Analyse du graphe de développement logiciel mondial

Antoine PIETRI

Inria, Paris

June 9th, 2022

Software Heritage
Software Mining

**Definition**

**Software mining**: studying existing software repositories to help improve software development processes and practices.

**Applications**

- Software health, software evolution
- Automated bug detection
- Automated vulnerability repair
- Code autocompletion
- Clone detection
- License compliance
- ...

Antoine Pietri

2 / 36

Analyse du graphe de développement logiciel mondial
Universal Software Mining

Current scale of software mining studies

- Individual projects
- Up to thousands of popular repositories (e.g., “top 1000 by stars”)
- Entire ecosystems (app stores, package managers, ...)

Universal software mining

Next step: a framework to run empirical studies on all the public software repositories?

- Less repetitive, no need to crawl the data for each study
- Easier to replicate studies
- High-level view of social processes in software development
In my thesis...

I study how to organize the graph of public software development, a comprehensive dataset of software development data, to make it accessible for software mining research.

Research direction: Working towards a research platform for Universal Software Analysis.

Antoine Pietri, Stefano Zacchirol
Towards Universal Software Evolution Analysis
BENEVOL 2018
We use the **Software Heritage archive** as our best approximation of the entire corpus of public software development.

Largest public source code archive in the world (more than 900 TB, growing daily).
A source code directory

```
/  
  |  
  | src
  |   
  |   evalexpr.c
  |   parser
  |   ast.c
  |   parser.c
  |   lexer.c
  |  
  | tests
  |   eval.c
  |   operands.c
```
Revisions (or “commits”) keep track of successive states of a source directory.
Developers can use “branches” to work on different features simultaneously.
Deduplication

- Instead of copying the nodes between each revision, we can identify & deduplicate them with **cryptographic hash functions** (e.g., SHA-1)
- Each object is identified by a unique identifier ("hash") computed from its entire subtree
In Software Heritage, *all* the repositories are consolidated into a single archive.

Software artifacts are deduplicated *across different repositories*.

The result is a single graph providing a *global, unified view on all the software development artifacts* from version control systems.

Helpful analogy: like a single Git repository but with all the public code in the world.
- Hash-based deduplication applied on every node in the graph ⇒ Merkle DAG
- Persistent structure: append only, great for archival

- 25B nodes
- 375B edges
Identifying researchers need

Literature review of 54 papers from the Mining Software Repositories conference (MSR 2019).

Categories of requested data

- Blobs
- Filesystem hierarchy (file names, directories)
- History graph (revisions)
- Content search (full-text search index)
- Provenance (backwards index)
- Commit diffs
- Community graph (revision authors)
- Dependency data
Data volume challenges

Local analysis

Handling data at that scale is a hard practical problem for researchers:
- Data does not fit on a single machine
- Downloading this volume of data can take months
- High deduplication: entangled structure, hard to parallelize

Approaches addressed in my thesis

- Sampling: access limited amounts of data
- Scale-out: platform for distributed computing
- Scale-up: compression
The **Software Heritage Graph Dataset**: a snapshot of the entire graph of software development (without the file contents).

Antoine Pietri, Diomidis Spinellis, Stefano Zacchirol

The Software Heritage graph dataset: public software development under one roof

Mining Software Repositories 2019

**Format**: A set of *relational tables* in columnar format (Apache ORC) for scale-out processing and graph analysis platforms

**Availability**

- Downloadable for local use
- Cloud processing platforms: Amazon Athena, Azure Databricks
**Example queries**

### Most frequent first commit words

```sql
SELECT COUNT(*) AS c, word FROM (
    SELECT LOWER(REGEXP_EXTRACT(FROM_UTF8(
        message), '^[^\w+]') AS word FROM revision)
WHERE word != ''
GROUP BY word ORDER BY COUNT(*) DESC LIMIT 5;
```

<table>
<thead>
<tr>
<th>Count</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>71,338,310</td>
<td>update</td>
</tr>
<tr>
<td>64,980,346</td>
<td>merge</td>
</tr>
<tr>
<td>56,854,372</td>
<td>add</td>
</tr>
<tr>
<td>44,971,954</td>
<td>added</td>
</tr>
<tr>
<td>33,222,056</td>
<td>fix</td>
</tr>
</tbody>
</table>

Analyzes 1.1 billion revision messages in 30 seconds.
Example queries

WITH revision_date AS
(SELECT FROM_UNIXTIME(date / 1000000) AS date
FROM revision)
SELECT yearly_rev.year AS year,
CAST(yearly_weekend_rev.number AS DOUBLE) / yearly_rev.number * 100.0 AS weekend_pc
FROM
(SELECT YEAR(date) AS year, COUNT(*) AS number
FROM revision_date
WHERE YEAR(date) BETWEEN 1971 AND 2020
GROUP BY YEAR(date) ) AS yearly_rev
JOIN
(SELECT YEAR(date) AS year, COUNT(*) AS number
FROM revision_date
WHERE DAY_OF_WEEK(date) >= 6
AND YEAR(date) BETWEEN 1971 AND 2020
GROUP BY YEAR(date) ) AS yearly_weekend_rev
ON yearly_rev.year = yearly_weekend_rev.year
ORDER BY year DESC;

Analyzes 1.1 billion revision timestamps in 7 seconds.
Recursive queries

This approach works really well for *embarrassingly parallel* queries.
Scale-out solutions are less efficient for *recursive queries* that exploit the hierarchical/structured nature of the graph.
BFS Traversal of the graph on Spark: 4 hours, 80 nodes (!), 5000 USD

Research question

Can recursive graph algorithms be performed in an accessible and cost-efficient way?
Compression approach

**Objective:** Analyzing the *entire graph of public software development* on a single machine.

Paolo Boldi, Antoine Pietri, Sebastiano Vigna, Stefano Zacchirol
Ultra-Large-Scale Repository Analysis via Graph Compression
SANER 2020, 27th Intl. Conf. on Software Analysis, Evolution and Reengineering. IEEE

**Advantages**

- Simpler for prototyping, no need to write distributed algorithms
- Cheaper than scale-out processing
- Allows us to run exhaustive analyses quickly

**Compression techniques**

Existing compression algorithms used with the *graph of the Web.*
Key for good compression of adjacency lists is a node ordering that ensures neighbor locality.

- Lexicographically-ordered URLs in the Graph of the Web have this property.
- It is *not* the case with cryptographic Merkle IDs...
- ...but is the case *again* after a breadth-first traversal

- **MPH**: minimal perfect hash, mapping Merkle IDs to 0..N-1 integers
- **BV compress**: Boldi-Vigna compression (based on MPH order)
- **BFS**: breadth-first visit to renumber
- **Permute**: update BV compression according to BFS order
We ran the compression pipeline on the input corpus using the **WebGraph** framework (Boldi, Vigna 2004).

- **Server equipped with 24 CPUs and 750 GB of RAM**
- **Compression time**: 138 hours (6 days)
- **Compression efficiency**: 6 TiB edge file → 91 GiB forward, 83 GiB transposed

**Benchmark**

Full traversal: 1h48min (1.81 M nodes/s) on a single thread

⇒ Huge improvement over Spark (4 h, 80 nodes, 5000 USD)
**Layered Label Propagation**: algorithm to uncover better locality-preserving node orderings (Boldi et al. 2010)

- Algorithm to uncover locality information
- Propagates labels on random nodes to discover neighborhoods
- Even more impressive compression ratio (91 GiB $\rightarrow$ 60 GiB, reduced by $\sim$35%)
- Compression requires more runtime memory
Graph Attributes

**Node attributes**
- The compressed in-memory graph structure has **no attributes**
- Usual data design is to exploit the 0..N-1 integer ranges to memory map **node attributes** from secondary storage (node ID → node attribute)
  - We do this for node types (mapping: 4 GiB), timestamps (mapping: 149 GiB), etc.
  - Data structures: integer/byte arrays, front-coded string lists, etc.

**Edge attributes**
- Built-in WebGraph support for attributes on the **edges** (generally integers)
- We convert *file names* to integers using a MPH
Option 1: Write a traversal algorithm using Java graph primitives

```java
HashSet<Long> visited = new HashSet<>();
Stack<Long> stack = new Stack<>();
stack.push(srcNodeId);
visited.add(srcNodeId);

while (!stack.isEmpty()) {
    long currentNodeId = stack.pop();
    LazyLongIterator it = graph.successors(currentNodeId);
    for (long neighborNodeId; (neighborNodeId = it.nextLong()) != -1; ) {
        if (!visited.contains(neighborNodeId)) {
            stack.push(neighborNodeId);
            visited.add(neighborNodeId);
        }
    }
}
```

- Efficient but low-level & requires local access to the graph server.
- Simpler/remote querying ⇒ need to build traversal query language
Option 2: HTTP API for simple graph traversals

- Generic remote API for graph traversals, Java/Python/aiohttp backend
- Limited to simple DFS from a single node (forward or backward graph)
- Traversal types: neighbors, leaves, all nodes, all edges
- Supports edge-type filtering

> GET /leaves/swh:1:rev:f39d[...]2a35?direction=backward
swh:1:ori:634a2b699d442aa9abd5008f379847816f54ab85
swh:1:ori:571a86b198c6c66ef33025249f7e455b529aae65
swh:1:ori:c15194d6cb59a6d32777ca3b287ea6664d540df3
...

swh:1:rel:c6df0a7ef73ca90825f1472b8a3c5f7a2ce3fc28
swh:1:rev:c8448ff2f9234332f0bc25dc3a13031f8ab3c73c
swh:1:rev:4b63dbd4e782e74bdc050c4579381d29b4bd41c0
...
The Software Heritage Graph Dataset materializes a *network of relationships between software artifacts* which has not yet been empirically studied as a whole.

**Research questions**

- What is the network topology of the graph of software development?  
  Network topology metrics: Degree distributions, connected components, distance between roots and leaves, clustering coefficient.
- What do these metrics tell us about this graph and its layers?
  - Best approaches for large scale analysis?
  - Methodological implications for software mining?

The **compressed graph framework** allows us to answer these questions experimentally.

Antoine Pietri, Guillaume Rousseau, Stefano Zacchioli

Determining the intrinsic structure of public software development history

*Mining Software Repositories 2020*
Average number of neighbors of all the nodes in the graph

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Average degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>swh-2020-commit</td>
<td>1.022</td>
</tr>
<tr>
<td>bitcoin-2013</td>
<td>6.4</td>
</tr>
<tr>
<td>dblp-2011 (Co-authorship)</td>
<td>6.8</td>
</tr>
<tr>
<td>swh-2020</td>
<td>11.0</td>
</tr>
<tr>
<td>swh-2020-filesystem</td>
<td>12.1</td>
</tr>
<tr>
<td>twitter-2010</td>
<td>35.2</td>
</tr>
<tr>
<td>clueweb12</td>
<td>43.1</td>
</tr>
<tr>
<td>uk-2014 (Web)</td>
<td>60.4</td>
</tr>
<tr>
<td>fb-2011 (Facebook)</td>
<td>169.0</td>
</tr>
</tbody>
</table>
Out-degree distributions: filesystem layer

Distribution of the number of entries of each directory in the graph

⇒ No characteristic number of entries in a directory.
Out-degree distributions: filesystem layer

Distribution of the number of entries of each directory in the graph

⇒ No characteristic number of entries in a directory.
Out-degree distributions: filesystem layer

Distribution of the number of entries of each directory in the graph

$\Rightarrow$ No characteristic number of entries in a directory.
Out-degree distributions: filesystem layer

Distribution of the number of entries of each directory in the graph

<table>
<thead>
<tr>
<th>Degree</th>
<th>Number of nodes</th>
<th>Frequency ( (= x) )</th>
<th>Cumulative freq. ( (\geq x) )</th>
<th>Power law fit ( (\alpha = 1.9) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10^0)</td>
<td>(10^1)</td>
<td>(10^2)</td>
<td>(10^3)</td>
<td>(10^4)</td>
</tr>
<tr>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>(3)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>(5)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>(7)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>(9)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
</tbody>
</table>

⇒ No characteristic number of entries in a directory.
Out-degree distributions: commit layer

Distribution of the number of parents of each commit in the graph

⇒ Characteristic number of parents due to development patterns.
### Distance between roots and leaves

#### (a) Depth of files in directory trees

<table>
<thead>
<tr>
<th>Size of each shortest path</th>
<th>Number of paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^0$</td>
<td>$10^1$</td>
</tr>
<tr>
<td>$10^1$</td>
<td>$10^2$</td>
</tr>
<tr>
<td>$10^2$</td>
<td>$10^3$</td>
</tr>
<tr>
<td>$10^3$</td>
<td>$10^4$</td>
</tr>
<tr>
<td>$10^4$</td>
<td>$10^5$</td>
</tr>
<tr>
<td>$10^5$</td>
<td>$10^6$</td>
</tr>
</tbody>
</table>

**Frequency ($= x$)**

**Cumulative freq. ($\geq x$)**

**Power law fit ($\alpha = 2.3$)**

#### (b) Length of commit chains

<table>
<thead>
<tr>
<th>Size of each shortest path</th>
<th>Number of paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^0$</td>
<td>$10^1$</td>
</tr>
<tr>
<td>$10^1$</td>
<td>$10^2$</td>
</tr>
<tr>
<td>$10^2$</td>
<td>$10^3$</td>
</tr>
<tr>
<td>$10^3$</td>
<td>$10^4$</td>
</tr>
<tr>
<td>$10^4$</td>
<td>$10^5$</td>
</tr>
<tr>
<td>$10^5$</td>
<td>$10^6$</td>
</tr>
</tbody>
</table>

**Frequency ($= x$)**

**Cumulative freq. ($\geq x$)**

**Power law fit ($\alpha = 1.5$)**

---

Antoine Pietri

Analyse du graphe de développement logiciel mondial
### Connected components

#### (a) Filesystem layer

<table>
<thead>
<tr>
<th>Size of each component</th>
<th>Frequency ($\geq x$)</th>
<th>Cumulative freq. ($\geq x$)</th>
<th>Power law fit ($\alpha = 2.3$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^3$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^5$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^7$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^9$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### (b) Commit layer

<table>
<thead>
<tr>
<th>Size of each component</th>
<th>Frequency ($\geq x$)</th>
<th>Cumulative freq. ($\geq x$)</th>
<th>Power law fit ($\alpha = 2.1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^3$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^5$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^7$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Statistics

<table>
<thead>
<tr>
<th>Layer</th>
<th># of WCC</th>
<th>Size of largest WCC</th>
<th>% in largest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full graph</td>
<td>33,104,255</td>
<td>18,902,683,142</td>
<td>97.79%</td>
</tr>
<tr>
<td>Filesystem layer</td>
<td>46,286,502</td>
<td>16,565,521,611</td>
<td>97.16%</td>
</tr>
<tr>
<td>Commit layer</td>
<td>88,031,649</td>
<td>51,543,944</td>
<td>2.61%</td>
</tr>
</tbody>
</table>
The filesystem and commit layers have almost opposite topological properties.

<table>
<thead>
<tr>
<th>Filesystem layer</th>
<th>Commit layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense, non-partitionable (giant WCC)</td>
<td>Sparse, partitionable (max WCC = 3%)</td>
</tr>
<tr>
<td>Characteristic depth</td>
<td>Arbitrary depth</td>
</tr>
<tr>
<td>Arbitrary out-degree</td>
<td>Characteristic out-degree (degenerate)</td>
</tr>
</tbody>
</table>
Implications for software mining research

**Layers**
- Large disparity in the topological structure of layers
- Important to study layers separately to understand the graph structure

**Methodology**
- No obvious threshold to “filter” outliers in many distributions
- Highlights the importance of exhaustive approaches

**Distributed analysis**
- No natural partitioning in small connected components
- Need for more subtle approaches?
Main contributions to universal software mining

- Making the graph available for research
  - Graph dataset (MSR 2019)
  - Graph compression (SANER 2020)
- Used for the **first exhaustive study on the graph structure of public software development**.

Future work

- Incremental graph compression
- Expressive query language for graph querying
- Derived graphs: commit diffs, co-authorship graph
Thanks!

All this work is open {source, data, access, ...}.


Why not...

**Comparison with other datasets**

- GHTorrent, Github on BigQuery: Github only
- source{d}: obsolete, top-bookmarked, Github only
- World of Code: Git only, mapping-oriented data model
- CodeDJ: GitHub only, model includes platform-specific metadata, interesting query system
- DejaVu: GitHub only, contains duplicate mappings but not the actual software objects

**Graph databases**

- Neo4J, Amazon Neptune, GraphFrames → Interesting to make them available, but costly. Scalability concerns?
Graph Subdatasets

Generating representative subgraphs

- Useful for smaller-scale experimentation, prototyping
- Focusing analysis on a relevant subset
- Representative samples → transitive closure of a subset of origins
- Use a fitted log model to estimate the size of the resulting subgraph