Ultra-Large-Scale Repository Analysis via Graph Compression

Paolo Boldi
Università degli Studi di Milano
Milan, Italy
paolo.boldi@unimi.it

Antoine Pietri
Inria
Paris, France
antoine.pietri@inria.fr

Sebastiano Vigna
Università degli Studi di Milano
Milan, Italy
sebastiano.vigna@unimi.it

Stefano Zacchiroli
University Paris Diderot and Inria
Paris, France
zack@irif.fr

***Abstract*—We consider the problem of mining the development history—as captured by modern version control systems—of ultra-large-scale software archives (e.g., tens of millions software repositories corresponding).**

We show that graph compression techniques can be applied to the problem, dramatically reducing the hardware resources needed to mine similarly-sized corpus. As a concrete use case we compress the full Software Heritage archive, consisting of 5 billion unique source code files and 1 billion unique commits, harvested from more than 80 million software projects—encompassing a full mirror of GitHub.

The resulting compressed graph fits in less than 100 GB of RAM, corresponding to a hardware cost of less than 300 U.S. dollars. We show that the compressed in-memory representation of the full corpus can be accessed with excellent performances, with edge lookup times close to memory random access. As a sample exploitation experiment we show that the compressed graph can be used to conduct clone detection at this scale, benefiting from main memory access speed.

***Index Terms*—mining software repositories, source code, version control systems, development history, software evolution, graph compression**

I. Introduction

Software evolution and clone detection have been very prolific research areas in software engineering over the past decades [23], [28]. Empirical methods, and software repository mining in particular, have been popular techniques used in those fields [20], [21]. An important factor behind these advances has been the increased availability of large collections of openly available software development artifacts such as source code and distributed version control system (DVCS) repositories. Such wealth of source code and development history data is the byproduct of, respectively, the popularization of free/open source software (FOSS) and the advent of social coding on collaborative development platforms [11], [17].

The *scale* at which it is nowadays possible to analyze *public* software development is exciting in terms of research outcome potential. GitHub alone hosts more than 100M repositories, GitLab.com several millions, plus a long tail of on-premise GitLab instances. Seminal work [30] on the evolution of the global corpus of public code shows that the amount of publicly available *original* code has been doubling every 24–30 months and is accelerating.

In addition to public software development, *private* large-scale collaborative software development is also on the rise in the corporate sector, due to the advent of inner source [9], [32]. Analyzing those private software collections—possibly *in combination* with the entire corpus of public code—has become nowadays a legitimate industrial need.

Most state-of-the-art approaches for analyzing this wealth of development artifacts mostly rely on classic "big data" approaches, partitioning the corpus over several machines and applying distributed algorithms. Alternatively, but less satisfactorily, sampling is used, incurring the risks of selection bias and over-generalized findings.

In this paper we evaluate the feasibility of a less resource-hungry approach to the analysis of the development history of very large software collections. Specifically, we will answer the following research question:

RQ: is it possible to efficiently perform software development history analyses at ultra large scale, on a single, relatively cheap machine?

As the question is partly quantitative and some terms in it are still vague, we further narrow it down as follows:

- with *development history* we mean the information usually captured by state-of-the-art DVCS [31], with commits as the finest available granularity;
- with *ultra large scale* we mean a scale similar to the known extent of all publicly developed software—to the best of our knowledge the best approximation of this is the Software Heritage archive [1], [14], whose dataset [26] will constitute our main benchmark;
- with *cheap machine* we mean commodity hardware, either desktop- or server-grade, that can be easily acquired by researchers with a moderate investment of a few thousand U.S. dollars.

In the following we will answer this research question *in the affirmative*, by applying lossless graph compression techniques to the underlying Merkle DAG [24] structure of modern DVCS. As a concrete use case we will compress the full development history of all publicly developed source code as captured by Software Heritage, consisting of 5 billion unique source code files and 1 billion unique commits, harvested from more than 80 million software projects.

As size benchmark, we show that the resulting compressed VCS graph, containing the development history of the entire corpus, can be loaded in ≈94 GiB of RAM, for an impressive

compression ratio of 4.9 bits/arc—as opposed to 8 bytes(!) per arc that a naive in-memory representation of the graph using adjacency lists would require. At current market rates the graph can hence be fit in RAM on commodity hardware with an investment of less than 300 U.S. dollar for main memory.

As a speed benchmark, we measure the time required to visit the entire graph, obtaining a visit time of less than 2 hours and a processing throughput of almost 2 million nodes per second using a single thread. We also measure the average time required to lookup successors of a given node, obtaining timings of 80 nanoseconds per arc, close to current DRAM random access times (50–60 ns).

To show the applicability of the proposed approach to repository analysis we use the compressed graph to conduct two classic experiments in software clone detection: we measure (1) how often identical file contents are found in different commits, and (2) how often identical commits are found in different repositories. Our experiences show the advantages of having the development history of the entire corpus in main memory as opposed to on secondary memory.

Replication package: A replication package for this paper is available from Zenodo [4].

II. Background

A. Graph compression

Many datasets are shaped into a graph structure that contains a wealth of information about the data itself, and many data mining tasks can be accomplished from this information alone (e.g., detecting outlier elements, identifying interest groups, estimating measures of importance and so on). Often, such tasks can be solved through suitable graph algorithms which typically assume that the graph is stored into main memory. However, this assumption is far from trivial to realize in many real-world cases, including the case of interest for the present paper, where many billions of nodes and arcs might exist. Finding effective techniques to store and access large graphs that can be applied fruitfully to these situations is one of the central algorithmic issues in the field of modern data mining.

A (lossless) *compressed data structure* for a graph must provide very fast access to the graph (let us say, slower but comparable to the access time required by its uncompressed representation in main memory) *without* decompressing it. While this definition is not formal, it excludes methods in which the successors of a node are not accessible unless, for instance, a large part of the graph is decompressed.

Different compressed data structures for graphs offer different trade-offs between *compression time* (the time required to produce a compressed representation from an uncompressed one) and *compression ratio* (the ratio between the size of the compressed data structure and its uncompressed counterpart, typically measured in bits per arc). One should also decide whether the data structure should be *static* or *dynamic* (whether it allows for changes), whether it is aimed at *directed* or *undirected* graphs (or both), and which *access primitives* the structure allows for. In most cases, these aspects can only be evaluated experimentally on a certain number of datasets, although in some rare circumstances it is possible to provide worst-case lower bounds under some assumption on the network structure (e.g., assuming some probabilistic or deterministic graph model, and evaluating the compression performances on that model).

TABLE I
NAIVE GRAPH REPRESENTATION USING ADJACENCY LISTS.

Node	Outdegree	Successors
...	...	...
15	11	13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	15, 16, 17, 22, 23, 24, 315, 316, 317, 3041
17	0	
18	5	13, 15, 16, 17, 50
...	...	...

TABLE II
COMPACT GRAPH REPRESENTATION USING GAPS AND COPY LISTS.

Node	Outd.	Ref.	Copy list	Extra nodes
...	...	...	...	...
15	11	0		3,1,0,0,0,0,3,0,178,111,718
16	10	-1	01110011010	6, 293, 0, 2723
17	0			
18	5	-3	11110000000	32
...	...	...	...	...

A common large, real-world graph that has been studied and compressed in the past is the graph of the Web (or *web graph*), consisting of one node per page and one arc per hyperlink between pages. A pioneering attempt at compressing the web graph is the LINK Database [27]. Suppose that nodes are ordered lexicographically by URL (i.e., node i is the node representing the i-th URL in lexicographic order); then the following two properties are true:

- *Locality*: Most arcs are between nodes that are close to each other in the order, because most links are intra-site, and URLs from the same site share a long prefix, which makes them close in lexicographic order. Locality can be exploited using *gap compression*: if node x has successors $y_1, y_2, \ldots, y_k$, gap compression stores for x the compressed successor list $y_1 - x, y_2 - y_1 - 1, \ldots, y_k - y_{k-1} - 1$; by locality, most values in this list will be small, and can be stored efficiently using variable-length encodings.
- *Similarity*: nodes that are close to each other in the order tend to have similar sets of neighbours. Similarity can be exploited by using *reference compression*: some successor lists are represented as a difference with respect to the successor list of a previous nearby node.

In Table I we show a sample of lists of successors of a graph, and in Table II we show the same lists in which reference compression (in the form of a bit mask) copies successors from a previous list (identified by the "Ref." column) following the information contained in a *copy list*, and then the remaining nodes (possibly all successors, if no reference compression is used) are gap-compressed.

B. The WebGraph framework

Some years later, building on the same approach, the WebGraph framework [6] attained using the *BV scheme* a web

graph compression of less than 3 bits/arc (and even less than 2 for the transposed graph) with a random-access successor enumeration time of a few hundreds of nanoseconds per arc (and much faster than that for sequential access).

The techniques described above are strongly sensitive to node order: this is not a big issue when applied to web graphs, because the lexicographic ordering of URLs is available, but makes it difficult to apply the same techniques to networks (e.g., social networks and, as we will see, version control system graphs) that do not have similarly meaningful canonical node identifiers.

A surprisingly effective ordering is simply that of enumerating nodes following a *visit*: in particular, a breadth-first visit [3] numbers nodes in such a way that might enable gap and reference compression to provide excellent results.

In this paper we will use the WebGraph open source implementation as technology to perform graph compression on a large corpus of VCS histories. We will also show that BFS visits are indeed an effective reordering strategy to achieve high compression on such corpus.

III. Data model

Analyzing software development histories requires dealing with the different data models that modern version control systems (VCS) implement [31]. While the differences among those data models are significant—and particularly so between "old school" VCSs like CVS and Subversion and modern distributed version control systems (DVCS) like Git and Mercurial—the latter are much more general and expressive and can faithfully represent development histories originally recorded in the former.

In this section we briefly present a *generic VCS data model*, with the same expressivity of modern DVCS data models, which we will use as input for the ultra-large-scale development history analyses that we aim to enable on limited resources using graph compression.

One peculiarity of modern collaborative software development is that source code artifacts are massively duplicated across repositories, commits and directories. That is the reason why most DVCS uses Merkle DAG [24] structures as the basis for their data models; we will do the same for our data model.

As a consequence of using Merkle structures, all represented software development artifacts are natively *de-duplicated* within and across projects—no source code file will be stored twice, no commit will be stored twice, etc. Also, every artifact gets attributed a persistent, cryptographically-strong hash as its intrinsic identifier, usually a SHA1. This approach has two main advantages:

- It considerably reduces the overall size of storing software development artifacts, thanks to deduplication.
- It enables tracking artifacts across the entire dataset, as identical artifacts found in multiple repositories will be represented as unique nodes in the graph, with ancestry relationships materializing their provenance.

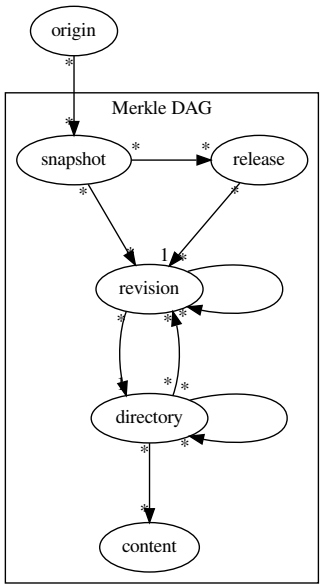

Fig. 1. Topology and cardinality diagram of a generic version control system graph, augmented with hosting URL nodes (origins).

The software artifacts that we support tracking in the proposed data model are those commonly found in state-of-the-art DVCS. The relationships among them and the topology of the induced graph with the respective cardinalities are shown in Fig. 1. A toy, yet fully detailed instantiation of the data model is shown in Fig. 2, in UML notation. These artifacts are organized in five different logical layers, discussed below.

Contents: are the leaves of the graph and represent source code files as byte sequences. File metadata (e.g., file names or access permissions) are not considered an intrinsic part of file contents and are only stored as part of directories.

Directories: materialize source code trees. Each directory is an associative list mapping local path names (e.g., `"foo.c"` or `"src/"`) to contents, directories (the recursive case of subdirectories), or in rare cases revisions (to represent, e.g., Git submodules or Subversion externals).

Revisions: (generally known as "commits") are time-indexed captures of the *root directory* of the source code of a given project. Each revision points to that root directory as well as an ordered list of parent revisions; a revision having more than one parent representing a *merge* commit.

Releases: (known as "tags" in some VCS) are revisions labelled with a specific, often human-meaningful name, to indicate their noteworthiness (e.g., `v3.11`).

VCS data models generally stop here. As we aim to analyze large collections composed of *many* repositories, we add an extra layer that allows to capture the *full state* of a given repository at a given point in time. This is useful to relate together different repositories used to collaborate on a single

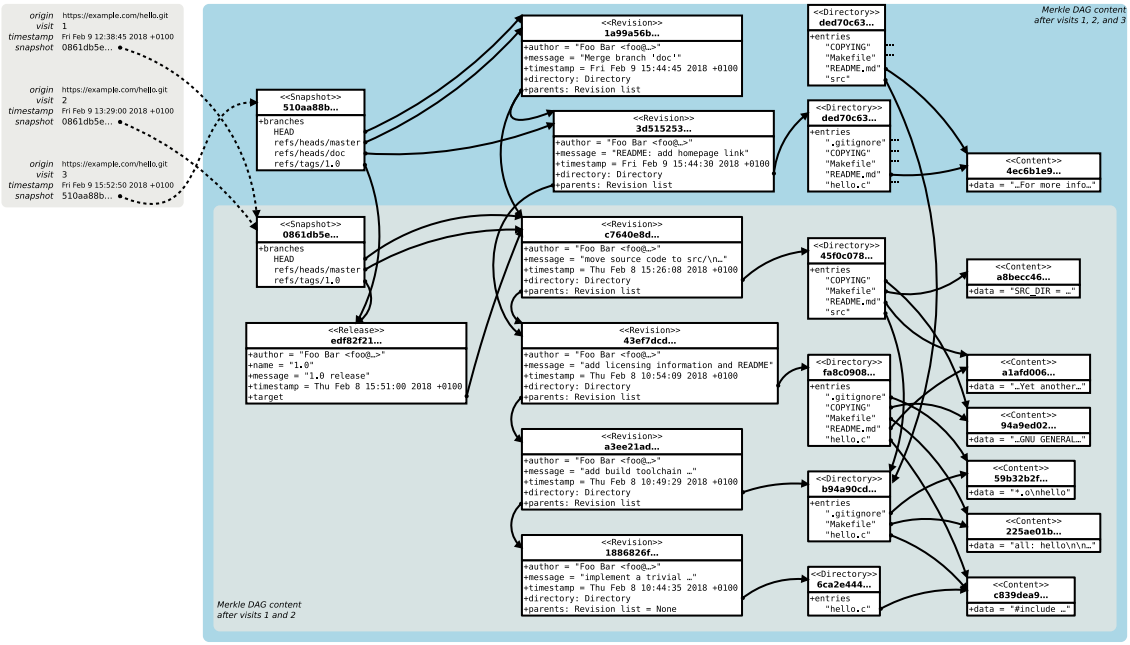

Fig. 2. Data model instantiation example, containing 7 file contents, 6 directories, 6 commits, and 3 snapshot artifacts, crawled during 3 visits of a single Git repository, 2 of which witnessed the same repository status.

project (as it is common in pull-request-based development models [17]). The additional artifacts we support are hence:

Snapshots: are point-in-time captures of the full state of a repository, as an associative list mapping branch names (e.g., `master` or `bug-42`) to releases or revisions objects. Snapshots are part of the Merkle DAG and allow to de-duplicate in it the full state of different repositories that happen to contain exactly the same development histories (e.g., unmodified forks of the same project created on GitHub).

Origins: are the URLs at which a given repository snapshot has been observed. They correspond to the hosting URLs of repositories (e.g., `https://github.com/user/repo`) but are more general as they point to all the repository snapshots that have been observed over time at a given repository hosting place.

IV. Version Control System Compression

We now set to establish whether graph compression is a suitable approach for enabling ultra-large-scale repository analysis on modest hardware resources. To that end we will conduct a case study by (1) obtain a suitably large dataset, (2) compress it using graph compression techniques, and (3) use the compressed result to conduct repository analysis. In this section we describe the dataset and compression results; in the next we will exploit the obtained compressed representation.

A. Case study: compressing the Software Heritage archive

As dataset we have chosen the largest publicly available archive of software source code and development history that we know of—the Software Heritage archive [1]. At the time of writing the project declares on its website[1] to have archived the development history of more than 90 million software projects, encompassing full mirrors of GitHub and GitLab.com, historical archives of Google Code and Gitorious, as well as repositories of popular package managers such as NPM, PyPI, and Debian.

TABLE III
Corpus statistics as a graph. For the sake of brevity the arcs table only reports about the most significant arc types; the total accounts for all arcs.

Nodes	
origins	88 M
snapshots	57 M
releases	9.9 M
revisions	1.1 B
directories	4.9 B
contents	5.5 B
Total nodes	12 B

Arcs	
origin → snapshot	195 M
snapshot → revision	616 M
snapshot → release	215 M
release → revision	9.9 M
revision → revision	1.2 B
revision → directory	1.1 B
directory → directory	48 B
directory → revisiony	482 M
directory → content	112 B
Total arcs	165 B

The project makes archive dumps available as research datasets [26]. We have retrieved the most recent dump (dated 2018-09-25) which is, to the best of our knowledge, the largest publicly accessible collection of version control system histories. The dataset is available for download[2] and can be used to independently reproduce the experiments described in

[1] https://www.softwareheritage.org
[2] https://docs.softwareheritage.org/devel/swh-dataset/

this paper. Table III shows statistics about the corpus size.

As Software Heritage also uses a common data model to archive different VCSs, the corpus data model matches the Merkle DAG structure described in Section III. The dataset is hence fully deduplicated by construction, so all node and arc counts in Table III are about *unique* entities—it hence contains 5B unique file contents, 49B *different* links from revisions to their parent revisions, etc.

B. Compression scope and metadata

Different analyses will need to access different information stored in VCS. A study of commit messages will not care about file contents, whereas one on code merges will need the revision graph. When you consider keeping the entire dataset in main memory for performance reasons, there is an inherent trade-off between access time and RAM requirements. A line has to be drawn to separate the data that benefits the most from fast RAM access, from the metadata that can be left on-disk without becoming a bottleneck.

The major bottleneck when performing development history analyses with on-disk data is generally the access to the neighbors of a given node during graph traversal. Since knowledge of node neighbors is necessary to advance in the iteration steps, these disk accesses cannot be deferred to a later processing stage. It is therefore very beneficial to keep as much *graph structure* and neighboring data as possible in memory to speed up the visits.

On the other hand, once a visit has been performed, the metadata of the visited nodes can be retrieved in a post-processing phase for subsequent analysis. As this metadata does not need to be sent back to the graph traversal routine for it to proceed, access latency of node metadata does not matter as much as it does for the graph topology. Node metadata can thus be retrieved in an asynchronous fashion without significantly impacting analysis time.

The scope of our compression experiment will hence primarily focus on compressing and storing in memory the *graph structure*, with the expectation that usage patterns will match the above scenarios—in-memory visits, then asynchronous retrieval and analysis of node metadata.

As sole exception we will also keep in memory *node types*, i.e., whether a node is a content, directory, revision, etc. That can be done very efficiently using a *type map* implemented as a bit array indexed by integer node identifiers and requiring only 3 bits per node (as there are 6 node types in total), or 4 GB of RAM for the entire graph. The reason to make this exception is that node type filtering is useful in many use cases to determine at runtime what kind of objects graph traversals should be looking at.

C. Compression pipeline

To compress the dataset we use the WebGraph framework to realize the compression pipeline shown in Fig. 3. The pipeline input is a simple graph representation (*Merkle DAG* in figure) as a pair of textual nodes and arcs file. The nodes file consists of one node label per line; the arcs file consists of one source/destination pair of node labels per line. At this stage node labels are textual and dataset-specific; in our case each label is a Software Heritage persistent identifier (PID) [13]. Then, the following steps are executed in order:[3]

a) MPH: Generate a *minimal perfect hash function* [16] that maps input node labels to the set $\{0, \ldots, N-1\}$ consecutive integers, where N is the number of input nodes. The resulting MPH function will be used in the following to quickly associate *an* integer to node labels without incurring the risk of collisions.

b) BV compress: Compress the adjacency matrix of the graph using gap compression and the other techniques described in Section II, but without relying on a sensible ordering of nodes yet. The output of this step is a *BV graph* [6].

c) BFS: As discussed in Section II, finding a node ordering that, by permuting rows, maximizes various locality properties on the graph adjacency matrix is key to achieve good compression. Differently from web graphs, version control system graphs do not sport ready-to-use ordering heuristics (such as the URL of each page) that are compression friendly. In fact, given native VCS node identifiers are generally based on cryptographic checksums (e.g., SHA1), the links from one node to the next will tend to jump *randomly* from one identifier to another in the space of all possible identifiers.

Experimentally we have verified that a breadth-first visit of the corpus graph, starting from graph roots (origin nodes) and traversing down toward leaves (file contents) achieve good compression results (compared to other orderings, e.g., LLP [5]). The *BFS* step of the compression pipeline hence performs such a visit on the entire BV graph.

d) Permute: Once the BFS ordering of nodes is known, this step will reorder nodes (as well as rows in the compressed adjacency matrix, performing all needed adaptations) according to BFS order. The result of this step is a compressed graph.

e) Transpose: Strictly speaking, the graph structure of the input dataset can be traversed only in one direction—from origins roots towards file content leaves. It is not uncommon for repository mining use cases to need to traverse the graph in the opposite direction though. For instance, looking up where a given file (or directory, or commit, etc.) has been found requires such *backward* visits.

As backward visits corresponds to forward visits on the *transposed* input graph, one can optionally generate a compressed representation of the transposed input graph and use it in addition (or alternatively) to the compressed graph obtained thus far. WebGraph supports generating the compressed representation of the transposed graph directly from the compressed (forward) graph. If desired, the *Transpose* step in the compression pipeline will take care of this.

D. Compression results

Compressing the full corpus is a resource-intensive endeavor. The wall time breakdown to perform the various steps

[3] we refer to the WebGraph documentation on how to practically run them: http://webgraph.di.unimi.it/

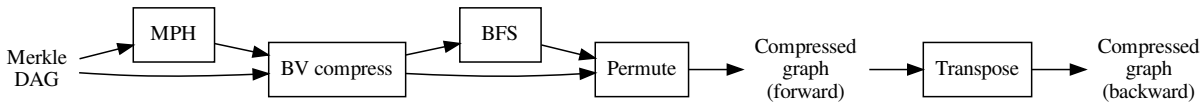

Fig. 3. Graph compression pipeline. Individual compression steps are denoted with boxes; notable compression input/output artifacts as free text. The last transposition step is optional and only needed to visit the graph backwards.

TABLE IV
COMPRESSION TIME BREAKDOWN

Step	**Wall time** (hours)
MPH	2
BV Compress	84
BFS	19
Permute	18
Transpose	15
Total	138 (≈6 days)

TABLE V
COMPRESSION RESULTS. COMPRESSION RATIOS ARE W.R.T. THE INFORMATION-THEORETICAL LOWER BOUND FOR GRAPHS WITH THE SAME DENSITY.

Forward (original) graph	
total size	91 GiB
bits per arc	4.91
compression ratio	15.8%

Backward (transposed) graph	
total size	83 GiB
bits per arc	4.49
compression ratio	14.4%

on the initial dataset is given in Table IV, totaling less than 6 days of compression time.

Timings have been taken on a server equipped with 24 CPUs and 750 GB of RAM. Note however, that such huge amount of RAM is not actually *needed* for compression; minimum RAM requirements correspond to the resources needed to load the final compressed graph in memory. The only step that used more than 100 GB of RAM was the BFS visit, which used a memory mapping to access the BV graph on disk using RAM as cache. Recent results on BFS memory efficiency [19] further confirm that the BFS step is not an impediment on the general applicability of the approach.

We also stress that even *if* significantly more resources were needed for compression than for exploitation of the compressed result, that would be an acceptable trade-off as: (1) compression can be done once and reused many times (possibly by other research groups), and (2) compression resources can be rented one-off, e.g., on public clouds.

Compression results are shown in Table V. Besides the raw datum of less than 5 bits per arc, which shows that the compression is very useful in practice, the compression ratio (≈15%) with respect to the information-theoretical lower bound of $\log \binom{n}{m}$ for a graph with n nodes and m arcs is about three times the one typical for web graphs, which are highly redundant, but significantly better than the typical values for social networks, which are above 50%. Moreover, as it often happens in networks generated by human activity, the transposed graph shows better compression performance because the indegree distribution has a fatter tail than the outdegree distribution, resulting in nodes with very dense predecessor lists.

TABLE VI
FULL GRAPH VISIT BENCHMARKS FOR A SINGLE-THREADED BFS VISIT

Forward (original) graph	
wall time	1h48m
throughput	1.81 M nodes/s (553 ns/node)

Backward (transposed) graph	
wall time	3h17m
throughput	988 M nodes/s (1.01 μs/node)

TABLE VII
ARC LOOKUP BENCHMARKS FOR 1 BILLION RANDOM NODES

Forward (original) graph	
visited arcs	13 644 656 586
throughput	12 018 223 arcs/s (83 ns/arc)

Backward (transposed) graph	
visited arcs	13 625 228 259
throughput	9 453 613 arcs/s (106 ns/arc)

Practically speaking, either direction of the input corpus can be fit in less than 100 GB of RAM, even including the 4 GB type map discussed above. Such an amount of memory can be easily installed on either powerful workstations or cheap server-grade hardware—at current market rates,[4] 100 GB of main memory costs less than 300 U.S. dollars. Fitting *both* graph directions on workstation deployments might be more challenging, but it is not always needed (e.g., one can load one graph direction depending on experimental needs) and still fit cheap server-grade deployments by today standards.

The main takeaway of this section is that, yes, from a size perspective, graph compression allows to fit the structure of enormous VCS datasets in memory on a single machine with limited hardware resources.

V. EXPLOITATION

We now move to the speed perspective to experimentally assess how effectively the obtained compressed representation of ultra-large-scale repository collections can be leveraged to perform repository mining experiments. We first perform a few domain-agnostic benchmarks (e.g., graph visits, arc traversal time) and then perform a domain-specific experiment.

A. Graph traversal

In the worst case of any given mining experiment, one will have to traverse the full corpus to obtain some insights. Hence it is important to know the baseline of how long it will take to do so. Table VI shows the results of benchmarking complete graph visits in breadth-first order (BFS), with no parallelism

[4] https://jcmit.net/memoryprice.htm, accessed 2019-10-18

(single thread) for both the original and transposed graphs. Timings have been measured on a server equipped with 3 GHz Intel Xeon Gold 6154 CPUs (only one of which has been used for the visits), with enough RAM to load either graph direction in memory without swapping to disk.

Results show that the in-memory approach delivers impressive performances. A full visit of the forward graph takes less than 2 hours with a throughput nearing 2 million nodes per second, or about 500 nanoseconds per node. Visiting the transposed graph is slower due to the already discussed differences in indegree v. outdegree distributions, but still very fast in absolute terms: the full transposed graph can be visited in little more than 3 hours with a throughput nearing 1 million nodes/second (1 μs/node).

Table VII shows the results of benchmarking random access to nodes and edges in the graph. A random sample of 1 billion nodes (8.3% of the entire graph) has been taken, enumerating for every node all of its successors. Results show that having the graph in memory pays back also in terms of random lookup time, with minimal overhead due to the compressed representation. On the original graph looking up the successor of a node takes 83 nanoseconds on average, close to the 50–60 ns estimates for current DRAM random access memory. Arc lookup on the transposed graph is slower, as already observed for full graph visits.

B. Source code artifact multiplication

As the goal is to benchmark the potential of the proposed approach for ultra-large-scale repository analysis we focus on a domain-specific experiment which needs to intensely crawl VCS histories in the studied corpus. Specifically, we will replicate the experiments of [30] to quantitatively assess the *multiplication factor* of source code artifacts in the corpus. We will measure:

1) The multiplication of source code file contents across commits (*content→revision multiplication* in the following), i.e., how much the same unmodified file content re-appears in different commits, no matter where (which origins) the commits come from. This measure correlates with and is a requirement for Type 1 (exact) clone detection, both within and across repositories.
2) The multiplication of commits across origins (*revision→origin multiplication*), i.e., how often the same commit re-appears in different repositories. The fact that the same commit re-appear at different origins is partly the result of collaborative social coding, but can also hint at the migration of development from one platform to another.

In the given data model, content→revision multiplication can be measured by iterating on all nodes of type content, then performing a visit on the transposed graph that only follows arcs leading to revision nodes—i.e., excluding (the transposed of) arcs snapshot→revision, release→revision, and origin→snapshot—and finally counting the number of revision leaves reached with the visit. Results can then be visualized as a distribution of the "popularity" of contents across commits.

Note that, even if the compressed graph representation does not natively store type information for arcs, we can use the type map discussed in Section IV to stop the visit when it is no longer possible to reach additional revision nodes.

The approach for determining revision→origin multiplication is similar. The only difference is that visits will start from commit nodes and stop at origin nodes (graph roots), which can then be counted and visualized as before.

The chosen approaches are naive from an algorithmic point of view—the same edges will be traversed over and over again for different input nodes, resulting in a time complexity of $O(V \cdot E)$, which is generally considered impractical on graphs of this scale. Having already established the overall efficiency of full graph visits, we chose these approaches for the sake of simplicity and explainability.

More time-efficient approaches are possible and would be equally well supported by the graph paradigm. For instance, one could propagate ancestry information during the visit, obtaining linear-time algorithms at the price of extra memory requirements (or extra memory mapping). Our main point is that the entire corpus *itself* can be fit in relatively little memory and accessed with excellent performances; other algorithmic considerations will vary according to the needs of the planned mining task, as they always do.

C. Results

Fig. 4 shows the multiplication factor distribution for content→revision, measured on a random sample of 953 M contents. Looking at the cumulative distribution (top line) the average multiplication factor appears to be very high. There are more than 100 million contents (≈20% of the sample) that are duplicated more than one thousand times in different commits; 10 million contents (1%) re-occur in more than ten thousand commits; and 1 million contents (0.1%) in more than a hundred thousand commits.

As we didn't partition by origin, these results do not say whether this multiplication is due to the long life of unmodified source code files in long-lived code bases, or instead due to reuse of the same unmodified files across different repositories. It would be easy to modify the experiment to follow edges up to origin nodes to determine that.

Fig. 5 shows analogous results for the revision→origin layer, on a random sample of 8.5 M revisions. To interpret them it is important to realize how it happens that the same commit (i.e., with an identical SHA1 identifier) is found in different repositories. The main reasons is the distributed nature of modern version control systems, whose repositories are massively represented in the corpus under analysis. Developers that work together using Git will have individual repositories that communicate by exchanging SHA1-identical commits. Furthermore the pull request development model [17] popularized by GitHub natively creates new repositories (hosted at different origin URLs) that initially contains all of the commits (unmodified) of the originally "forked" repository.

The cumulative distribution of Fig. 5 measure the amount of re-distribution of the same commits via different repositories

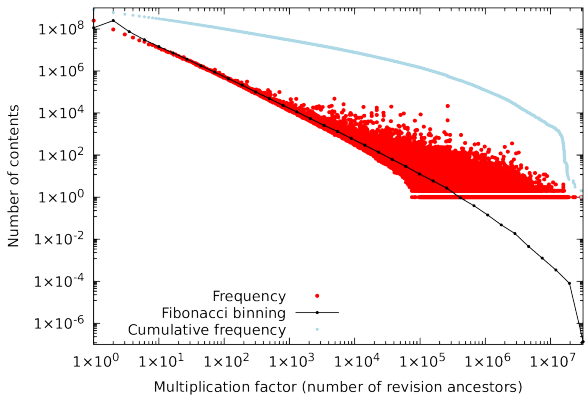

Fig. 4. Content→revision multiplication, i.e., how often file contents (Y axis) re-occur unmodified in different commits (X axis), based on a random sample of 953 M contents (17% of all contents).

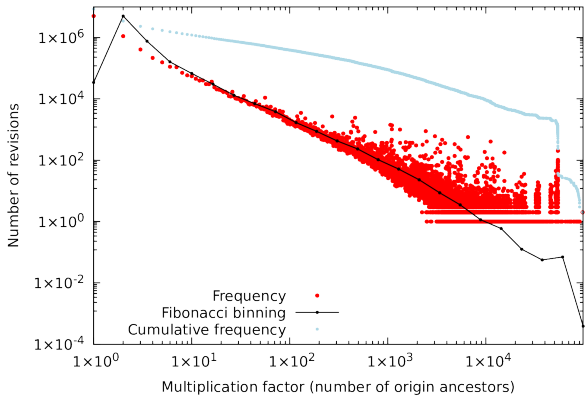

Fig. 5. Revision→origin multiplication, i.e., how often commits (Y axis) re-occur in different repositories (X axis), based on a random sample of 8.5 M revision (0.77% of all revisions).

in our corpus. 5 million commits (60% of the sample) can be found in a single repository only, but multiplication grows quickly from there: 100 thousand commits (1%) can be found in 1000 repositories or more and 10 thousand commits (0.01%) can be found in 10 thousand repositories or more. Development really is distributed these days as well as resilient to the disappearance of a single point of source code distribution.

To better appreciate the graph-intensive nature of realizing the above two experiments in practice, we have also measured visit sizes. Fig. 6 and 7 show the results in terms of visited nodes; results in terms of visited edges have been omitted for brevity, but they exhibit nearly identical patterns.

Content→revision visits traverse significantly more nodes (and edges) than revision→origin visits. This is expected due to the respective sizes of the traversed subgraphs (see Table III) and it is a dominant factor over the fact that, on average, the filesystem layer of the graph (content ∪ directory nodes) has shorter graph paths than the revision layer (VCS repositories

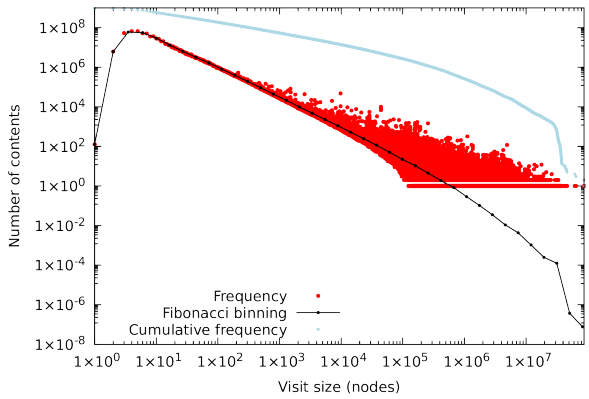

Fig. 6. Visit size, as the number of visited nodes, for measuring content→revision multiplication on the same sample of Fig. 4.

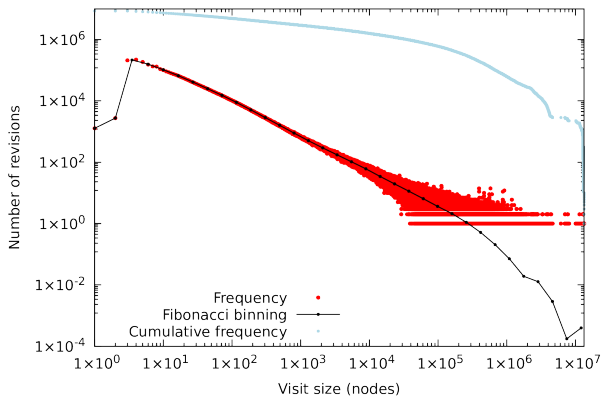

Fig. 7. Visit size, as the number of visited nodes, for measuring revision→origin multiplication on the same sample of Fig. 5.

of large software projects such as the Linux kernel can have commit chains nearing 1 M commits in length).

In terms of timings, the presented results have been obtained on multi-core machines with, respectively, 20 x 2.40 GHz CPUs (for content→revision) and 36 x 3.00 GHz CPUs (for revision→origin), letting the experiments run for about 2.5 days. In spite of the naive $O(V \cdot E)$ algorithmic approaches chosen, such short running times have allowed to process very large subgraphs.

The take-home messages for this section are that: (1) graphs are suitable data models for conducting version control system analyses, including code duplication experiments; and (2) *compressed* graphs allow to perform such analysis at ultra-large-scale with impressive graph traversal performances.

VI. Discussion

Both size-wise and performance-wise the results of applying graph compression to VCS graphs to support their ultra-large-scale analysis appear to be more than satisfactory. VCS graphs compress well both in absolute terms and in comparison with

other large graphs compressed in the past (e.g., web graphs). Graph visit performances are consistent with main memory access time and can support visit-intense VCS analysis needs well on limited resources. That notwithstanding we are not claiming that graph compression is a silver bullet for VCS analysis. We discuss in this section limitations and trade-offs that apply to the proposed approach.

A. Graph design

As mentioned in Section IV-B there exists a clear *space/time trade-off* between what fits in main memory and what should be left on secondary storage. Our choice—graph structure + node types in memory, everything else in secondary storage—will enable speeding up many use cases, under the assumption that VCS traversal is a common performance bottleneck of large-scale analyses.

Other choices are possible, valid, and can still be supported by graph compression, allowing to fit *more* information in RAM than what would be possible otherwise. We discuss a few possible scenarios below.

One might decorate graph nodes with *additional metadata* to be used during traversals and should hence be looked up efficiently. For instance, commit nodes might be equipped with in-RAM timestamps (either absolute, or relative, as in logical clocks) to direct visits to choose the earliest occurrence of what is being sought. Graph compression will be useful nonetheless, and the induced re-labeling of nodes to integers often enables storing additional metadata in memory in very compact ways, as we did for the type map. Attaching metadata to graph *arcs* is more tricky, but limited support for doing so is offered by WebGraph[5] and other state-of-the-art frameworks.

Graph semantics is another variable that can be played with. In the presented case study graph leaves are unmodified file contents, allowing to track bit-identical clones. One might use instead checksums of *normalized* source code files (e.g., removing spaces), obtaining the canonical definition of Type 1 clones, or parse source code files to ASTs and associate intrinsic identifiers to them (for Type 2 clones).

Graph granularity can also be increased, e.g., by adding nodes corresponding to individual lines of codes (normalized or otherwise). Doing so would enable tracking SLOC cloning and migrations at an unprecedented scale. It will also significantly increase the graph size, by a factor close to the average length of source code files (in SLOCs). The resulting graph will be huge, but as graph compression techniques are used in production on web graphs (with nodes in the trillions), we are confident they can scale up to SLOC analysis needs.

B. Limitations

A limitation of using static graph compression over classic information systems to store the data to be analyzed is that *compression is not incremental* w.r.t. the arrival of new artifacts (commits, files, etc.), due to the need of reordering nodes and updating the compressed representations of adjacency lists. Considering that popular VCS repositories receive hundreds of new commits a day, the already mentioned exponential growth of original publicly available code, and the observed multi-day compression times—this means that analyzed data will be chronically out-of-date w.r.t. reality.

This limitation is not problematic for research use cases, where datasets are generally frozen before conducting an experiment; but it might be problematic for other needs, like live-monitoring of interconnected private/public code bases. This is not a novel problem for other fields in which graph compression is applied, such as web search and social network analysis. Some compression techniques (e.g., k^2-trees [7]) lend themselves more or less naturally to be adapted to the dynamic case, but with a definite degradation in compression performances. A common mitigation technique that can be applied to all static compressors is to use dynamic, updatable *overlay graphs* on top of the compressed one. Overlays will be more memory-hungry, but they are ephemeral: periodically the underlying compressed graphs will be recompressed to regain storage-efficiency.

It is also theoretically possible to exploit knowledge of the graph topology to leave "gaps" in the node ordering that can be used as room for adding new nodes of a given type dynamically to the compressed graph representation. We intend to explore this possibility in future work.

VII. Related work

A. Large-scale mining approaches

Other large-scale repository mining approaches have been proposed in the past. Boa [15] has pioneered the idea of a mutualized infrastructure hosting both data and compute resources to perform large-scale analyses on source code artifacts such as those discussed in this paper. Our notion of "ultra-large-scale" is different, with a case study 100x larger by several metrics (projects, files, commits, etc.). The compute approach is also different, with Boa relying on distributed clusters (Hadoop) and our approach relying on a single machine. Direct performance comparisons are not possible as we have not replicated their experiments, but our results hint at a very significant speedup when the main bottleneck is history traversal. It would be interesting—and it seems entirely possible—to realize an infrastructure like Boa, based on a compressed VCS representation like the one proposed in this paper.

World of Code (WoC) [22] is a recent attempt at a mutualized infrastructure for large-scale VCS analyses. While limited to GitHub—contrary to our case study that also encompasses GitLab and major package repositories—the target scale of WoC is similar to ours. The compute approach is different, with WoC relying on distributed databases running and ours on a single machine. The advantage of WoC is that it maintains pre-computed mappings, e.g., from files and directories to the places they come from, choosing a different spot than ours in the classic space/time trade-off. The approach proposed here looks more appealing in terms of cost and ease of deployment. But the two approaches are complementary: WoC might

[5] via the `webgraph.labelling` package

benefit from an *additional* compressed graph representation that would shine when users need to explore on the fly (and as quickly as possible) artifact relationships that are not available as pre-computed mappings.

Large-scale VCS analyses have been conducted in the past, usually at much smaller scale than ours. An exception is [30], which developed a compressed representation for software provenance tracking and used it to conduct the multiplication experiments that we have replicated in Section V-B. Both approaches can be applied to conduct analyses on commodity hardware, but the compression trade-offs are different: specialized, lossy, and incremental in [30]; general purpose, lossless, but not incremental here.

LISA [2] is a framework for reducing artifact redundancy when analyzing VCS-stored source code. It is more fine-grained than our case study, reaching down to abstract syntax tree nodes. As such it could deduplicate more, but at the price of requiring a proper parser, which is not always available and might fail on syntactically incorrect files that one might still want to analyze. Also, we deal with all kinds of VCS artifacts while LISA is specific to source code files.

Large-scale experiments on development activities *not* captured by VCS histories (e.g., pull requests, code reviews, bug tracking, etc.) have also been conducted. They generally rely on dedicated activity databases, such as GHTorrent or GitHub Archive [18], [29]. Differences from the proposed approach are significant both in terms of scope (we focus on VCS histories, them on other activities) and needed resources.

B. Graph compression techniques

The problem of finding compression-friendly node orderings was studied from a theoretical viewpoint in [10], where the authors show that the problem of determining the optimal renumbering of nodes is NP-hard, but propose a heuristic (called "shingle ordering") for the problem, based on a fingerprint of the out-neighborhoods.

A set of different approaches is based on *clustering*: *layered labelled propagation* [5] is a reordering technique which combines the information from a number of clusterings to reorder the nodes. More recently, [12] extended the theoretical model of [10] and designed a novel ordering technique, called *recursive graph bisection*, that yields in several cases the most competitive compression ratios. The basic idea is to divide recursively the graph in two clusters of the same size so to minimize an objective function that estimates the compressibility of the overall graph when nodes in the first cluster have smaller identifiers than nodes in the second cluster.

To the best of our knowledge, the first two techniques would not scale to the case of interest for this paper. Recursive bisection might have some margin of benefit over a BFS, but at the price of a significantly slower computation. We plan to study this problem in the future.

A complementary approach to storing adjacency lists is that of k^2-trees [8]. In this case, one aims at representing compactly the adjacency *matrix* of the graph (as opposed to its adjacency *lists*), exploiting its sparseness. However, k^2-trees do not scale beyond a few dozen million vertices. They are hence inapplicable to the target scale of this paper.

Since successor lists are increasing integers, one can use *succinct data structures* [25] to store them. A succinct data structure does not *compress* data, it represents it using the minimum possible number of bits instead. Succinct approaches are particularly useful when the graph has no evident structure (as in that case reference compression does not really help) and when reordering the nodes is not possible (as the compression obtained is agnostic with respect to the distribution of gaps and to similarity). We hence did not consider these approached relevant to the case at hand.

VIII. Conclusion

The amount of freely available source code development artifacts has reached mind-blowing amounts, with GitHub alone surpassing 100 million repositories. That constitutes a treasure trove for empirical software engineering, but also a chore, due to the resources needed to efficiently mine the full extent of this global *software commons*. As an alternative to common "big data" approaches we have proposed in this paper to apply graph compression techniques to reduce the memory fingerprint of ultra-large-scale version control systems (VCS) repository analyses.

We have shown that the largest available collection of VCS repositories—namely Software Heritage, having archived 80 million repositories totaling 1 billion commits and 5 billion source code files—can be compressed down to and then loaded into less than 100 GB of RAM. At current market rates that corresponds to a RAM investments of just a few hundreds U.S. dollars.

In addition to being very compact, such a compressed in-memory representation is also very efficient: the entire corpus can be visited as a graph in less than 2 hours, with a arc lookup time of 80 nanoseconds. Those performances and the graph visit paradigm have enabled us to conduct classic code-clone detection experiments on significant subsets of the corpus in a couple of days of computation and in spite of having chosen very naive $O(V \cdot E)$ algorithmic approaches.

We conclude that graph compression has a huge potential in enabling unprecedented scale analyses of software repositories or, alternatively, in dramatically reducing the cost of analyses that are already possible today but can be performed with significantly fewer resources tomorrow.

As future work we plan, on the one hand, to experiment with variations in the graph nature, such as increasing leaf granularity to individual lines of code. On the other hand we intend to work on mitigating the main limitation of graph compression, i.e., its lack of incrementality, by developing compressed graph representations that leave room available for the addition of future nodes and edges, trading off some size efficiency for increased flexibility.

Acknowledgments

The authors thank Thibault Allançon for his internship work on `swh-graph` and Guillaume Rousseau for suggesting the multiplication experiments and comments on the paper.

References

[1] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. Building the universal archive of source code. *Commun. ACM*, 61(10):29–31, September 2018.

[2] Carol V. Alexandru, Sebastiano Panichella, Sebastian Proksch, and Harald C. Gall. Redundancy-free analysis of multi-revision software artifacts. *Empirical Software Engineering*, 24(1):332–380, 2019.

[3] Alberto Apostolico and Guido Drovandi. Graph compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.

[4] Paolo Boldi, Antoine Pietri, Sebastiano Vigna, and Stefano Zacchiroli. Replication package: Ultra-large-scale repository analysis via graph compression. Zenodo https://zenodo.org/record/3574459, doi:10.5281/zenodo.3574459, 2019. Persistent Software Heritage identifier [13] of the main software component used in the paper (swh.graph): swh:1:rel:132327bf1601eb890e28e96353bd61141923d9e1.

[5] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM, 2011.

[6] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[7] Nieves R. Brisaboa, Ana Cerdeira-Pena, Guillermo de Bernardo, and Gonzalo Navarro. Compressed representation of dynamic binary relations with applications. *Information Systems*, 69:106–123, 2017.

[8] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. Compact representation of Web graphs with extended functionality. *Information Systems*, 39(1):152–174, 2014.

[9] Maximilian Capraro and Dirk Riehle. Inner source definition, benefits, and challenges. *ACM Computing Surveys (CSUR)*, 49(4):67, 2017.

[10] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228, New York, NY, USA, 2009. ACM.

[11] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work*, pages 1277–1286. ACM, 2012.

[12] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1535–1544, New York, NY, USA, 2016. ACM.

[13] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. 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*, September 2018. Available from https://hal.archives-ouvertes.fr/hal-01865790.

[14] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017*, September 2017.

[15] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431. IEEE Press, 2013.

[16] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In Andrew V. Goldberg and Alexander S. Kulikov, editors, *Experimental Algorithms: 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, number 9685 in Lecture Notes in Computer Science, pages 339–352. Springer, 2016.

[17] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[18] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In Michele Lanza, Massimiliano Di Penta, and Tao Xie, editors, *9th IEEE Working Conference of Mining Software Repositories, MSR*, pages 12–21. IEEE Computer Society, 2012.

[19] Torben Hagerup. Fast breadth-first search in still less space. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 93–105. Springer, 2019.

[20] Ahmed E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008.

[21] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. 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.

[22] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. World of code: an infrastructure for mining the universe of open source vcs data. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 143–154. IEEE Press, 2019.

[23] Tom Mens. Introduction and roadmap: History and challenges of software evolution. In *Software evolution*, pages 1–11. Springer, 2008.

[24] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO'87, A Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.

[25] Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

[26] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. The Software Heritage graph dataset: public software development under one roof. In Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc, editors, *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada.*, pages 138–142. IEEE / ACM, 2019.

[27] Keith Randall, Raymie Stata, Rajiv Wickremesinghe, and Janet L. Wiener. The LINK database: Fast access to graphs of the Web. Research Report 175, Compaq Systems Research Center, Palo Alto, CA, 2001.

[28] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[29] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[30] Guillaume Rousseau, Roberto Di Cosmo, and Stefano Zacchiroli. Growth and duplication of public source code over time: Provenance tracking at scale. Technical report, Inria, 2019. https://hal.archives-ouvertes.fr/hal-02158292.

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

[32] Klaas-Jan Stol and Brian Fitzgerald. Inner source–adopting open source development practices in organizations: a tutorial. *IEEE Software*, 32(4):60–67, 2014.