures of the full state of a project development repository. Unlike revisions, which capture the state of a single development branch, snapshots capture the state of all the branches and releases in a repository. Snapshots do not represent source code artifacts that are found in VCS *per se*, but rather a "picture" of a VCS repository at the time it is visited (or archived, in the case of Software Heritage).

Each snapshot has an outgoing edge for each branch present in a repository, labeled with the branch name (e.g., "main", "refactoring", or "v0.1.2"), and pointing to the most recent node in that branch (which is usually either a commit or release node). The data model also supports branch aliasing: some branches stored in snapshots do not point to a specific artifact, but rather reference another branch name in the same snapshot. These are to be treated as symbolic links to the target of the branch they reference. Even if the canonical model does not associate specific nodes in the graph to each of the branches that co-exist at a given time, as some representations might do, the information about the branches is preserved and such representations can therefore be reconstructed a posteriori if needed.

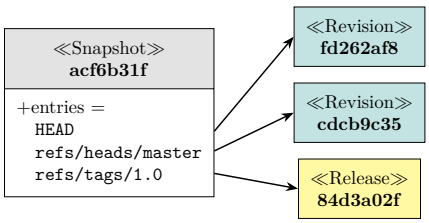


Fig. 6: Example of a snapshot object.

Snapshot nodes are identified by a cryptographic intrinsic hash, computed on a canonical textual manifest that associates to each branch name the identifier of the target node. An illustration of a release node is shown in Figure 6.

Origins

Software *origins* represent the specific places from which source code artifacts were retrieved by a Software Heritage crawler. Each origin is represented by a canonical URL (e.g., https://github.com/octocat/Hello-World for a Git repository or https://pypi.org/project/black/ for a source package archived from a package manager repository).

Each origin can be "visited" multiple times by a crawler of a given type, e.g., Git repositories will be repeatedly visited by Git crawlers. At each visit the state of the repository will be associated to a snapshot object, possibly a new one if the repository state has changed since the last visit, possibly the same if it has not.

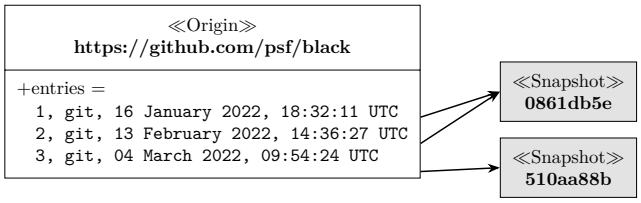


Fig. 7: Example of an origin object.

Each origin node in the graph has outgoing edges to snapshot nodes observed there, one for each visit performed on the corresponding repository (or source package). Edges from origin to snapshot nodes have as properties the visit timestamp, the involved crawler (e.g., git, svn, etc.), and a sequential visit number. An illustration of an origin node is given in Figure 7.

Note that the visit frequency of an origin is not only correlated to project activity (larger and/or active projects are expected to be visited and updated more frequently) but also depends on the specific crawling processes deployed by the platform responsible for content aggregation and updates. While a detailed description and comparison of such processes is beyond the scope of this study, it is important to acknowledge that they may introduce significant biases concerning certain *intrinsic* properties of

the software development project history. These biases will be discussed in detail in Section 6.

3.2 Merkle properties

With the notable exception of origin nodes (which are *not* identified by intrinsic cryptographic hashes, but by URLs), the rest of the graph forms by construction a *Merkle structure* [58]. Specifically this graph representation of public software development history forms a Merkle direct acyclic graph (or "Merkle DAG" for short).

Like all Merkle structures, this graph enjoys interesting and useful properties. In particular it is worth noting that as long as two Merkle DAGs are *complete* (i.e., no node is missing), one can efficiently determine if they are identical or not by simply comparing the identifiers of all their root nodes, which in our case are snapshot nodes. Also, in case they differ, one can efficiently identify the topmost differences by performing a parallel visit on the two graphs.

Furthermore, Merkle structures of the graph representation used natively deduplicate all artifacts stored in it. As node identifiers are not assigned but rather computed intrinsically on the content of the nodes and their outgoing edges, adding a node to a Merkle structure is an idempotent operation. Trying to add to our data model the same source code blob or tree multiple times (e.g., because it is found in multiple commits in the same repository) will result in adding it only once; the same applies to all types of nodes, so a commit that occurs in multiple branches or repositories will be stored only once, and even a full unmodified repository state (i.e., a snapshot node) found across multiple repository forks will be present in the graph at most once.

3.3 Cardinality

One immediate property to draw from this data model is that the different node types are arranged hierarchically in "layers" and are linked together in a way that induces typing rules on graph edges. For instance, a directory cannot reference, neither directly nor transitively, a release or an origin; and a blob node cannot reference any other graph node at all. More generally, artifacts have descendants from lower hierarchical layers in the graph structure.

Figure 8 shows a cardinality diagram of the relationships between the different types of nodes in the graph, which allows one to visualize the construction constraints that apply to graph edges. A few observations are in order:

- Most relationships are many-to-many. Only release → commit and commit → directory relationships are many-to-one, as they can only point respectively to a single commit and a single directory.
- Commit and directory nodes are parts of recursive relationships. Commits point to their parent commits, while directories point to their children (sub)directories and files. These two node types are thus what gives the DAG an arbitrary depth, as all the other layers have a fixed maximum height. Note however that this cycle in the graph cardinality diagram does *not* induce cycles in the graph itself, because graph nodes are created bottom-up and need to know in advance the cryptographic

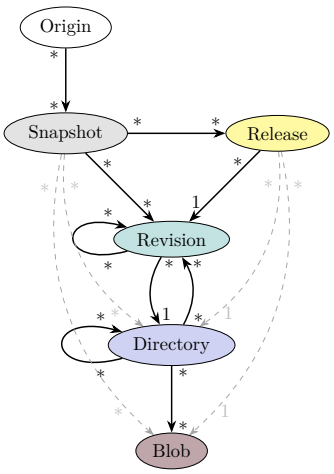


Fig. 8: Cardinality diagram of the edge types the graph of software development, with the usual notation conventions: $* \to *$ arrows denote many-to-many relationships, $* \to 1$ many-to-one relationships. Dashed arrows represent rare relationships supported by the data model but ignored in the study.

identifiers of target nodes (which hence need to exist *before* their parents) in order to be identifiable.
- Commits and directories are also mutually recursive, as commits point to directories and directories can occasionally point to commits (for submodules/externals). This is the only case in which a node can point to a node from an upper layer in the cardinality diagram of Figure 8.

Note: in the wild one can encounter (and we have encountered) VCS repositories that contain relationships between source code artifacts that are not depicted in Figure 8. For instance, Git allows you to tag blobs and directories as releases or to use blobs as branch destinations. These are anomalies that in most cases result in non usable VCS data. We measured these occurrences in our corpus and verified that they are rare and unconventional (less than 0.0004% of the total number of edges from releases and snapshots) and thus, even if they *can* be represented in the Software Heritage data model, we have excluded them from our discussion for the sake of simplicity of presentation.

3.4 Layers

While each relationship can be analyzed independently, it is useful to regroup nodes and edges by type into logical layers that are conceptually meaningful. To that end we define the following subgraphs of the global public software development graph:

- *Full graph*: the entire graph of public software development;
- *Filesystem layer*: subset of the full graph consisting of blob and directory nodes only, and edges between them;

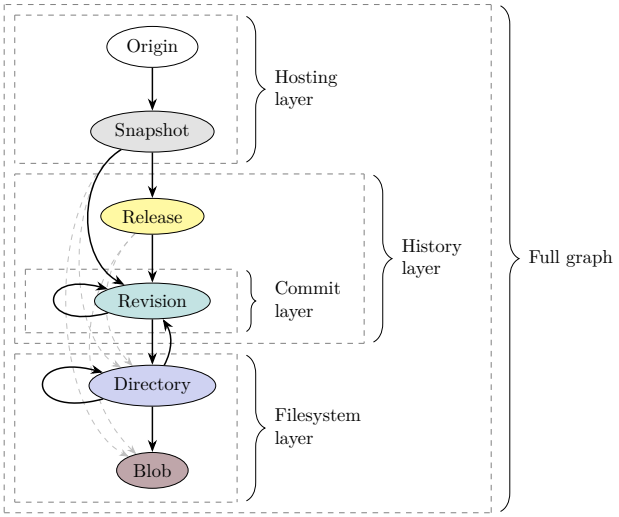


Fig. 9: Logical layers used to analyze subsets of the graph.

- *History layer*: subset of the full graph consisting of commit and release nodes only, and edges between them;
- *Commit layer*: subset of the history layer consisting of commit nodes only, and edges between them;
- *Hosting layer*: subset of the full graph consisting of origins and snapshot nodes only, and edges between them.

Figure 9 shows the various layers, which types of nodes belong to each of them, as well as how edges connect them. In the following we will study properties of both the full graph of public software development and the specific subgraphs named above.

4 Datasets and methodology

Now that we have identified the object of our study—the global graph of public software development, as captured by Software Heritage—we present in this section our methodology for analyzing it. We describe first the exact data corpus from which the graph was extracted and how we obtained it; then we detail the methodology used for the actual analyses.

4.1 The Software Heritage graph dataset

All research questions require analyzing a specific representation and version of the public VCS history graph. Software Heritage provides periodic dumps of the archive under the name of the *Software Heritage graph dataset* [33]. The graph dataset contains the entire fully deduplicated development graph with all associated metadata, but not the actual content of archived source code files, which are much more voluminous and should be retrieved separately. This exclusion is not problematic for our

Table 1: Node (top) and edge (bottom) statistics of the studied graph dataset.

Layer	Node type	Nodes	%
hosting	origins	147,453,557	0.76%
	snapshots	139,832,772	0.72%
history	releases	16,539,537	0.09%
	commits	1,976,476,233	10.22%
filesystem	directories	7,897,590,134	40.86%
	contents	9,152,847,293	47.35%
Total nodes		19,330,739,526	100%

Layer	Edge type	Edges	%
hosting	origin → snapshot	776,112,709	0.35%
	snapshot → commit	1,358,538,567	0.61%
	snapshot → release	700,823,546	0.32%
history	release → commit	16,492,908	0.01%
	commit → commit	2,021,009,703	0.91%
	commit → directory	1,971,187,167	0.89%
filesystem	directory → directory	64,584,351,336	29.16%
	directory → commit	792,196,260	0.36%
	directory → blob	149,267,317,723	67.39%
Total edges		221,488,073,659	100%

needs as all research questions revolve around studying the *structure* of the VCS graph, without having to mine the content of individual source code *files*.

The experiments we present here are based on the data export dated 2020-12-15 – see Section 6.2 for a discussion on the quality of the dataset. In terms of size, this export contains 19 billion nodes and 221 billion edges in total. Table 1 gives a detailed breakdown of each node and edge types. At first glance we can see that the filesystem layer of the graph contains most of the nodes (88%) and edges (97%) in the graph: new versions of source code files and directories in public code are produced in much higher volumes than other source code artifacts such as commits and releases. The number of visits and origins on the other hand depend only on the crawling throughput of the Software Heritage archive and its coverage of real-world collaborative development platforms.

Tables 2 and 3 give an overview of such coverage in the studied corpus[4], broken down along various dimensions: the mechanism used to retrieve the source code artifacts (for the most part a VCS or a source package format), the domain of the forge or package repository where they were hosted, and the number of origins and visits of them present in the archive. We can notice that Git dominates the corpus as a source code distribution mechanism, and that GitHub is the dominant forge. Nonetheless

[4]Platform coverage information are not directly stored in the representation used for this study. Data from Tables 2 and 3 are extracted from [59] and correspond to the SWH dataset published three months after the one targeted in this study (2021-03-23 vs. 2020-12-15). The number of origins varies by about 2% between the two snapshots, and no significant difference is expected between the two datasets in that respect.

Table 2: Crawling statistics: number of origins and visits by origin type ("$< \varepsilon$" denotes percentages below 0.01%).

Origin type	No. of origins	%	No. of visits	%
git	136,684,905	98.0%	545,124,995	53.9%
npm	1,533,346	0.9%	300,806,714	29.7%
svn	575,952	0.3%	735,135	0.07%
hg	381,058	0.2%	6,105,706	0.60%
pypi	239,522	0.1%	147,102,654	14.5%
deb	72,303	$< \varepsilon$	10,894,679	1.07%
cran	18,019	$< \varepsilon$	29,596	$< \varepsilon$
ftp	1,205	$< \varepsilon$	1,205	1.19%
deposit	900	$< \varepsilon$	1,277	1.26%
tar	385	$< \varepsilon$	955	9.44%
nix/guix	2	$< \varepsilon$	445	4.40%
Total	139,507,597	100%	1,010,810,868	100%

Table 3: Crawling statistics: number of origins and visits by forge domain, for domains with at least 1,000 origins ("$< \varepsilon$" denotes percentages below 0.01%).

Forge domain	No. of origins	%	No. of visits	%
github.com	147,881,630	96.1%	546,877,021	54.1%
bitbucket.org	2,058,279	1.33%	10,871,128	1.07%
www.npmjs.com	1,534,976	0.99%	300,808,344	29.7%
gitlab.com	990,334	0.64%	5,022,358	0.49%
pypi.org	239,620	0.15%	147,102,752	14.5%
gitorious.org	120,380	0.07%	120,392	0.01%
Debian	38,414	0.02%	10,661,918	1.05%
salsa.debian.org	33,617	0.02%	105,690	0.01%
snapshot.debian.org	33,044	0.02%	33,044	$< \varepsilon$
git.launchpad.net	19,571	0.01%	21,198	$< \varepsilon$
framagit.org	18,433	0.01%	132,803	0.01%
cran.r-project.org	18,019	0.01%	29,596	$< \varepsilon$
hdiff.luite.com	13,861	$< \varepsilon$	191,570	0.01%
gitlab.gnome.org	8,016	$< \varepsilon$	21,837	$< \varepsilon$
gitlab.freedesktop.org	4,752	$< \varepsilon$	1,172,842	0.11%
gitlab.inria.fr	3,628	$< \varepsilon$	9,921	$< \varepsilon$
codeberg.org	3,623	$< \varepsilon$	3,733	$< \varepsilon$
git.savannah.gnu.org	2,959	$< \varepsilon$	7,008	$< \varepsilon$
git.baserock.org	2,912	$< \varepsilon$	4,687	$< \varepsilon$
anongit.kde.org	2,488	$< \varepsilon$	7,389	$< \varepsilon$
code.google.com	2,240	$< \varepsilon$	2,240	$< \varepsilon$
phabricator.wikimedia.org	2,224	$< \varepsilon$	631,835	0.06%
git.kernel.org	2,083	$< \varepsilon$	4,224	$< \varepsilon$
fedorapeople.org	1,691	$< \varepsilon$	4,173	$< \varepsilon$
ftp.gnu.org	1,590	$< \varepsilon$	2,160	$< \varepsilon$
gitlab.ow2.org	1,119	$< \varepsilon$	3,111	$< \varepsilon$
phabricator.kde.org	1,030	$< \varepsilon$	6,269	$< \varepsilon$
Debian-Security	1,028	$< \varepsilon$	199,900	0.02%
git.torproject.org	1,014	$< \varepsilon$	2,503	$< \varepsilon$
Total	153,838,272	100%	1,010,810,868	100%

there is a long tail of both source code distribution mechanisms[5] and hosting platforms[6] that are also present in the corpus. They account for a very diverse corpus, even though it is *quantitatively* dominated by the popular technological choices of the day among developers.

4.2 Graph processing approach: graph compression

The Software Heritage graph dataset is available as a set of relational tables in a columnar format. These tables define the nodes of the graph and the edges that link them, as well as their metadata (including the node type, cf. Section 3.1).

One of the available representations of this graph is the "edge dataset", an enormous list of edges in compressed CSV format[7], with one source/destination pair of *node identifiers* per row, looking like this:

```
swh:1:snp:0c89dafb2703f1ba544f4b3d6ceed77ede26bf60  swh:1:rev:ec5b32b2bf01f8736df11a55aeae5b2f070c1aac
swh:1:rev:ec5b32b2bf01f8736df11a55aeae5b2f070c1aac  swh:1:rev:0ff853c564c0760132430d037bee26608572cb05
swh:1:rev:ec5b32b2bf01f8736df11a55aeae5b2f070c1aac  swh:1:dir:2d95effdef99af584d46a3b88285fb62ff0ad243
swh:1:dir:2d95effdef99af584d46a3b88285fb62ff0ad243  swh:1:cnt:8d83a613b56f7583dd79dff917de4236195d9172
swh:1:dir:2d95effdef99af584d46a3b88285fb62ff0ad243  swh:1:dir:1271bfc7aaf29f91bd3c65df24ce439d3e6e1c76
...
```

Graph nodes are identified in the edge dataset by their Software Heritage Identifiers (SWHID) [60] (starting with "`swh:1:...`"), which are standardized textual representations of the intrinsic cryptographic identifiers of each Merkle DAG node, together with node type information: `snp` for snapshots, `rev` for commits ("revisions"), `rel` for releases, `dir` for directories, and `cnt` for blobs ("file contents"). The flatness and simplicity of this format make it appropriate to study the topology of the graph, as it is the *de facto* standard format used as the input of most graph analysis pipelines.

Some of the analyses we require could easily be performed in a scale-out/distributed fashion (e.g., degree measurements, for RQ1), others benefit more from a scale-up/centralized approach (e.g., connected components for RQ2 and path lengths for RQ3) to avoid incurring the price of costly synchronization points. For uniformity, and also because there is no real drawback in doing *all* analyses with a scale-up approach since we have to do at least some of them in that fashion, we adopted a centralized approach for all analyses.

In particular, based on previous work on large VCS graphs [30], we decided to use the pre-existing `swh-graph` pipeline to load and then analyze the entire graph structure in memory on a single machine. `swh-graph` is a set of tools based on the graph compression and analysis framework WebGraph [42], used to analyze other large graphs such as the graph of the Web and the graphs of large social networks. Note that the version of `swh-graph` used to carry out this study does not preserve some of the relationships between nodes, with the benefit that it improves the compression factor but at the cost of the irreversibility of the transformation (see the discussion in Section 6.2).

[5] Other popular and historical VCS; Debian, NPM, PyPI, CRAN, Nix, and Guix source packages.

[6] Several popular and historical forges, various self-hosted instances of GitLab and other forges, as well as GNU/Linux distribution and package manager repositories.

[7] At the time of writing the CSV format has been replaced by Apache ORC, which allows efficient parallel processing. The CSV files are still available for the version of the dataset used for this paper (2020-12-15) at the Amazon S3 public bucket s3://softwareheritage/graph/2020-12-15/edges/.

According to [30], and as of 2020, using `swh-graph` both the direct and transposed graphs can be loaded into main memory occupying ≈ 200 GiB of RAM (≈ 100 GiB for each graph "direction") and then processed efficiently: a full breadth-first search (BFS) visit with a single-thread requires 2-3 hours, whereas the amortized cost of accessing a single random edge in the graph is close to the cost of a single memory access.

The compression pipeline included in `swh-graph` is able to compress the original 7.4 TiB edge dataset given as its input into a few different files:

- `graph.graph`: a 134 GiB file containing a random access compressed representation of the edges in direct order;
- `graph-transposed.graph`: a 107 GiB file, using the same format as the previous file, but containing the transposed edges;
- `graph.node2type.map`: a 7 GiB file containing a random access mapping between node identifiers and their type.

Once memory-mapped by `swh-graph`, these files together give access, via a Java API, to all the required primitives to perform computations on the topological structure of the graph: iterating on all graph nodes, iterating on the predecessors and successors of a given node, as well as efficiently looking up the indegree and outdegree of a node. As the files obtained have a relatively reasonable size (less than 250 GiB, or about 4% of the edge dataset before compression), they can be loaded in memory on server-grade commodity hardware equipped with a few hundred GiB of RAM.

Layer sub-datasets

Most of the conducted analyses needed to be run both on the graph as a whole and on the various layers described in Section 3.4, each of which is a subgraph of the initial corpus. After trying out different options to export these layer subgraphs, we settled on a solution based on a lazy lightweight wrapper, as it offers reasonable performance and flexibility and reduces the complexity of the pipeline. In practice, this means that only the main graph is stored in memory, and the "layers" are implemented as virtual wrappers which reimplement the graph access primitives by checking whether the node types belong to the current layer in use; when they do not, the corresponding nodes and the edges that connect them are hidden and hence excluded from analysis.

4.3 Analysis methodology

We now present the algorithmic and practical approaches we have implemented to extract from the graph the raw data needed to answer our research questions, as well as the analysis protocol followed to process it.

4.3.1 Graph directionality

The graph of software development is a directed acyclic graph (DAG). While its directionality has an important semantic meaning (e.g., parenthood relationships between revisions or directories do not commute), taking it into account is appropriate for

some of the observables we are after (e.g., indegree/outdegree) but not for others. For instance, all strongly connected components in a DAG have a size of one.

As such, it is sometimes useful to consider the graph as being undirected by "symmetrizing" it, that is, computing the union of the directed graph and its transposition. Conceptually, doing so corresponds to studying how the nodes are linked together by the underlying relationships, rather than the relationships themselves. Symmetrization can always be performed lazily and in constant memory using a directed graph and its transposition. Each observable of interest is computed on both graphs, then the results are added together.

4.3.2 Degree distributions

To analyze degree distributions we measure the indegree and outdegree distributions of each layer of the graph. That is, for each node, we count the number of edges pointing *to* it (indegree) and the number of edges starting *from* it (outdegree). This can be done performing a single pass on the graph in $\mathcal{O}(|V| + |E|)$ while maintaining a frequency histogram of negligible size (a few megabytes) per layer.

A single pass is sufficient to compute the degree distributions of all layers. As we iterate on the edges adjacent to each node, we check the node type of their source and destination, and only increment the frequency counter of a specific layer if these types belong to the layer.

4.3.3 Scaling factor

A common and useful way to quantitatively characterize degree distributions in complex networks is to focus on their "tails", notably by estimating the exponent associated with the best power law fitting the distribution tail. A pure power law distribution $p(d_i) \propto d_i^{-\alpha}$ has a well-defined mean if the scaling factor $\alpha > 2$ and has a finite variance if $\alpha > 3$, due to the behavior of its first and second moments at the limit [61].

Determining precisely whether any of the tails of the distributions we obtained correspond to a power law is beyond the scope of this particular study. However, getting an indication about the behavior of the mean and variance when considering the most extreme statistical events (such as the highest degrees or the longest chains of revisions) of this information is nevertheless useful from the point of view of large-scale empirical software engineering, as it can help to determine the most appropriate analysis approaches (cf. supra).

We follow the approach proposed in Clauset et al. [62], limiting ourselves to the first two steps (out of the three) of their methodology. We use the estimator $\hat{\alpha}(d_{min})$ of the best power law exponent, which depends on an arbitrary degree threshold d_{min} below which the behavior of the distribution is ignored:

$$\hat{\alpha}(d_{min}) = 1 + \left[\sum_{d_i \geqslant d_{min}} n_i \log \frac{d_i}{d_{min}} \right]^{-1}$$

where n_i is the number of nodes with a value equal to d_i.

In the first step of the methodology, the value d_{min} corresponds to the "beginning" of the tail of the distribution for which $\hat{\alpha}$ is the power law exponent α maximizing the likelihood of the observed tail of the distribution. The second step – following the method proposed by Clauset et al. [62] – aims to find the best cut-off value d_{min}. It is done by computing the Kolmogorov-Smirnov distance for all possible d_{min} and picking the one minimizing this distance between the distribution and the power law tail. The last step consists in testing the relevance of the whole procedure determining the p-value, which is the probability of observing the same deviation between a power law with the exponent $\hat{\alpha}$ (Step 1) for d_i greater than the best d_{min} value found (Step 2), and synthetic samples of equal size $N = \sum_{d_i \geqslant d_{min}} n_i$ (Step 3).

Since (as per the registered study protocol) we do not evaluate the impact of the outliers on the reliability of the method [63], and in particular on Steps 2 and 3, we stop short of determining whether the distributions actually correspond to power laws, and we show, for each distribution in this study, the best power law fit according to Steps 1 and 2 of this method. Consequently, we will limit the discussion of the characteristics of the distributions by identifying those whose decay for the largest values is sufficiently slow to rule out an exponential decay. We will then speak of "heavy tailed" distributions and will use the measured exponents only as an indication of this behavior, leaving to future studies the finer analysis when it seems relevant.

4.3.4 Clustering statistics

To get an overall sense of how the nodes in the different layers tend to cluster together in high-density groups, we estimate the global undirected clustering coefficient [64] of the entire graph. As getting an exact value of the clustering coefficient has a complexity of $\mathcal{O}(|V|^3)$, which is impractical at this scale, we use the approximation heuristic described in [65], which is based on uniform random sampling.[8]

At a finer-grained level, we also analyze the local undirected clustering distribution: for each node, the number of edges between nodes pointing to or from it. This is equivalent to the local clustering coefficient without dividing it by the number of possible triangles in the undirected graph. As before, to estimate this distribution we resort to uniform sampling of the nodes of each layer.

4.3.5 Connected components

We compute connected components on (various layers of) the undirected graph obtained by "symmetrizing", as discussed above, the original Merkle DAG. The distribution of connected component sizes is a crucial metric to understand whether the global corpus of public code development can be divided in reasonably-sized partitions, potentially as a way to perform large-scale analyses of it in a distributed fashion.

We can assign each node to a connected component using a BFS traversal of each layer, and then compute the size of each component by simply counting the number of nodes in it. The traversal has linear complexity of $\mathcal{O}(|E|)$ time and $\mathcal{O}(|V|)$ memory, which allows us to get an exact result for the entire graph. During traversal the frontier

[8] See the complexity discussion in Section 6.2.

of nodes still to be visited is maintained as an on-disk queue in order to lower RAM usage.

We give particular attention to the *giant components* of the different layers: large connected components which contain a significant fraction of all the nodes in the graph. The relative size of these giant components are the main indication as to whether the graph can be partitioned in isolated clusters. Our expectation is that the deeper we go in the layers of the graph, the more the nodes will be connected together in giant components.

For the top layers of the graph, connected components intuitively capture the notion of "repository forks" [17]. If two origins are in the same connected component when considering the hosting and history layers, it means that they share some amount of development history (e.g., they both contain the same commit). Therefore, the distribution of connected component sizes in this case will correspond to the distribution of fork network sizes, and the giant components will contain projects with large numbers of forks.

When the filesystem layer is included in the connected component analysis however, common artifacts such as the empty file or the empty directory tend to connect together a large number of unrelated origins in the same component. The quantitative analysis will help measure the extent to which this process connects the nodes together in giant components as we include deeper layers.

In order to analyze how the connected components merge together as deeper layers are included, we also report these distributions in terms of the number of software origins they contain, which remains invariant. To compare the distributions, we calculate the Kolmogorov-Smirnov distance, i.e., the difference between the weighted partition functions. We note $f_1(s)$ the probability of having a connected component of size s within the layer L_1, and $f_{1+2}(s)$ the same probability considering layers L_1 and L_2 together.

We then compute $F_w(s, f_1, f_{1+2}) = \sum_{s_i<s} s_i(f_1(s_i) - f_{1+2}(s_i))$, whose maximum absolute value is equal to the Kolmogorov-Smirnov distance of the size distributions of the components to which the origin nodes belong. Note that $F_w(1) = 0$ and $F_w(s_{max}+1) = 0$, where s_{max} is the size of the largest existing component (i.e., the largest s, such that $f_{1+2}(s) \neq 0$). The F_w function measures a *flux*, which describes how origins from separate components get aggregated in a single component as additional layers are included. When the difference is positive, the sum includes mostly origins which were in small components within L_1 and are now in larger components within L_{1+2}. This approach is similar to other analyses of aggregation mechanisms of connected components such as [18], although with a more accurate way of measuring it.

The aggregation process of connected components can thus be characterized layer after layer, until the largest components are formed. One of the questions we will address is whether this aggregation mechanism only contributes towards the largest giant component or, conversely, if the entire distribution is affected by the aggregation of smaller extant components (see Section 5.3).

4.3.6 Path length distribution

For the filesystem and history layers, we compute the distribution of the length of the *shortest* paths between all root nodes (i.e., nodes with indegree 0) and all leaves (outdegree = 0) in the given layer.

This is a common measure of network topology and, in the context of collaborative development, gives an idea of the difficulty of finding the *provenance* of an artifact (either "on the fly" as allowed by the graph representation we used, or via pre-computation, as done by WoC, cf. Section 2). Indeed, given a leaf artifact in a given layer, such as a file in the filesystem layer or a commit in the history layer, the computational cost of finding a root node (e.g., a commit containing a given file or an origin containing a commit) will be impacted by the average length of the shortest paths leading from the roots to the leaves.

Computing this metric requires one breadth-first search per root node, making its complexity superlinear in time, which is potentially unfeasible on our entire corpus (without resorting to approximate algorithms and sampling). However, as the degree distribution will show in Section 5.1, commit chains are generally long and degenerate, allowing us to run an exact algorithm on the entire commit layer. For the filesystem layer on the other hand, we resort to uniform sampling of root nodes.

The computation of the shortest paths between nodes of the graph is also necessary to determine other properties related to the connectivity of the graph, such as the betweenness distribution, whose study, feasible at least by limiting itself to a sampling of the graph, was not anticipated in the registered study protocol and is therefore left as a matter for further study.

5 Experimental results

We can now present our findings about the most salient topological properties of the global graph of public software development.

5.1 Degree distributions

Average degrees

The first indicator that is commonly used to indicate how "dense" a graph is is its *average degree*, that is the ratio between the number of edges and the number of nodes it contains. The higher the average degree, the closer the graph is to being fully connected, i.e., having an arc between any two pairs of nodes. In the entire Software Heritage graph, there is an average of 11.0 edges per node, making it a relatively sparser graph in comparison to other large graphs studied in the literature: the graph of public code is around 5 times sparser than the graph of the Web and 15 times sparser than the social graph of Facebook.

Table 4 shows a breakdown of the average degree of each layer of the public code graph, as well as a comparison with various other networks generated by human activities, showing how the graph density lies in the lower range of other real-world networks.

Table 4: Average degree ($|E|/|V|$) for the graph of public software development, its layers, and other large graphs.

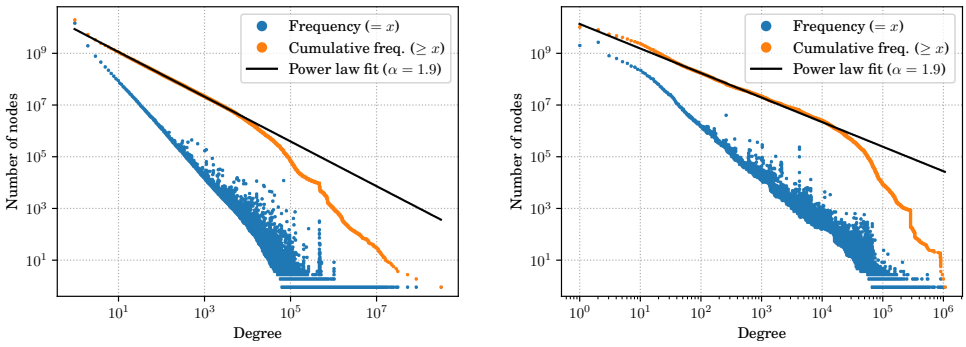
Graph	Description	Average degree
swh-2020-history	**public code — history layer**	1.021
swh-2020-commit	**public code — commit layer**	1.022
swh-2020-hosting	**public code — hosting layer**	3.39
bitcoin-2013 [66]	Bitcoin transactions	6.4
dblp-2011	CS paper citation network	6.8
swh-2020	**public code — full graph**	11.0
swh-2020-filesystem	**public code — filesystem layer**	12.1
twitter-2010 [67]	Twitter followers	35.2
clueweb12 [68]	733 M web pages in English	43.1
uk-2014 [69]	`.uk` web pages	60.4
fb-2011 [70]	Facebook friends	169.0

Because most commits generally only have one parent, with the exception of occasional merge commits which can have more, it is not surprising that the commit layer has a density only slightly larger than 1. The commit chains conform to the general structure of *degenerate trees*. The history layer adds a relatively low number of releases to the commit layer and each release always adds one node and one edge. The hosting layer still has a low density, although around three times denser than the commit layer, because its contents are arranged as a *bipartite graph*: each vertex connects one origin and one snapshot, which limits graph density by construction.

The filesystem layer has a higher density, with an average degree of 12.1. Being the dominant component of the graph, it drives the average of the entire graph up to 11.0. The filesystem layer can be seen as a forest of directory trees, with deduplication of identical subtrees. When a new commit changes a file at a depth of h in the tree, a new tree will be created with h new nodes, corresponding to the path from the root of the original tree to the modified file. All the other nodes of the tree are shared with the previous tree thanks to deduplication, which increases the density of the graph by adding new edges to the shared nodes without creating new nodes. The more deduplication there is in the filesystem layer, the higher we can expect its density to be. While the filesystem layer is the densest part of the graph, since it still is a directed acyclic graph, it is by nature sparser than cyclic or undirected graphs (e.g., social networks).

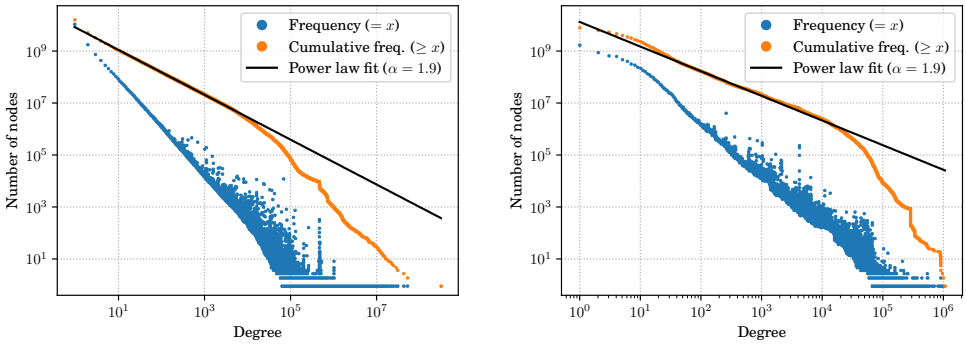
Full graph

Figure 10 shows the frequency and cumulative frequency plots of indegrees and outdegrees of the entire graph in log-log scale. These plots respectively show, for each degree d on the x-axis, the number of nodes that have a degree of exactly d, and a degree of d or more. Using the methodology described in Section 4.3.3, we fit a power law to the tail of the distributions. We obtain a scaling factor of ≈ 1.9 for both distributions for degrees below 10^4. We notice that the indegree distribution is "heavy tailed" for degrees above this value, showing a slow and regular decrease which could indicate a power law tail with scaling factor closer to 3.



(a) Indegrees (b) Outdegrees

Fig. 10: Degree distributions: full graph.



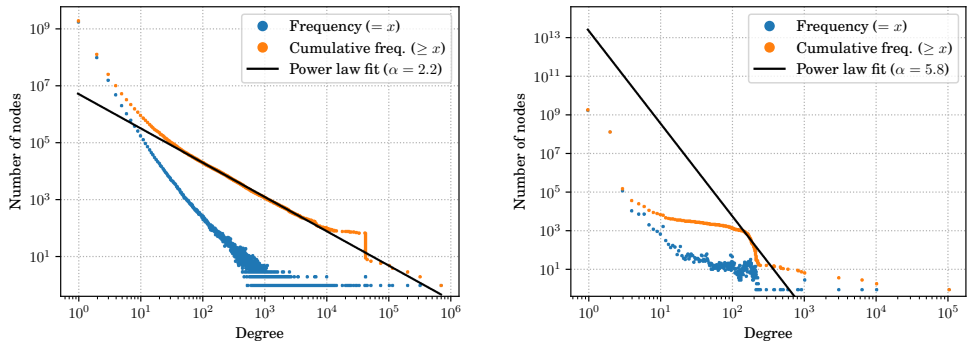
(a) Indegrees (b) Outdegrees

Fig. 11: Degree distributions: filesystem layer.

Filesystem layer

Figure 11 shows the frequency distributions for the filesystem layer. Its similarity with Figure 10 underscores that the filesystem layer largely dominates the frequency distribution of the entire graph.

The outdegree distribution gives an idea of the number of objects present in the archived directories. Most directories seem to contain less than ten entries, with a threshold effect at around ten files after which the frequency drops at a faster pace. The points at the end of the tail are gigantic directories containing millions of entries. These are mostly binary files generated by scripts or the result of user errors (e.g., the



(a) Indegrees ("fork-degrees") (b) Outdegrees ("merge-degrees")

Fig. 12: Degree distributions: commit layer.

largest directory in the corpus[9] contains millions of generated shaders for an amateur video game, in a now deleted branch).

The indegree distribution of the filesystem layer represents the number of directories referencing some specific directory or file content and gives some insights on the deduplication process of directories and contents (as the indegree would always be one with no deduplication). The extreme values at the end of the distribution are peculiar objects that are omnipresent in software repositories: the empty file and the empty directory. Other remarkable outliers are directories containing an empty `.keep` or `.gitkeep` file, which is a common workaround to the fact that the Git version control system cannot store empty directories.

It is also interesting to look at the "bumps" in the indegree distribution that break its regularity, as they are generally not isolated anomalies but consistent deviations centered around a range of values. In the filesystem layer indegree distribution (visible in the raw data available in the replication package), there is a bump at around $d = 92{,}000$ which can be explained by the `CocoaPods/Specs` GitHub repository,[10] a package manager that uses GitHub as a CDN. This repository contains more than 368 k commits, each of which edits a single file in a giant directory containing all the packages of the distribution. Another "bump" can be observed in Figure 11a for values around 454,050.

As in the full graph case, we also observe here in both the indegree and outdegree distributions a transition between two heavy-tailed ranges around degrees 10^4.

Commit layer

The degree distributions for the commit layer are shown in Figure 12. As the parent/children terminology can be confusing when dealing with commits (since *parent* commits are *children* nodes in the DAG), we refer to the indegree of commits as the

[9] This directory could be found in the `lorow/Beabest` GitHub project and was archived in Software Heritage under its SWHID: `swh:1:408689dbd61c9e7888b6556ee5b0ada72935871d`.

[10] https://github.com/CocoaPods/Specs, accessed 2021-09-29.

"fork-degree", that is the number of commits that were based on a specific commit, and to the outdegree as the "merge-degree", i.e., the number of commits that were merged together into a specific commit. The fork-degree distribution is very smooth with no notable threshold effect. This can be explained by common development patterns: forks (in the commit graph) are generally feature branches based on the latest revision in the main development branch, which is generally random, so we do not observe any notable threshold effect or regime change in the distribution.

The merge-degree distribution has a large threshold effect at $d = 2$. The vast majority of commits only have one parent, but occasionally two branches are merged back together, which creates a merge commit with two parents. These are the two most common cases, separated by one order of magnitude ($\approx 10^9$ simple commits, $\approx 10^8$ merge commits). Commits with more than one parent—called "octopus merges" in Git terminology—are exceedingly rare occurrences, not part of standard development workflows, which explains the gap of three orders of magnitude between $d = 2$ and $d = 3$. We expect most of these octopus merges with large degrees to be generated by scripts in very peculiar environments (e.g., large continuous integration pipelines), so the irregularities observed in the tail of the distribution are not particularly surprising.

The distribution for incoming degree values greater than 100 appears to be well described by a regular power-law behavior for more than two decades (between 10^2 and 10^4), with an estimated scaling factor slightly greater than 2. This measure is likely impacted by the presence of many outliers around 5 10^4, corresponding to the gap of the cumulative frequency visible in Figure 12a. If we exclude these outliers, the scale invariant regime would likely cover four decades instead of two.

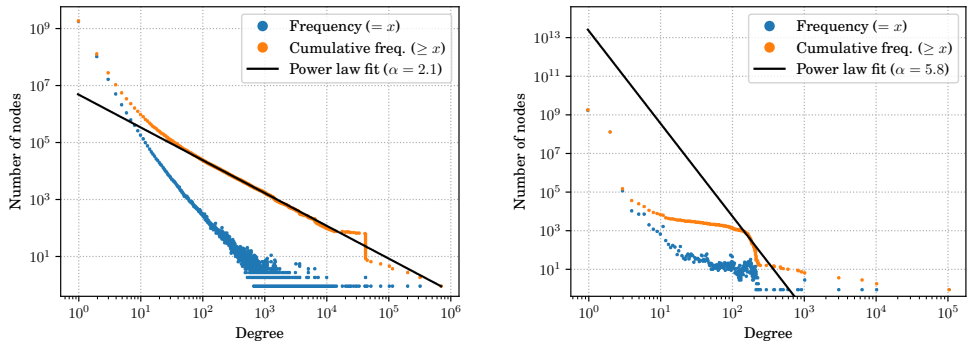
The outdegree distribution is very different, showing a clear first cutoff between 2 and 3 and a second one between 200 and 300, where the cumulative frequency is divided by a factor of 100 (Figure 12b), aside from a few outliers leading to some very large values between 10^3 and 10^5. The largest degree is at $d = 10^6$; it comes from a GitHub repository called `test-commit-many-parents-1m`,[11] which contains two commits linked together by one million of edges. It is the most forked commit in the Software Heritage archive.

History layer

The degree distributions of the history layer, shown in Figure 13, are extremely similar to those of the commit layer as the history layer is largely dominated by commit nodes. However, it is still interesting to look at the indegree distribution of the commit layer from the releases, i.e., the distribution of the number of releases that point to a given commit. It is shown in Figure 14.

There is a noticeable threshold effect between $d = 1$ and the rest of the distribution, attributable to development practices. Releases, or named tags, are generally used to denote specific versions of a software. It makes little sense to have two different versions pointing at a single commit, since there would be no code change to justify the version increment. Occasionally releases can be used to annotate some specific milestones in a project in addition to its current version, so commits pointed by more

[11] https://github.com/cirosantilli/test-commit-many-parents-1m accessed 2021-09-29.



(a) Indegrees (b) Outdegrees

Fig. 13: Degree distributions: history layer.

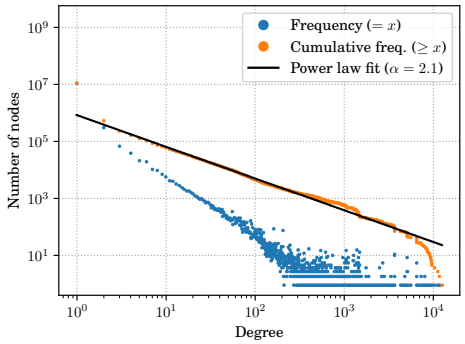


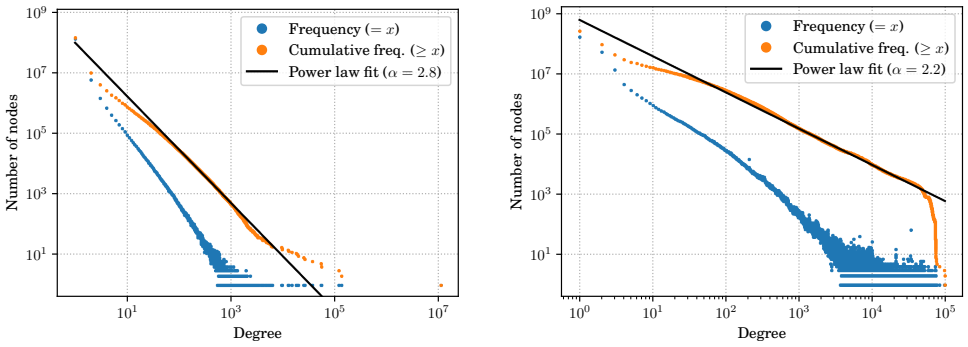
Fig. 14: Indegrees of commits from releases.

than two releases do have some significance, although their importance diminishes rapidly in the distribution.

Hosting layer

The distribution of the hosting layer is shown in Figure 15. Since within the history layer of the graph an origin cannot have ancestors and a snapshot cannot have descendants, the two distributions show very different things. The outdegree distribution describes the number of snapshots associated with each origin. This is not an intrinsic property of development workflows because it is highly dependent on the crawling process of Software Heritage: if a repository changes constantly, but is only visited once every month, the distribution will not capture how frequently the repository is updated, but rather how often the crawlers visit it.

On the other hand, the indegree distribution describes the number of origins associated to each snapshot, that is, the number of "exact forks" ever recorded of a given



(a) Indegrees

(b) Outdegrees

Fig. 15: Degree distributions: hosting layer.

repository. This happens anytime someone makes an exact copy of a repository, for instance by clicking on the "fork" button in GitHub, without then updating it with new commits or branches: a new origin is created, but it points to the same repository state as the first origin.

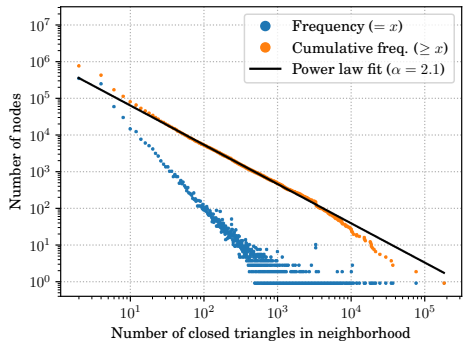
We can notice here a transition of the incoming degrees around 3 10^3 and a cut-off of the outgoing degrees between 5 10^4 and 10^5. We can also underline the presence of outliers in the outdegree distribution between 200 and 300.

5.2 Clustering

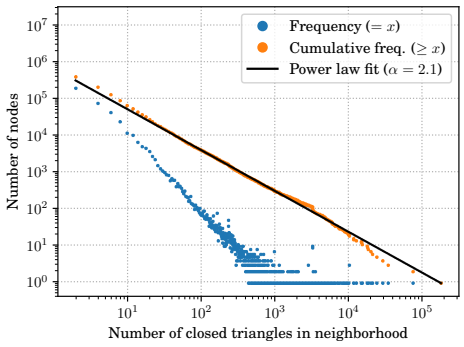
We compute the local clustering distribution on the different layers of the undirected graph, which describes the extent to which the nodes tend to cluster together in tight cliques. In the original DAG, which is directed, the local clustering coefficient is always 0 since a closed triangle corresponds to a cycle which cannot exist in a DAG. However, the distribution of the number of closed triangles in the *undirected* version of the graph, shown in Figure 16a, can be interpreted meaningfully.

Again, the distribution for the full graph is completely dominated by the filesystem layer and cannot be interpreted on its own; each layer has to be looked at individually. In the filesystem layer, a closed triangle corresponds to a file being both in a directory D and in a subdirectory D' of D. A common instance of this happening is when developers copy the contents of a directory in a `backup/` or `old/` directory to take a snapshot of a previous version of the directory, rather than relying on version control system capabilities. Figure 16b shows the frequency of these triangles forming in the filesystem layer, with an apparent scale-invariant regularity.

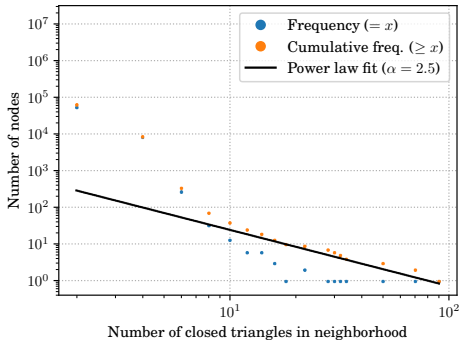
In the commit and history layers a closed triangle corresponds to a merge commit C with two parents A and B, with B being a parent of A. When using the Git version control system, for example, this often happens when merging multiple commits using the "no fast-forward" strategy (`git merge -no-ff`). Here, the distribution (Figure 16c) displays a similar pattern to what we observed in the outdegree distribution of the commit layer (Figure 12b): the two common cases are having one or two



(a) Full graph.



(b) Filesystem layer.



(c) Commit and history layers.

Fig. 16: Clustering distributions, computed on uniform 0.1% node samples of each (sub-)graph.

closed triangles, while having more triangles requires an octopus merge and is relatively rare in most development workflows, which explains the important threshold effect for local clustering value greater than 2.

Being a bipartite graph, the undirected hosting layer cannot contain closed triangles and its local clustering distribution is not shown.

5.3 Connected components

For this part of the analysis, we symmetrize the graph to analyze it as an undirected graph and use a breadth-first traversal to compute the sizes of its weakly connected components (WCC). Table 5 shows a breakdown of the number of components and sizes of the largest components for each layer.

Table 5: Connected components per layer.

Layer	# of WCC	# of isolated nodes	Size of largest WCC	% of nodes in largest WCC
Full graph	33,104,255	22,710,547	18,902,683,142	97.79%
Filesystem layer	46,286,502	79,293	16,565,521,611	97.16%
Commit layer	88,031,649	38,150,369	51,543,944	2.61%
History layer	88,040,059	37,687,392	52,176,239	2.62%
Hosting layer	108,342,722	22,768,378	13,841,855	4.82%

Full graph

In the entire graph, we find a giant component of 18.9 billion nodes, in which a whole 97.8% of the nodes in the graph are reachable from one another by following undirected edges. The size distribution for the full graph shown in Figure 17a clearly indicates the extent to which the largest connected component is an outlier that dominates the entire distribution, being 8,345 times larger than the second largest WCC.

One could wonder whether this high connectivity results from a few highly-connected nodes, like the empty file which is present in millions of repositories. Surprisingly, this turns out not to be the case: repeating the same WCC experiment after removing the top 1 million nodes with the largest indegrees from the graph still yields a giant component of about the same order of magnitude (only about 5% smaller). This implies that the connectivity of the graph is highly resilient and does not depend on the existence of a few high-degree nodes, and that those highly reused software artifacts exist in a graph that is already well-connected without them. This finding is similar to what has been shown for the graph of the Web, where removing pages which are central hubs and have a high PageRank does not significantly reduce the connectivity of the graph [41].

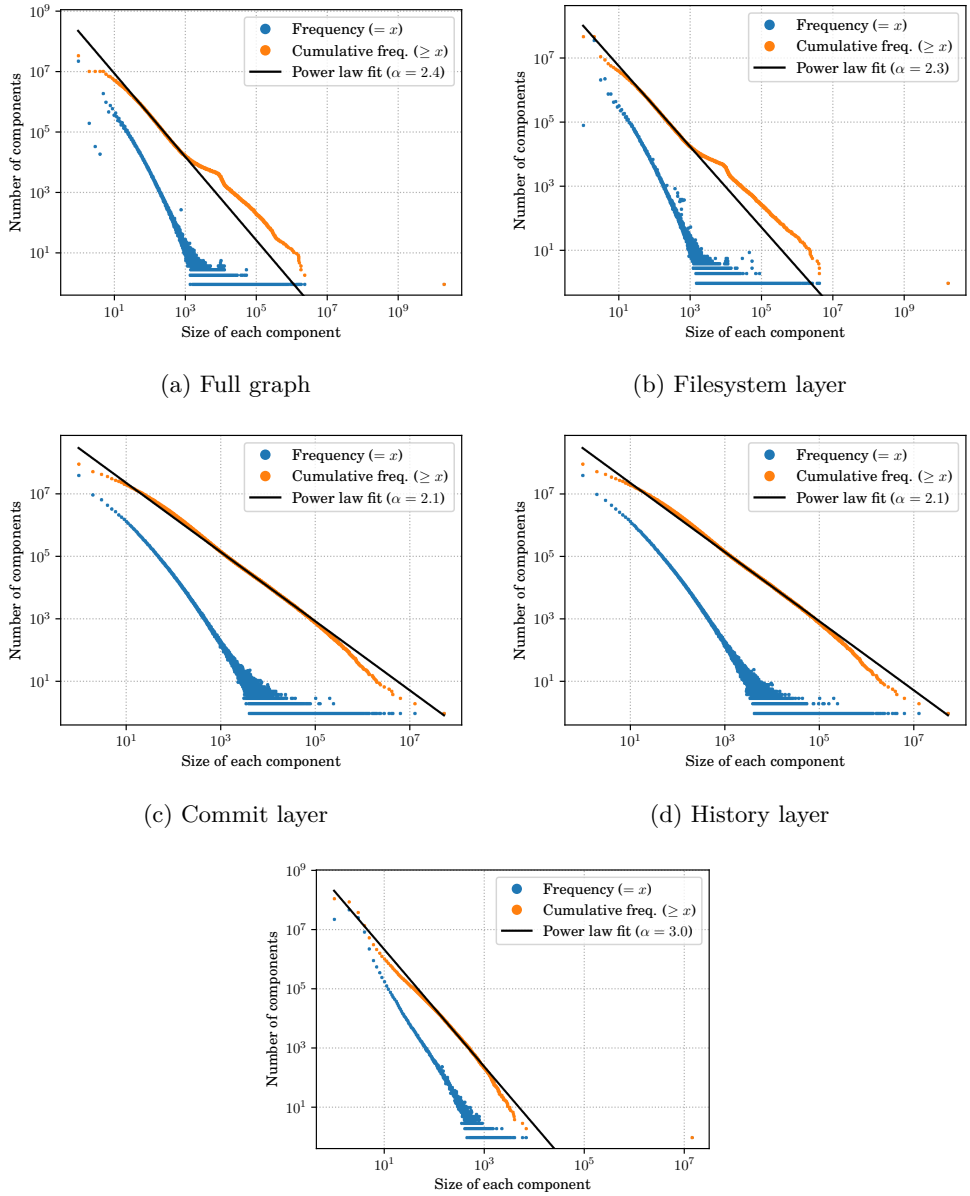
These observations apply similarly to the WCC size distribution for the filesystem layer (shown in Figure 17b) which again dominates the distribution of the full graph.

Commit and history layers

On the other hand the distributions of the commit and history layers shown in Figure 17c and Figure 17d exhibit a very different graph connectivity. The largest component encompasses less than 3% of the graph, which indicates that the history layer can be partitioned into reasonably sized units. Furthermore, an in-depth investigation of this large component reveals that most of the commits it contains belong to various forks of the Linux kernel, which is suspected to be the largest open source software by number of commits across all its different forks. We can infer from this observation that the connected components of the history layer delineate structures of "fork networks" [17] in the graph, by clustering together projects that have a shared development history.

Hosting layer

The largest component in the hosting layer (see Table 5 and Figure 17e) contains a relatively small share of the layer's nodes (around 5%). Yet it has 2,051 times



(a) Full graph

(b) Filesystem layer

(c) Commit layer

(d) History layer

(e) Hosting layer

Fig. 17: Connected components distributions.

more nodes than the second-largest component so may be considered an outlier when compared to the other regular points of the distribution.

The presence of this giant connected component is particularly noteworthy, especially when compared to the history and commit layers, where one expects to observe numerous forked projects sharing at least one revision. In the hosting layer, for two origin nodes to belong to the same giant connected component, the corresponding origins must have been traversed by the Software Heritage's crawlers in the same state at different times. This implies that they were pointing to the same list of branches, with identical names, and in the same state. Essentially, they were pointing to the intrinsic identifiers of the same commits and releases.

This scenario may be indicative of projects that are no longer actively maintained, involving a substantial number of developers/users utilizing a distributed version control tool such as Git. In such instances, the deliberate discontinuation of a project leading to its fragmentation into sub-projects could account for the existence of multiple identical "views" (i.e., an origin linked to the snapshot node) with distinct visit dates recorded by the crawler. This pattern is also observed in projects characterized by a low commit count but frequent forking. A notable example is the GitHub repository `jtleek/datasharing`[12] (accessed on 2022-10-04), which has been forked more than 244,000 times. Some forks of this project like the GitHub repository `1louder/datasharing`[13] (accessed 2022-10-04) only contain commits prior to November 25, 2013. If the Software Heritage crawlers have traversed the original repository at least once between November 25, 2013 and November 6, 2016 (which is very likely), all forks created during this period belong to the same connected component, provided they were visited before any modifications were made.

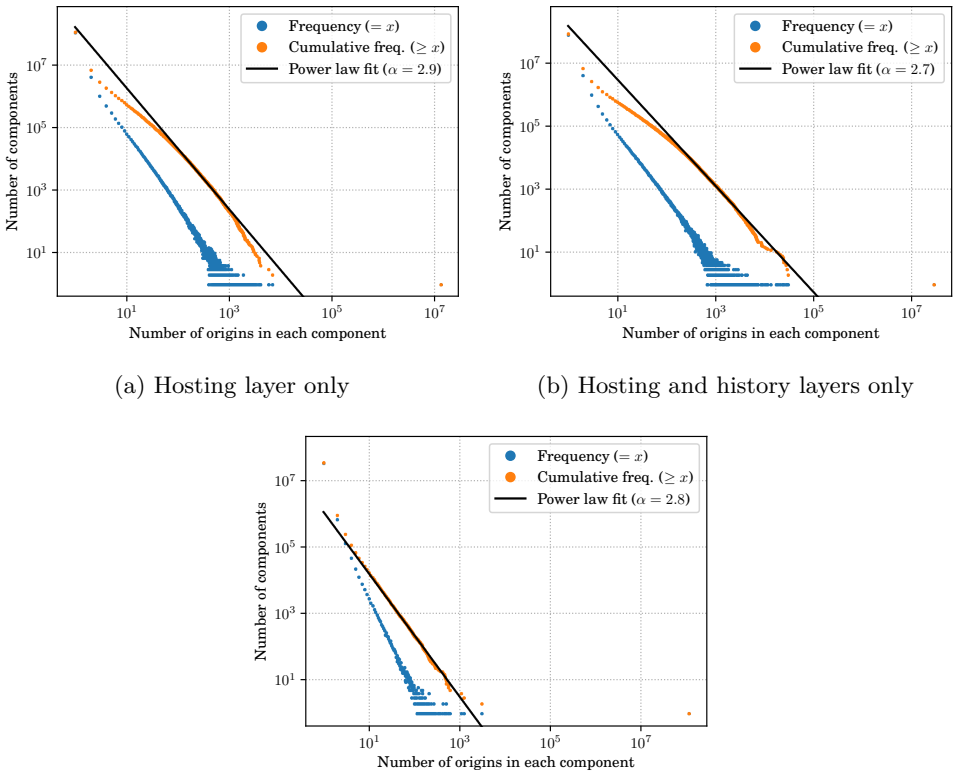
By walking random paths in the giant component, it is possible to see other patterns that could explain the size of this component. One such pattern appears to be that some developers sometimes fork well-known repositories then rewrite their history to replace them with a completely different content. Again, if the Software Heritage crawlers visited this project before modifications were made, it explains its inclusion in the same connected component. Each occurrence of this scenario interlinks unrelated networks of forks, consolidating them within a larger component. This seemingly improbable situation can be quite prevalent, especially if crawlers prioritize a list of newly created projects for initial visits, which includes treating new forks as distinct projects. Subsequent studies could provide a quantitative analysis of the relative impact of the aforementioned mechanisms and potentially unveil additional contributing factors.

Aggregation across layers

In Figure 18 we show the distribution of component sizes measured as the number of software origins they contain. In Figure 19 we show their Kolmogorov-Smirnov distances, as discussed in Section 4.

The first point of interest concerns the gap for size $s = 2$, which corresponds to the percentage of isolated origins (i.e., connected components containing a single

[12] https://github.com/jtleek/datasharing.
[13] https://github.com/1louder/datasharing.

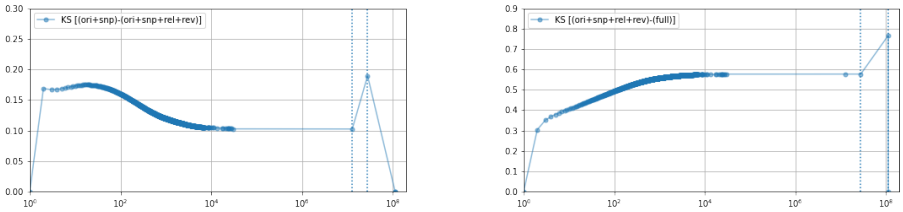


(a) Hosting layer only

(b) Hosting and history layers only

(c) Full graph

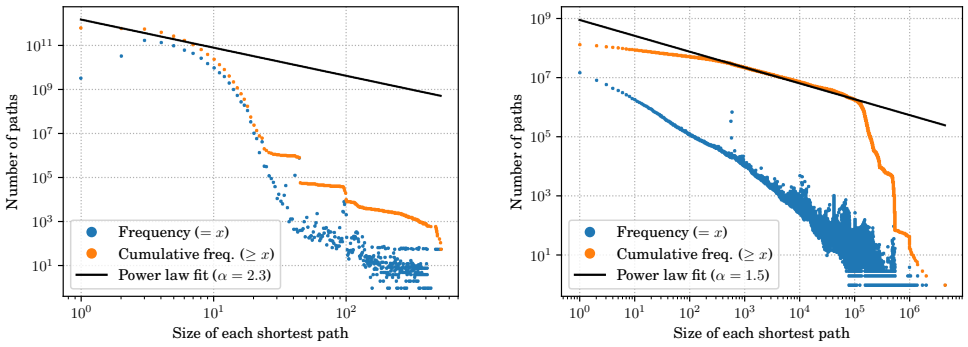
Fig. 18: Connected component size distributions as the number of software origins (e.g., Git repositories, distribution packages, etc.) in each component.



(a) Hosting vs. Hosting+History layers

(b) Hosting+History layer vs. full graph

Fig. 19: Kolmogorov-Smirnov distance between weighted connected component size distribution functions.



(a) Filesystem layer (10% sample)

(b) Commit layer

Fig. 20: Shortest path length distributions.

origin) which are then found in components containing several origins when we include the next layer. Out of a total of 147 million origins, 71.6% are isolated origins in separate connected components in the hosting layer. This number decreases by 17% when merging this layer with the history layer (Figure 19a) and by 30% when taking into account the complete graph (Figure 19b). The sharp decreases at the right of these figures correspond to the number of origins that form the giant component in the initial layer, plus the number of origins that integrate the giant component when the next layer is included (respectively $10.2\% + 8.8\% = 18.9\%$, and $57.8\% + 18.9\% = 76.8\%$) This means that the growth of the giant component does not occur by mere aggregation of components containing isolated origins. This phenomenon is further confirmed by the progressive decrease of the curve at intermediate sizes (left figure), which indicates that the components of these sizes include more origins, previously included in components of smaller sizes. This aggregation phenomenon concerns all layers and all component sizes, limiting the usefulness of attempting to partition the graph based on these two criteria alone.

5.4 Shortest paths

The last topological property we look at is the average length of the shortest paths between root and leaf nodes in the filesystem (Figure 20a) and commit layer (Figure 20b), as defined in Section 4.3.6.

These properties of the various graphs have directly transposable meanings in software development. In the filesystem layer, they correspond to the minimum directory depth at which a given file content can be found on average. In the commit layer, they are the lengths of the commit chains from the first commit of the project to the heads of the branches (again, on average). These are particularly interesting alongside the degree distributions, as they help us understand the shape of the graph given its density.

We saw in Section 5.1 that the filesystem layer was dense, with nodes in close proximity with each other. The distribution obtained in Figure 20a gives an idea

of the depth of the files in the directory trees, which interestingly appear in a very characteristic configuration. Typically, most files are located at a depth of less than 10. The most common depths are 3 (27%), 4 (22%) and 5 (16%). This makes intuitive sense, as the source files which are modified by developers tend to be organized inside (not too deep) directory hierarchies, and rarely reside at the top level.

For values beyond the maximum, two regimes should be distinguished. The first one corresponds to an exponential decay from $\approx 2\ 10^{11}$ to $\approx 2\ 10^{2}$ from lengths 4 and up to lengths of about 40. This very fast decay means that in this value range the probability of observing a file 4 folders down in the filesystem layer is divided by 10. Beyond a depth of 30 or 40 directories, we observe a second regime corresponding to a slower decrease, strongly marked by the presence of numerous outliers at 41, 44, 60, between 94 and 99, ..., as well as for the number of path occurrences where the values 56 and 60 are over-represented and form a clearly distinguishable horizontal line for depths above 100. The existence of these two regimes has direct practical consequences on all measures requiring to efficiently traverse the whole graph. For example, the first traversal of the whole graph produced in the scope of the Software Heritage project – in order to obtain a global analysis identifying the first occurrences of all the directory and file nodes of the graph [34] – was obtained combining the sharding of the revisions based on their intrinsic identifier, a breadth-first search (BFS) on the first 10 depths of the *filesystem* layer, coupled with a depth-first search (DFS) on depths higher than 10. Very long chains, even on small number of paths between revisions and files, made a BFS on all depths inefficient. Inversely, a DFS required traversing all the existing paths without benefiting from deduplication. This empirical optimization finds its explanation in the results we present in this study.

In contrast, Section 5.1 showed that the history layer was sparse with an average degree close to one, indicating that it was mainly constituted of degenerate commit chains. Figure 20b shows the distribution of the lengths of these chains. Outliers are also present in the tail of the distribution, mostly test projects like the GitHub repository `test-many-commits-1m`[14] which contains 2 million commits, but also for intermediate lengths. They justify more detailed studies (outside the scope of this exploratory study) of the outliers of the commit layer, and of their probably important influence on the measure of the scaling factor (especially, for lengths around 400, for particular values 557, 561, 571, 653, 922, ..., see raw data in the replication package). The observed excesses are very significant and can explain the observed variations of the slope of the cumulative distribution and, consequently, induce important biases in the measurement of the scaling factor (cf. Section 6).

The distribution of commit chain lengths shows a cutoff for chains longer than 10^5 commits, but a very large variance and a very large mean, characterized by a scaling factor of about 1.5. We thus expect both the variance and the mean to diverge as the size of the graph increases. Further study is needed to understand the origin of this cutoff, which might be related to such factors as: distribution of project longevity; number of developers involved; or other mechanisms such as changes of version management tools, hosting platforms, and development practices, particularly regarding branch management.

[14] https://github.com/cirosantilli/test-many-commits-1m, accessed 2021-09-29.

Table 6: Frequency distributions of metrics computed on the graph of software development, and their main characteristics.

Metric	Layer	Events	Exponent	X decades	Y decades
Indegrees	Full	1.93×10^{10}	1.86	8.47	10.14
	Filesystem	1.70×10^{10}	1.86	8.47	10.02
	Commit	1.97×10^{9}	2.20	5.84	9.23
	History	1.99×10^{9}	2.14	5.84	9.23
	Hosting	2.87×10^{8}	2.76	7.03	8.16
Outdegrees	Full	1.93×10^{10}	1.94	6.01	9.96
	Filesystem	1.70×10^{10}	1.94	6.01	9.96
	Commit	1.97×10^{9}	5.80	5.00	9.24
	History	1.99×10^{9}	5.80	5.00	9.24
	Hosting	2.87×10^{8}	2.20	4.98	8.22
Connected components	Full	3.31×10^{7}	2.37	10.27	7.35
	Filesystem	4.62×10^{7}	2.25	10.21	7.54
	Commit	8.80×10^{7}	2.10	7.71	7.58
	History	8.80×10^{7}	2.10	7.71	7.57
	Hosting	1.08×10^{8}	2.97	7.14	7.67
Clustering coefficient	Full	1.37×10^{7}	2.06	5.25	7.11
	Filesystem	1.37×10^{7}	2.10	5.25	7.12
	Commit	1.37×10^{7}	2.52	1.95	7.13
	History	1.37×10^{7}	2.52	1.95	7.13
Shortest path	Filesystem	5.86×10^{11}	2.27	2.71	11.20
	Commit	1.72×10^{8}	1.53	6.62	7.58

The value of this scaling factor is significantly less than 2, which is an exception among the other measurements done in this study. Contingent on measurement uncertainty, it would mean that the observed average length of commit chain is only limited by the finiteness of the sample and by phenomena directly related to the dynamics of the projects.

5.5 Summary

Table 6 summarizes all the metrics computed on the graph of software development and its layers, as part of our exploratory study. Each distribution is characterized by its number of observed *events* (i.e., number of objects they represent), the amplitude of its values on both axes expressed in *decades*, as well as the scaling factor (or *exponent*) obtained when fitting a power law. Numerical values are truncated at the second digit.

6 Discussion

6.1 Key findings and their relevance

Our results shed light on some properties of the public software development history which have important implications for empirical research, in particular for large-scale analyses.

6.1.1 Existence and implications of a giant connected component

The first salient characteristic is the large topological disparity between the different layers that constitute the graph, both at the local level and in their global structure. If we break down the graph in three layers—hosting, history, filesystem—we see that they have dramatically different shapes, densities and connectivity.

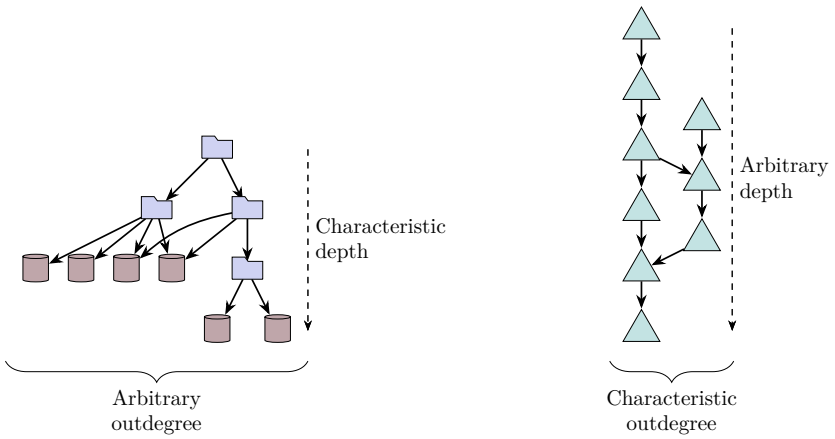
The filesystem layer contains 90% of the nodes and 97% of the edges of the full graph, and thus largely dominates its high-level topological properties. This layer is dense and highly connected, due to the high amount of deduplication of the software artifacts it contains. This high connectivity naturally leads to the existence of a giant weakly-connected component, containing more than 97% of all the files and directories in the graph that are all reachable from each other by simply following directory hierarchy vertices. The degree distributions in the filesystem layer are heavy-tailed and exhibit a clear transition between two different regimes. The directory trees have a characteristic depth with a converging mean, characteristic of a first regime observed for depths below 20, then a second regime closer to the heavy-tailed distributions, the relative weight of which is less than 0.01%.

By contrast, the history layer has almost exactly opposite topological properties. It is sparse and mildly connected, mainly consisting of degenerate commit chains, with relatively low deduplication compared to the filesystem layer. Its largest weakly-connected component is less than 3% the size of the entire graph, which implies that the nodes are well separated within the layer. Commits have a characteristic outdegree, with very few of them merging more than 3 commits together. However, the commits chain lengths show large fluctuations characterized by a very high mean and variance.

This discrepancy between the topological properties of the filesystem and the history layers is illustrated in Figure 21.

Finally, the hosting layer is a bipartite graph containing a small fraction of the nodes in the graph. It is also sparse and minimally connected, with some deduplication for identical forks, which are frequently found in modern collaborative code hosting platforms like GitHub.

A first important practical implication of these findings is that because the filesystem vertices largely aggregate in a giant weakly-connected component, it is not possible to apply strict component separation to easily partition the entire graph in smaller tightly connected clusters in order to perform scale-out computations. However, because the hosting and history layers are sparse, have low connectivity and smaller giant components, it is possible to easily separate them into multiple partitions that can be processed in parallel while retaining the performance advantages of exploiting node locality within them. In so doing, empirical researchers should keep in mind that distributed/parallel processing of *separate development history components* will eventually lead to *common source code* artifacts; they will either have to be re-analyzed multiple times, or will require some distributed caching mechanism to share analysis results across distributed/parallel workers.



(a) Filesystem layer (b) Commit layer

Fig. 21: The filesystem and commit layers have drastically different topological properties: the filesystem layer has a characteristic depth (generally under 20 directories) but no characteristic outdegree (very large mean/variance), whereas the commit layer has a characteristic outdegree (generally two parent commits or less) but no characteristic depth.

6.1.2 Generalization to other graph representations

The file system layer can be viewed as a dense network of highly duplicated software artifacts. The density of this network depends on the chosen representation. For instance, representations derived from deduplication of paths between commits and files aim to find the best trade-off by limiting themselves to file and root directory nodes. It has been previously shown (see [26, Figs. 3, 4, and 6, for instance]) that the distribution of incoming degrees between files and commits is highly similar to what is presented here – exhibiting a heavy tail over many decades, numerous outliers, and more. In the scope of this exploratory study, this observation suggests that heavy-tailed distributions are commonly found in exploitable graph representations at this scale, constituting an intrinsic property of the graph structure of public software development history. While fine-grained characteristics, such as cutoff values and scaling factors (assuming they are well defined), are expected to potentially vary among different graph representations, this aspect is a point of consideration for further analysis.

The presence of heavy-tailed degree distributions alone does not necessarily imply the existence of a giant connected component. Scale-free networks, a subset of heavy-tailed distributions that follow power laws for extreme values, are known for their resilience to random failures, primarily due to the presence of a giant component. However, in the case of an attack targeted at hubs (nodes with the highest degrees), we anticipate a threshold phenomenon beyond which the giant connected component would disappear [71, 72]. In the context of this study, a test was conducted by removing

the 1 million most connected nodes (as mentioned earlier). However, given that this represents only a small fraction of the nodes, it does not provide conclusive evidence of the existence of a threshold beyond which the giant component would disappear. This leaves open the possibility of partitioning strategies where only a few million or tens of millions of nodes would be duplicated.

The observed dual regime with two scaling factors in the filesystem layer (one close to 2, then a second closer to 3) complicates analysis. Without a finer understanding of the nature of this transition, such as the potential role of deduplication of both file and directory nodes, and its dependence on graph representations, the analysis remains challenging. Additionally, the observed variation in the scaling factor with characteristics such as file size ([26, Fig. 3]) suggests that partitioning strategies not solely based on the node degrees criterion might prove to be efficient.

To study the partitioning of the entire graph, we have highlighted in this study an aggregation mechanism that merges connected components of various sizes from distinct layers. Despite prior discussions on different representations and partitioning strategies within the same layer of the graph, implementing a measure that encompasses several layers presents a greater challenge. There is no guarantee that an efficient partitioning, if it exists on a layer-by-layer basis, will also hold for two layers together. For a more detailed understanding of the mechanisms at play, it will be necessary, for example, to study degree correlations between nodes of different types.

6.1.3 Implications regarding feasibility of large-scale measurements

Another key consideration for empirical software engineering studies is that the distributions examined in this paper often exhibit slow variations across multiple decades (see Table 6), in line with the characteristics of heavy-tailed distributions. The observed maxima of topological properties directly influence the feasibility of certain measurements at this scale, as detailed below.

Those distributions for which the values of the scaling factor are between 2 and 3 correspond to (sub)networks that are in a *scale-free* regime. This has a direct impact on graph processing algorithms whose complexity depends on intrinsic properties of the network structure. Distributions in the scale-free regime have a defined mean and an undefined standard deviation as the sample size increases [61]. Since the graph of public software development grows (exponentially) over time, the value of the scaling factor α of each distribution can be used to forecast how various graph metrics are expected to increase with the number of nodes. For power-law distributions, the maximums of a distribution characterizing a system of size N will tend to vary as $N^{1/(\alpha-1)}$. If a distribution has an α value of 2, its expected maximum will grow linearly; if $\alpha = 3$, then it will grow proportionally to $\sqrt{N}$.

While an algorithm-by-algorithm discussion is beyond the scope of this study, we have mentioned the practical consequences for optimizing studies that combine BFS and DFS. The computation of local clustering coefficient distributions in this study is based on an optimized implementation of the simplest algorithm, with a complexity of at most $\mathcal{O}(Nd_{max}^2)$, where N is the number of nodes in the graph and d_{max} is the maximal degree. The existence of large ranges of values over several decades, and the likely variation of maximums based on scaling factors, will therefore introduce a

variation in $\mathcal{O}(N^2)$ – or even in $\mathcal{O}(N^3)$ – for the simplest algorithm calculating the clustering coefficient as a function of the graph size N (see [65] for details and the used approximation).

This challenging situation may impede the feasibility of certain large-scale measurements. It can be addressed by using a graph representation more tailored to the computations at hand. Note however that a different representation may simultaneously reduce the number of nodes and increase the number of edges or result in a partial loss of information; although a representation may be more suitable for certain experiments, we cannot rule out, at this stage, that this won't come at the expense of poorer performance for other experiments. Comparing the efficiency of different graph representations requires benchmarks beyond the scope of the current study.

6.1.4 Existence and implications of outliers

Outliers are also a common feature of most of the observed distributions, not only located at the end tail of the distributions but also at early values. It can be particularly difficult to disentangle rare events, i.e., extreme events at the end of the tail created due to legitimate development practices, and outliers, i.e., unusual events that are disconnected from the underlying distribution. In the scope of this study, we have discussed the potential influence of outliers, such as when assessing the scaling factors in the distribution of shortest paths in the commits layer (Figure 20b). This consideration likely extends to the size distribution of connected components on the filesystem layer (Figure 17b) and the entire graph (Figure 17a).

This highlights the need to systematically document the methods used to filter outliers in extant and future empirical software engineering studies (e.g., to build subgraphs or derived graphs), as these can have a significant impact on the results and their interpretation.

6.2 Threats to validity

6.2.1 Internal validity

Computational complexity challenges

This study is an implementation of the preregistered phase 1 study protocol described in [32]. Starting from the Software Heritage graph dataset as a raw corpus, we have analyzed it using the algorithms and statistical tools described in the phase 1 protocol, without any significant change to the methodology. However, at the time of designing the phase 1 protocol, we underestimated the execution time of two algorithms, which we were therefore not able to run on the entire graph.

According to our estimates, the path length distribution of the filesystem layer would have taken around two months to compute on an expensive server, exceeding our available resources. Thus, we restricted ourselves to compute path length distribution on a uniform sample of 10% of the nodes ($n = 2$ billion nodes) of the filesystem layer. On the other hand, we were able to compute the path length distribution in full on the entire commit layer in around 3 hours, without having to resort to sampling.

We also underestimated the time required to compute the clustering coefficient of the entire corpus as an undirected graph, which would have taken several years to

complete on the resources we had at our disposal. Instead we have analyzed a uniform sample of 0.1% of the entire graph ($n = 20$ million nodes), which is in line with the sample sizes of clustering coefficient estimates in other analyzes of large networks [65]. This misestimation stems from the fact that it is not possible to check in $O(1)$ whether some node is in the neighborhood of another node using the adjacency data structures of the compressed graph.

As this study was exploratory in nature with no preconstructed hypotheses or comparable experiences, we had anticipated the possibility of having to resort to sampling in the protocol [32, Section 7]. Also, the graph subsamples we ended up analyzing are on par with other analyses of large networks. Hence, we do not consider this sampling to be a significant threat to experiment validity.

Simple graph vs. multigraph

As discussed in Section 3, an appropriate metamodel for the global VCS graph of public code are *property graphs*, where both nodes and edges are associated to properties that are meaningful from the point of view of software engineering. For example, edges between directories and file blobs have a property denoting local path names, and edges between commits have a property denoting the branching order in merges. In graph theory a property graph is not a *simple graph*, where at most one edge can exist between two vertices, but a *multigraph*, where multiple edges can be incident to the same two vertices. Consider for instance the case of a single directory `"test/"` containing twice the same file blob (e.g., the empty file), under two different names (e.g., `"a"` and `"b"`). The simple graph view of such a VCS will represent it as a single edge `"text"` $\rightarrow$ *empty file*, while the multigraph view will represent it as *two* edges between the same nodes.

`swh-graph` implements the compression of adjacency lists, known for its effectivenesss with the graph of the Web; it addresses previously identified bottlenecks of the Software Heritage dataset [26, 73], specifically: (a) the removal of the $file/entry\_file$ and $directory/entry\_dir$ subtypes existing in the SWH project reference model; (b) the use of an adjacency list for the graph and its transpose; and (c) a unified representation that connects nodes of different types[15].

In the version of `swh-graph` used for this work, certain relationships between nodes are not preserved. The effect of this deduplication is particularly notable for edges to and from snapshot nodes (hosting layer), as well as for some edges between directory and file nodes. This results in a reduction of 75% and 15% of edges whose source or target is a snapshot node respectively. From a methodological standpoint, these deduplications are not very concerning as the removed multiple edges stem from crawling processes, reflecting the number of visits to these origins and how often crawlers have 'seen' these snapshots.

The results reported in Section 5 therefore concern the simple graph view of the global VCS graph rather than the multigraph. This should be taken into account when interpreting results; for instance, the outdegree distribution of a directory should be

[15]Note that any approach that overcomes these bottlenecks should be able to reproduce the obtained results.

interpreted as the number of *unique objects* present in a directory, and not the number of directory and file entries. The impact of edge compression on the number of edges is limited overall, amounting to a few percentage points (as we have quantified using integrity criterion #2, see the corresponding script in the dataintegrity notebook of the replication package). Note however that this difference – which cannot be attributed to crawling processes as with the hosting layer – may seem small or negligible compared to the total number of nodes or edges, but is not necessarily so when compared to the weight of the tails of the distributions, the characteristics of which have been extensively discussed in the context of this study. The log-log scale produces a magnifying effect, as evident in Figure 14, where one can observe that the tail of the distribution for incoming degrees with a degree greater than or equal to 10 represents 10^5 nodes, approximately 1% of the total of 10^7 edges. As a result, it cannot be ruled out that the results presented in this study may be impacted.

In practice, we have not attempted to measure the impact of edge compression on the studied distributions, but it is likely to have a quantitative impact too on some of the results presented in this study (such as the estimates of the scaling factors, ranges and means, or properties of the shortest paths for instance), without invalidating them qualitatively.

Recent versions of the graph compression pipeline now support multigraphs [59], making it possible to characterize both the graph and multigraph views of the global VCS graph, and better quantify their differences. Doing so is left as future work.

Multiple representations

As recalled in Section 2, different representations of the global VCS graph exist. Each representation may be more amenable to a specific analysis than the others. The practical choice of a representation involves well-known trade-offs between the complexity of certain operations (such as querying the first or last version in the history) and the resources required to store the software development history. In addition, the tools available to carry said analysis may impose constraints on the representation and may necessitate simplifying/compressing the representation to lift some bottlenecks, in a way that is potentially not reversible. Both of these points have started to be illustrated by the simple graph vs. multigraph discussion of the previous section.

Particular attention has been given to discussing the potential transferability of the results obtained in this study to other representations of the global VCS graph, or in other words to the intrinsic nature of the results. Some of our results are indeed intrinsic. An example is the distribution of the shortest paths between all files found in a commit which holds true regardless of the chosen filesystem layer representation (as long as the definition of path length is the same). Some other results are impacted by the edge vs. multiedge limitation of the previous section – as indegree/outdegree distributions – and may therefore not be regarded as really intrinsic, even though some version of them probably hold regardless of representation at least in a qualitative manner. Further work will be needed to fully assess the extent to which the results obtained in this study reflect the intrinsic nature of the global VCS graph of public code.

In the same vein, the layer structure used here has its drawbacks as it may be insufficient to study some interesting topological properties, independently of their dependence on a specific graph representation. For example, the indegree/outdegree distributions aggregate edges from file and directory nodes. For directory nodes, outdegrees include both outgoing edges pointing to files and those pointing to directories. This aggregation by layer, rather than by type of edges, is likely to have influenced some of the results we obtained.

We would like to emphasise here that using a protocol based on the prior publication of a Registered Report was key for this study to better account for the existence of diverse representations of the same graph. It enhanced the consideration and understanding of the dependence of the study results on the chosen representation and lead to a more detailed and relevant analysis of the intrinsic properties of the public software development history.

Quality of "real-world" datasets

Another potential internal threat is the validity of the dataset itself. Because this study is the first of its kind ever performed on the graph of software development, there is no existing dataset to which its properties can be cross-compared. We performed a series of quality and integrity checks (documented in the replication package) to ensure that the properties were consistent to our own expectations or to the raw original graph, which allowed us to iteratively find and correct errors in both the dataset export pipeline and the implementation of our experiments.

Note first that our experiments are based on the Software Heritage data export dated 2020-12-15. This constitutes a minor deviation from what was announced in the execution plan of the Registered Report. Indeed the latest export that was available at the time we started our experiments, dated 2018-09-25, turned out to have a corrupted snapshot layer and we had to restart with a more recent export.

But even dataset version 2020-12-15 suffers from some data quality issues. In particular there are nodes without ancestors with, for example, 1% of the revisions not linked to any snapshot or release (see Criterion #4 in the replication package and associated script, as well as a list of such nodes). The replication package includes a discussion of potential causes and ways to verify them.

The Software Heritage dataset is a *real-world* dataset (as opposed to synthetic datasets), and it is unavoidable to have this kind of inconsistency. Moreover, it is not possible to guarantee that the checks we performed were exhaustive, especially since we are handling a data set of several billion nodes. Although we cannot rule it out, we have no evidence that these data quality issues have a significant impact on the results presented in this study.

6.2.2 External validity

While our data corpus is the largest dataset of software development history and *aims* to be as exhaustive as materially possible, it remains a subsample of the full software commons, and as such the way it is constructed is a source of various potential biases.

Exhaustiveness of VCS and package managers

Software Heritage covers the most popular DVCS (Git, Mercurial, Subversion, ...) as well as distribution and language-specific packages (`dpkg-source`, Nix, Python, NodeJS, ...), and regularly adds support for new systems. The Software Heritage snapshot we used as dataset did not cover less commonly used systems, like Bazaar, Darcs and CVS for example. If software development patterns on these platforms are significantly different, this study cannot properly capture them in a representative manner.

Exhaustiveness of data sources

Likewise, the representativeness of the study is limited by the extent of the data source coverage of Software Heritage. The archive contains the main centralized software forges and package repositories (GitHub, GitLab, Bitbucket, PyPI, Debian, NixOS, ...), as well as instances of decentralized forges (e.g., various self-hosted GitLab or Phabricator instances). As the archive cannot realistically cover the long tail of smaller self-hosted forges, this is another source of popularity bias in the input dataset.

Archival process

The process of listing data sources and loading repositories and software packages in the Software Heritage archive is heterogeneous across data sources, which can skew the representativeness of the data. Data sources are crawled at varying frequencies depending on multiple factors: for instance, some forges support subscription-based APIs that allow the crawlers to archive repositories as soon as a change is pushed to them. Some repositories are considered more critical to software infrastructures and are crawled daily. Other scheduling heuristics are in place to maximize resource usage efficiency of data crawlers. Overall, this means that the topology of the "hosting" layer is endogenous to the archival process, rather than being an intrinsic property of software development happening there. This is mainly reflected in the number of snapshots that neighbor a given origin, since more frequent crawling generally produces more snapshots.

Non-software data

We acknowledge the habit of developers to use software development platforms and hosts for non-software projects (e.g., collaborative writing, websites, open datasets, art assets, etc.). However, we expect software development to be the dominant content hosted in these platforms. We also assume that the results of this work would be most useful for researchers when applied to similar corpuses, which would contain the same kind of non-software data, as opposed to carefully curated ones.

7 Conclusion

This article describes the first exploratory study on the intrinsic structure of the version control system (VCS) graph of public software development. We use the Software Heritage graph dataset as a corpus, which is the largest public archive of source code and gives us a unified view of public software development. This graph encodes the

fundamental relationships between the 20 billion software artifacts it contains (source code files and directories, commits, releases, repository states) and materializes them as an immense complex network of VCS data.

The ambition of this study is to lay the groundwork for future large-scale analyses in the graph of development history by comprehensively documenting its structure, and to draw a few key implications for empirical software mining. For that purpose, we systematically analyzed robust and classic measures of network topology (indegree and outdegree distributions, connected component sizes, shortest path lengths, and clustering coefficient) and discussed their intrinsic or non-intrinsic nature.

We reached the following conclusions:

- There is a large disparity between the topological properties of the different layers in the graph. The three main layers (filesystem, history, and hosting) have dramatically different shapes, densities and connectivities, and they take up very uneven shares of the total size of the graph. The properties of the full graph are largely dominated by those of the filesystem layer, which represents the vast majority of nodes (90%) and edges (97%) in the graph.
  - The filesystem layer is dense, with an average degree of ≈ 12, and highly connected with more than 97% of the nodes aggregated in a single giant connected component. Files have a characteristic depth of less than 20 directories.
  - The history layer is sparse, with an average degree of ≈ 1, presenting itself as a collection of degenerate strings of commits. It has a comparatively lower connectivity, with the largest connected component containing 3% of its nodes.
- Most of the distributions are heavy-tailed, which can jeopardize the feasibility of some algorithms in future studies as the VCS graph grows exponentially.
- The entangled nature of the graph as a whole, highlighted by the multiscale aggregation process of the connected components, makes it difficult to use naive sharding approaches for scale-out processing of its full extent or large shares of it, as it cannot be easily partitioned into clusters of connected components. The history layer itself, having a lower connectivity, can be partitioned in reasonably sized groups of connected clusters.

There are several ways in which the present study could be refined and expanded in the future. In particular, we could investigate other topological properties (such as the betweenness distribution, not originally planned in the registered study protocol so left out here), explore the implications of the results presented here on such aspects as dynamic evolution and maintenance, and further delve into the pros and cons of the different known models and representations to determine the intrinsic structure of the graph of public software development history.

Acknowledgements

The authors would like to thank the reviewers for their insightful comments, and S. Petitjean for his thorough review of the revised version of the manuscript and his suggested amendments, that led to a much improved and focused version of this work.

Declarations

Funding

Not applicable.

Ethical approval

Not applicable.

Informed consent

Not applicable.

Re-use of material

Some of the results presented in this study appeared in preliminary form in the PhD thesis of Antoine Pietri [59].

Data Availability Statement

A replication package for this study is available on Zenodo: https://zenodo.org/records/15038707. It contains the raw data of the experiment results, as well as detailed instructions for how to reproduce the experiments. We also provide several Jupyter notebooks to plot the graphs shown in Section 5 and run quality/integrity tests discussed in Section 6.2.

Conflict of Interest

The results of this study are based on the use of a representation of the Public Software Development History made available by Software Heritage. At the time this study was initiated, the authors were members of Software Heritage.

Clinical Trial Number

Not applicable.

References

[1] Feller, J., Fitzgerald, B., *et al.*: Understanding Open Source Software Development, (2002). Addison-Wesley London

[2] Begel, A., Herbsleb, J.D., Storey, M.-A.: The future of collaborative software development. In: Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work Companion, pp. 17–18 (2012)

[3] Hassan, A.E.: The road ahead for mining software repositories. In: Frontiers of Software Maintenance, 2008. FoSM 2008., pp. 48–57 (2008). IEEE

[4] Spinellis, D.: Version control systems. IEEE Software **22**(5), 108–109 (2005)

[5] Kagdi, H., Collard, M.L., Maletic, J.I.: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. Journal of software maintenance and evolution: Research and practice **19**(2), 77–131 (2007)

[6] Demeyer, S., Mens, T.: Software Evolution, (2008). Springer

[7] Pan, W., Li, B., Ma, Y., Liu, J.: Multi-granularity evolution analysis of software using complex network theory. Journal of Systems Science and Complexity **24**(6), 1068–1082 (2011)

[8] Wittern, E., Suter, P., Rajagopalan, S.: A Look at the Dynamics of the JavaScript Package Ecosystem. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pp. 351–361 (2016)

[9] Howison, J., Conklin, M., Crowston, K.: Flossmole: A collaborative repository for FLOSS research data and analyses. International Journal of Information Technology and Web Engineering (IJITWE) **1**(3), 17–26 (2006). https://doi.org/10.4018/jitwe.2006070102

[10] Gao, Y., VanAntwerp, M., Christley, S., Madey, G.: A research collaboratory for open source software research. In: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development, FLOSS'07 (2007). IEEE

[11] Van Antwerp, M.V., Madey, G.: Advances in the SourceForge research data archive. In: Workshop on Public Data About Software Development (WoPDaSD) at The 4th International Conference on Open Source Systems, Milan, Italy (2008)

[12] Mockus, A.: Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In: Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009, pp. 11–20 (2009). https://doi.org/10.1109/MSR.2009.5069476. IEEE Computer Society. https://doi.org/10.1109/MSR.2009.5069476

[13] Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 422–431 (2013). IEEE Press

[14] Di Cosmo, R., Zacchiroli, S.: Software Heritage: Why and how to preserve software source code. In: Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017 (2017). https://hal.archives-ouvertes.fr/hal-01590958/

[15] Ma, Y., Dey, T., Bogart, C., Amreen, S., Valiev, M., Tutko, A., Kennard, D., Zaretzki, R., Mockus, A.: World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data. Empir. Softw. Eng. **26**(2),

22 (2021). https://doi.org/10.1007/s10664-020-09905-9

[16] Robles, G., González-Barahona, J.M.: A comprehensive study of software forks: Dates, reasons and outcomes. In: Hammouda, I., Lundell, B., Mikkonen, T., Scacchi, W. (eds.) Open Source Systems: Long-Term Sustainability - 8th IFIP WG 2.13 International Conference, OSS 2012, Hammamet, Tunisia, September 10-13, 2012. Proceedings. IFIP Advances in Information and Communication Technology, vol. 378, pp. 1–14 (2012). https://doi.org/10.1007/978-3-642-33442-9_1. Springer. https://doi.org/10.1007/978-3-642-33442-9_1

[17] Pietri, A., Rousseau, G., Zacchiroli, S.: Forking without clicking: on how to identify software repository forks. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 277–287 (2020). https://doi.org/10.1145/3379597.3387450

[18] Meusel, R., Vigna, S., Lehmberg, O., Bizer, C.: The graph structure in the web–analyzed on different aggregation levels. The Journal of Web Science **1** (2015)

[19] Ugander, J., Karrer, B., Backstrom, L., Marlow, C.: The anatomy of the facebook social graph. arXiv preprint arXiv:1111.4503 (2011)

[20] Myers, S.A., Sharma, A., Gupta, P., Lin, J.: Information network or social network? the structure of the twitter follow graph. In: Proceedings of the 23rd International Conference on World Wide Web, pp. 493–498 (2014)

[21] Albert, R., Barabási, A.-L.: Statistical mechanics of complex networks. Reviews of modern physics **74**(1), 47 (2002)

[22] Coutinho, B.C., Hong, S., Albrecht, K., Dey, A., Barabási, A.-L., Torrey, P., Vogelsberger, M., Hernquist, L.: The Network Behind the Cosmic Web. arXiv (2016)

[23] Concas, G., Marchesi, M., Pinna, S., Serra, N.: Power-laws in a large object-oriented software system. IEEE Transactions on Software Engineering **33**(10), 687–708 (2007)

[24] Louridas, P., Spinellis, D., Vlachos, V.: Power laws in software. ACM Trans. Softw. Eng. Methodol. **18** (2008)

[25] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 92–101 (2014). ACM

[26] Rousseau, G., Di Cosmo, R., Zacchiroli, S.: Software provenance tracking at the scale of public source code. Empirical Software Engineering **25**(4), 2930–2959 (2020). https://doi.org/10.1007/s10664-020-09828-5

[27] Flint, S.W., Chauhan, J., Dyer, R.: Escaping the time pit: Pitfalls and guidelines for using time-based git data. In: 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021, pp. 85–96 (2021). https://doi.org/10.1109/MSR52588.2021.00022. IEEE. https://doi.org/10.1109/MSR52588.2021.00022

[28] Sasaki, Y., Yamamoto, T., Hayase, Y., Inoue, K.: Finding file clones in FreeBSD Ports Collection. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), pp. 102–105 (2010). IEEE Press

[29] Abramatic, J.-F., Di Cosmo, R., Zacchiroli, S.: Building the universal archive of source code. Communications of the ACM **61**(10), 29–31 (2018). https://doi.org/10.1145/3183558

[30] Boldi, P., Pietri, A., Vigna, S., Zacchiroli, S.: Ultra-large-scale repository analysis via graph compression. In: SANER 2020: The 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (2020). IEEE

[31] Lopes, C.V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajnani, H., Vitek, J.: DéjàVu: a map of code duplicates on GitHub. Proceedings of the ACM on Programming Languages (OOPSLA) (2017)

[32] Pietri, A., Rousseau, G., Zacchiroli, S.: Determining the intrinsic structure of public software development history. In: MSR 2020: The 17th International Conference on Mining Software Repositories (2020). https://doi.org/10.1145/3379597.3387506. IEEE. OSF registration available online at: https://osf.io/7r2w4

[33] Pietri, A., Spinellis, D., Zacchiroli, S.: The Software Heritage graph dataset: public software development under one roof. In: Storey, M.D., Adams, B., Haiduc, S. (eds.) Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada., pp. 138–142 (2019). IEEE / ACM. https://dl.acm.org/citation.cfm?id=3341907

[34] Rousseau, G., Di Cosmo, R., Zacchiroli, S.: Growth and duplication of public source code over time: Provenance tracking at scale. Technical report, Inria (2019). https://hal.archives-ouvertes.fr/hal-02158292

[35] Mockus, A., Spinellis, D., Kotti, Z., Dusing, G.J.: A complete set of related git repositories identified via community detection approaches based on shared commits. In: MSR '20: 17th International Conference on Mining Software Repositories, pp. 513–517 (2020). https://doi.org/10.1145/3379597.3387499. ACM. https://doi.org/10.1145/3379597.3387499

[36] Alexandru, C.V., Panichella, S., Proksch, S., Gall, H.C.: Redundancy-free analysis of multi-revision software artifacts. Empirical Software Engineering **24**(1), 332–380 (2019). https://doi.org/10.1007/s10664-018-9630-9

[37] Trujillo, M.Z., Hébert-Dufresne, L., Bagrow, J.P.: The penumbra of open source: projects outside of centralized platforms are longer maintained, more academic and more collaborative. ArXiv preprint **abs/2106.15611** (2021)

[38] Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**(6684), 440–442 (1998). https://doi.org/10.1038/30918. Accessed 2021-12-05

[39] Barabási, A.-L., Albert, R.: Emergence of Scaling in Random Networks. Science **286**(5439), 509–512 (1999). https://doi.org/10.1126/science.286.5439.509. Publisher: American Association for the Advancement of Science. Accessed 2021-12-05

[40] Albert, R., Jeong, H., Barabási, A.-L.: Diameter of the World-Wide Web. Nature **401**(6749), 130–131 (1999). https://doi.org/10.1038/43601. Accessed 2021-12-05

[41] Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., Wiener, J.: Graph structure in the web. Computer networks **33**(1-6), 309–320 (2000)

[42] Boldi, P., Vigna, S.: The WebGraph framework I: compression techniques. In: Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E. (eds.) Proceedings of the 13th International Conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004, pp. 595–602 (2004). https://doi.org/10.1145/988672.988752. ACM. https://doi.org/10.1145/988672.988752

[43] Boldi, P., Vigna, S.: The WebGraph framework II: codes for the world-wide web. In: 2004 Data Compression Conference (DCC 2004), 23-25 March 2004, Snowbird, UT, USA, p. 528 (2004). https://doi.org/10.1109/DCC.2004.1281504. IEEE Computer Society. https://doi.org/10.1109/DCC.2004.1281504

[44] LaBelle, N., Wallingford, E.: Inter-package dependency networks in open-source software. arXiv preprint cs/0411096 (2004)

[45] Abate, P., Di Cosmo, R., Boender, J., Zacchiroli, S.: Strong dependencies between software components. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 89–99 (2009). IEEE

[46] Maillart, T., Sornette, D., Spaeth, S., von Krogh, G.: Empirical tests of zipfs law mechanism in open source linux distribution. Physical Review Letters **101**(21), 218701 (2008)

[47] Myers, C.R.: Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. Physical Review E **68**(4), 046116 (2003)

[48] Valverde, S., Solé, R.V.: Hierarchical small worlds in software architecture. arXiv

preprint cond-mat/0307278 (2003)

[49] Wen, L., Kirk, D., Dromey, R.G.: Software systems as complex networks. In: Zhang, D., Wang, Y., Kinsner, W. (eds.) Proceedings of the Six IEEE International Conference on Cognitive Informatics, ICCI 2007, August 6-8, Lake Tahoe, CA, USA, pp. 106–115 (2007). https://doi.org/10.1109/COGINF.2007.4341879. IEEE Computer Society. https://doi.org/10.1109/COGINF.2007.4341879

[50] Bhattacharya, P., Iliofotou, M., Neamtiu, I., Faloutsos, M.: Graph-based analysis and prediction for software evolution. In: Proceedings of the 34th International Conference on Software Engineering. ICSE '12, pp. 419–429 (2012). IEEE Press

[51] Concas, G., Marchesi, M., Pinna, S., Serra, N.: Power-Laws in a Large Object-Oriented Software System. IEEE Transactions on Software Engineering **33**(10) (2007)

[52] Chaikalis, T., Chatzigeorgiou, A.: Forecasting Java Software Evolution Trends Employing Network Models. IEEE Transactions on Software Engineering **41**(6) (2015)

[53] Fortuna, M.A., Bonachela, J.A., Levin, S.A.: Evolution of a modular software network. Proceedings of the National Academy of Sciences **108**(50) (2011)

[54] Singh, P.V.: The small-world effect: The influence of macro-level properties of developer collaboration networks on open-source project success. ACM Trans. Softw. Eng. Methodol. **20**(2), 6–1627 (2010). https://doi.org/10.1145/1824760.1824763

[55] Hassan, A.E., Holt, R.C.: The small world of software reverse engineering. In: 11th Working Conference on Reverse Engineering, WCRE 2004, pp. 278–283 (2004). https://doi.org/10.1109/WCRE.2004.37. IEEE Computer Society. https://doi.org/10.1109/WCRE.2004.37

[56] Angles, R.: The property graph database model. In: Olteanu, D., Poblete, B. (eds.) Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018. CEUR Workshop Proceedings, vol. 2100 (2018). CEUR-WS.org. http://ceur-ws.org/Vol-2100/paper26.pdf

[57] Bonifati, A., Fletcher, G.H.L., Voigt, H., Yakovets, N.: Querying Graphs. Synthesis Lectures on Data Management, (2018). https://doi.org/10.2200/S00873ED1V01Y201808DTM051. Morgan & Claypool Publishers. https://doi.org/10.2200/S00873ED1V01Y201808DTM051

[58] Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara,

California, USA, August 16-20, 1987, Proceedings. Lecture Notes in Computer Science, vol. 293, pp. 369–378 (1987). https://doi.org/10.1007/3-540-48184-2_32. Springer. https://doi.org/10.1007/3-540-48184-2_32

[59] Pietri, A.: Organizing the graph of public software development for large-scale mining. (organisation du graphe de développement logiciel pour l'analyse à grande échelle). PhD thesis, University of Paris, France (2021). https://tel.archives-ouvertes.fr/tel-03515795

[60] Di Cosmo, R., Gruenpeter, M., Zacchiroli, S.: Identifiers for digital objects: the case of software source code preservation. In: Proceedings of the 15th International Conference on Digital Preservation, iPRES 2018, Boston, USA (2018). https://doi.org/10.17605/OSF.IO/KDE56. https://hal.archives-ouvertes.fr/hal-01865790

[61] Newman, M.E.: Power laws, pareto distributions and zipf's law. Contemporary physics **46**(5), 323–351 (2005)

[62] Clauset, A., Shalizi, C.R., Newman, M.E.: Power-law distributions in empirical data. SIAM review **51**(4), 661–703 (2009)

[63] Stumpf, M.P., Porter, M.A.: Critical truths about power laws. Science **335**(6069), 665–666 (2012)

[64] Watts, D.J., Strogatz, S.H.: Collective dynamics of small-worldnetworks. nature **393**(6684), 440–442 (1998)

[65] Schank, T., Wagner, D.: Approximating clustering coefficient and transitivity. Journal of Graph Algorithms and Applications **9**(2), 265–275 (2005)

[66] Maesa, D.D.F., Marino, A., Ricci, L.: Data-driven analysis of bitcoin properties: exploiting the users graph. International Journal of Data Science and Analytics **6**(1), 63–80 (2018)

[67] Kwak, H., Lee, C., Park, H., Moon, S.: What is twitter, a social network or a news media? In: Proceedings of the 19th International Conference on World Wide Web, pp. 591–600 (2010)

[68] Callan, J.: The lemur project and its clueweb12 dataset. In: SIGIR 2012 Workshop on Open-Source Information Retrieval (2012)

[69] Boldi, P., Marino, A., Santini, M., Vigna, S.: BUbiNG: Massive crawling for the masses. In: Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web, pp. 227–228 (2014). International World Wide Web Conferences Steering Committee

[70] Backstrom, L., Boldi, P., Rosa, M., Ugander, J., Vigna, S.: Four degrees of separation. In: Proceedings of the 4th Annual ACM Web Science Conference, pp. 33–42 (2012)

[71] Albert, R., Jeong, H., Barabási, A.-L.: Error and attack tolerance of complex networks. Nature **406**(6794), 378–382 (2000)

[72] Wang, X.F., Chen, G.: Complex networks: small-world, scale-free and beyond. IEEE Circuits and Systems Magazine **3**(1), 6–20 (2003)

[73] Rousseau, G., Di Cosmo, R., Zacchiroli, S.: Software provenance tracking at the scale of public source code. Empirical Software Engineering **25**(4), 2930–2959 (2020). https://doi.org/10.1007/s10664-020-09828-5