Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 223 (2023) 209–216

Procedia Computer Science

www.elsevier.com/locate/procedia

XII Latin-American Algorithms, Graphs and Optimization Symposium (LAGOS 2023)

The Problem of Discovery in Version Control Systems

Laurent Bulteau[a,b], Pierre-Yves David[c], Florian Horn[b,d]

[a]*Laboratoire d'Informatique Gaspard Monge, Université Gustave Eiffel*
[b]*Centre National de la Recherche Scientifique*
[c]*Octobus France*
[d]*Institut de Recherches en Informatique Fondamentale, Université Paris Cité*

Abstract

Version Control Systems, used by developers to keep track of the evolution of their code, model repositories as Merkle graphs of revisions. In order to synchronize efficiently between different instances of a repository, they need to determine the common knowledge that they share. This process is called *discovery*.

In this paper, we provide theoretical definitions for the problem of discovery, establish some universal upper and lower bounds on the amount of data that needs to be exchanged, as well as NP-hardness for a restricted variant (with only 2 round-trips). We also present and analyze some algorithms that are used in extant VCSs, such as Mercurial and Git, and propose an algorithm based on chain-decomposition.

© 2023 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0)
Peer-review under responsibility of the scientific committee of the XII Latin-American Algorithms, Graphs and Optimization Symposium

Keywords: Merkle Graphs; Version Control Systems

---

1. Introduction

Version Control Systems (VCSs) are tools that help developers follow the evolution of code over the lifetime of a project. They allow multiple developers to interact concurrently with the source code and keep track of every version of the project.

The first VCSs, *Source Code Control System* [1] and *Revision Control System* [2], tracked files separately and locally: they allowed collaboration, but only on one site, and only one programmer at a time. Both systems allowed users to check out earlier versions of the code (SCCS through regular deltas, RCS through reverse-deltas), although the process could be painstakingly slow.

A second generation of VCSs, spanning from *Concurrent Versioning System* [3] through *Subversion* [4] introduced a central repository containing all the versions of the files. This repository could be accessed remotely, allowing programmers in different locations to work on the same project. Locks were also removed and replaced by merging procedures, so multiple programmers could write code simultaneously.

1877-0509 © 2023 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0)
Peer-review under responsibility of the scientific committee of the XII Latin-American Algorithms, Graphs and Optimization Symposium
10.1016/j.procs.2023.08.231

The current generation of VCSs, represented by tools like *Git* [5] and *Mercurial* [6], does away with the notions of a central repository and current version. Instead, each developer has their own copy of a given repository, which is structured as a Merkle graph, *i.e.* an acyclic graph in which each node includes the hash(es) of its parent(s).

This allows anyone to add freely to their own copy, adding revisions, creating new branches or merging existing ones. They can also exchange nodes with another peer, pushing the revisions that they know and pulling those that they do not.

In larger projects such as Linux (Git) or Firefox (Mercurial), these operations can be very time-consuming: repositories can contain millions of revisions and grow daily by thousands of revisions, with peaks at several revisions per second. At this scale, it becomes necessary to optimize the exchange of information. This requires different agents to be able to determine efficiently which revisions they have in common. This is the problem of *discovery*, which we study in this paper from an abstract point of view. VCSs have been an object of academic interest in recent years [7, 8, 9, 10], although to the best of our knowledge, this article is the first to study this specific problem.

The remainder of this paper is organized as follows. In Section 2, we formally define our model for VCSs as well as the computation model that we use to analyse discovery algorithms. Then, in Section 3, we present several algorithms, including the ones used by Git and Mercurial. Section 4 studies the theoretical complexity of Graph Discovery, in terms of the number and volume of exchanges between the local and remote agents. In Section 5, we build test examples from large real-world repositories and analyze the performance of existing algorithms in terms of round-trips and total query size.

2. Definitions

2.1. Merkle graphs

A *directed graph* G is a pair (V, E) where V is a set of nodes and $E \subseteq V^2$ is a set of edges. It is a *directed acyclic graph* (DAG) if it does not contain any cycles. If $u \to v$ is an edge of a DAG, we call u a *parent* of v and v a *child* of u. The set of *ancestors* of u is the set recursively defined as the union of $\{u\}$ with the ancestors of the parents of u (if any). Similarly, the *descendants* of u is the union of $\{u\}$ with the descendants of its children, if any. We write $u \to^+ v$ if u is an ancestor of v with $u \neq v$.

A node with two or more parents is a *merge node* and a node with no parents is a *root*. A node with two or more children is a *branchpoint* and a node with no children is a *head*. A node with a single parent and a single child is a *linear node*. A *linear section* is a (possibly empty) sequence of linear nodes where each node is the parent of its successor, preceded by a *base* and followed by a *tip* which are non-linear nodes.

A *Merkle graph* is a DAG where nodes contain the hash of their parents (assuming hashes to be collision-free), which must belong to the graph. A Merkle graph is therefore entirely determined by the set of its heads. Furthermore, if two Merkle graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ have a common node, it has the same parents in both graphs. Formally, if $v \in V_1 \cap V_2$, then for any $u \to v \in E_1$, then $u \to v \in E_2$.

In the remainder of this paper, whenever we use the term *graph*, we mean a Merkle graph. When discussing existing algorithms, we sometimes use the terms *repository* and *revision* instead of graph and node.

2.2. Discovery Problem

We study the problem of Graph Discovery, in which a local agent knows a graph $G_\ell = (V_\ell, E_\ell)$ and has to determine its intersection with another graph known by a remote agent $G_r = (V_r, E_r)$. Formally, the common subgraph G_c has vertex set $V_c = V_\ell \cap V_r$. It is defined as $G_c = G_\ell[V_c] = G_r[V_c]$ by definition of Merkle graphs.

Graph Discovery
Input: local graph G_ℓ, remote graph G_r
Output: the common subgraph G_c of G_ℓ and G_r.

The local agent, who runs the algorithm, does not have direct access to information about the remote graph. Instead, it has to send network requests (prefixed by `remote` in our pseudo-code algorithms). Each algorithm may implement a different set of requests, possibly involving non-trivial computations on the remote side. A commonly used request, `remote.known(X)` with $X \subseteq V_\ell$, returns the subset of remotely known nodes among X (i.e., $X \cap V_r = X \cap V_c$).

In this article, we do not consider the classical complexity of Graph Discovery in terms of unit operations. We are only interested in network requests, specifically two quantities: the number of separate requests (*round-trips*) and the total size of the requests (*queries*), defined as the number of unique node identifiers sent in either direction.

Many algorithms start with a pre-processing step (*e.g.* an initial exchange of local and remote heads). We ignore this step in our pseudo-code algorithm, as they add a lot of complexity for little theoretical gain, but keep them for the experimental cases.

Although we focus on the theoretical study of the *exact* Discovery problem, an under-approximation, returning only a subset of the common vertices (or equivalently, an over-estimation return a super-set of the unknown vertices), is acceptable in practice. This case, which happens for example in Git, results in unnecessary node *data* being exchanged between peers, although the Discovery phase only exchanges node *identifiers*.

3. Discovery Algorithms in Git and Mercurial

3.1. Git: default discovery

In Git, discovery is done by default in breadth-first fashion, starting with the local heads (Algorithm 1). The local revisions are sorted in topological order[1] as `undecided`, with children preceding their parents. Then local asks remote which revisions they know over a channel with capacity 32, using a request `remote.known` returning the subset of known nodes. When an answer is positive, the revision and all its ancestors are added to `common` and removed from `undecided`. Furthermore, Git has a timeout threshold: if 256 successive revision are unknown to remote, it considers that no other revisions are known to remote and returns the current `common` set (resulting in an under-approximation).

3.2. Git: skipping discovery

As timeouts occurred too often in larger projects, Git introduced a skipping variant of its discovery algorithm. Instead of checking each revision in order to find the exact border of `common`, this version jumps over an ever growing number of generations in order to find a known ancestor (this corresponds to the two-parameter `ancestor` function line 22). This means that the skipping discovery may return an under-approximation of the set of `common` revisions even if the timeout is not reached. We transcribe this algorithm in pseudo-code as Algorithm 2. Note that the candidates are represented as a heap, rather than a queue, to ensure that candidates are always queried in topological order, even if they may have been added in a different order.

3.3. Mercurial: tree discovery

We include in our analysis the Mercurial *tree discovery* algorithm in use until 2010. Informally, the local agent asks for remote heads, and successively asks for linear sections (i.e. separated by merge nodes) in the remote graph as long as both ends of the sections are unknown. Finally, it uses a binary search over the linear sections starting with a known node to compute the precise border of the common subgraph.

In contrast with the gitaxian approach, tree discovery always answers the exact `common` set. It is also the only algorithm in this article that works from the structure of the remote graph, rather than the local one. The tree discovery algorithm performs well on cases with a small number of large linear section, but there were catastrophic cases where several thousands round-trips were necessary to complete discovery (see Section 5).

3.4. Mercurial: set discovery

In 2011, Mercurial replaced its tree discovery algorithm with a set-based discovery algorithm, which samples repeatedly the still undecided subset of the local repository (Algorithm 3).

It works by updating a partition of the local nodes in three sets: `common`, `missing` and `undecided`. In the beginning, all `local` nodes are `undecided`. In the end, they are all either `common` or `missing`.

[1] Git sorts by date, which may not be exactly topological, but their discovery algorithms ignore these cases and so shall we.

```
Data: local, remote
1  common ← ∅
2  undecided ← sort(local.nodes)
3  while undecided ≠ ∅ do
4  |  sample ← undecided[0:32]
5  |  present ← remote.known(sample)
6  |  for node ∈ sample do
7  |  |  if node ∈ present then
8  |  |  |  anc ← ancestors(node)
9  |  |  |  common.add(anc)
10 |  |  |  undecided.remove(anc)
11 |  |  |  timeout ← 0
12 |  |  else
13 |  |  |  undecided.remove(node)
14 |  |  |  timeout += 1
15 |  |  |  if timeout = 256 then
16 |  |  |  |  return common
17 return common
```

Algorithm 1: Git: default discovery

```
Data: local, remote
1  common ← ∅
2  undecided ← local.nodes
3  candidates ← ∅
4  for h in local.heads do
5  |  candidates.heappush((h,0))
6  while candidates ≠ ∅ do
7  |  sample ← ∅
8  |  for i in range(32) do
9  |  |  sample.add(candidates.heappop())
10 |  present ← remote.known(sample)
11 |  for (node,i) ∈ sample do
12 |  |  if node ∈ present then
13 |  |  |  anc ← ancestors(node)
14 |  |  |  common.add(anc)
15 |  |  |  undecided.remove(anc)
16 |  |  |  candidates.remove(anc)
17 |  |  |  timeout ← 0
18 |  |  else
19 |  |  |  timeout += 1
20 |  |  |  if timeout = 256 then
21 |  |  |  |  return common
22 |  |  |  for a ∈ ancestors(node,⌊i/2 + 1⌋) do
23 |  |  |  |  candidates.heappush(a,⌊3i/2 + 1⌋))
24 return common
```

Algorithm 2: Git: skipping discovery

```
Data: local, remote
1  common ← ∅
2  missing ← ∅
3  undecided ← local.nodes
4  while undecided is not empty do
5  |  sample ← sample_set(undecided)
6  |  present = remote.known(sample)
7  |  for node ∈ sample do
8  |  |  if node ∈ present then
9  |  |  |  common.add(ancestors(node))
10 |  |  |  undecided.remove(ancestors(node))
11 |  |  else
12 |  |  |  missing.add(descendants(node))
13 |  |  |  undecided.remove(descendants(node))
14 return common
```

Algorithm 3: Mercurial set discovery

In each iteration of the `while` loop, the algorithm samples a non-empty subset of the undecided nodes, using the `sample_set` method, and asks `remote` whether they know these nodes. All nodes in the sample are thus sorted in `common` or `missing`. Furthermore, thanks to the Merkle graph properties, the descendants of a missing node are also missing, and the ancestors of a common node are also common. The termination of the algorithm follows from the depletion of the `undecided` set. There are many possible variants of the Set-Discovery algorithm, depending on how one implements `sample_set`.

Mercurial defines *samplable nodes* according to the following rules: all heads and roots of `undecided` are samplable, and any other node is samplable if its height or depth (i.e., distance to some head or root) is a power of 2. In each round, the set of samplable nodes is computed, and a random subset of size 200 is selected from it. If there are less than 200 samplable nodes, additional random nodes from undecided are added.

4. Complexity bounds and chain-based algorithm

In this section we study the worst-case complexity of the discovery problem. Note that we assume that algorithms must be name-agnostic: it is impossible to derive any non-topological information from an identifier.

Remark 1. *In* Graph Discovery*, for any local node v, any exact algorithm needs to exchange at least one node u such that either u is an ancestor of v in* $V_\ell \setminus V_r$*, or u is a descendant of v in* $V_\ell \cap V_r$*.*

First, we give a "classical" complexity analysis of the discovery problem restricted to only 2 round trips. We show that finding a first query that would guarantee a limited number of overall node queries is NP-hard.

Theorem 1. *Given* G_ℓ *and an integer* $q > 1$*, it is NP-hard to decide whether the discovery can be completed in two rounds with at most q node queries in total.*

For worst-case pairs of local and remote graphs, the total query size in the problem of discovery is linear:

Lemma 2. *For any n, there exists a size-n local graph* $G_{\ell,n}$ *such that* Graph Discovery *needs a total query size of at least n nodes in the worst case over all graphs* G_r*.*

Proof. Let $G_{\ell,n}$ consist in n unrelated revisions. By Remark 1, since the nodes in V_ℓ do not have any ancestor or descendent except themselves, they must all be part of the exchanged set. □

We note that the above bound is tight, since it is always possible to solve the Discovery problem with the single size-n request `remote.known(local.nodes)`. However, the graphs involved in this lower bound are not very interesting from a practical point of view. There is also a universal logarithmic lower bound for any local graph of size n:

Lemma 3. *For any fixed local graph* G_ℓ *with n nodes,* Graph Discovery *needs a total query size of at least* $\lceil \log_2(n+1) \rceil$ *nodes in the worst case over all graphs* G_r*. This bound is tight.*

Our main theorem for this section deals with a wider range of possible graphs, giving a tight bound for the total number of revisions queried in terms of height and width of the graph.

A *chain* of a graph G is a sequence of nodes $(v_0, v_1, \ldots, v_m)$ where each v_i is an ancestor of v_{i+1}. An *antichain* is a set of nodes such that none of them is an ancestor of another. The maximal length of a chain in G is called the *height* of G. The maximal size of an antichain is its *width*. We recall Dilworth Theorem [11], stating that the size of the largest antichain in a DAG is equal to the minimal number of chains required to cover the graph.

Theorem 4. *There is a family of graphs of arbitrary width w and height h for which any algorithm needs a total query size of at least* $w\lceil \log_2(h+1) \rceil$*.*

There is a protocol that realizes discovery in $w\lceil \log_2(h+1) \rceil$ *total query size over* $\lceil \log_2(h+1) \rceil$ *round-trips for each graph of width w and height h.*

Proof. We first give a construction of the set of graphs yielding this worst-case lower bound. Given w and h, $G_\ell^{w,h}$ is a graph containing w disjoint paths of length h. As each path has size h, it follows from Lemma 3 that there needs to be $\lceil \log_2(h+1) \rceil$ queries related to that path. As the paths are disjoint, no information from one path can be useful to solve the other paths. It follows that any protocol solving discovery for $G_\ell^{w,h}$ requires at least $w\lceil \log_2(h+1) \rceil$ total queries in the worst case.

We describe an algorithm realizing discovery in $w\lceil \log_2(n+1) \rceil$ total query size over $\lceil \log_2(n+1) \rceil$ round-trips as Algorithm 4 (see Figure 1 for an example). It consists in finding a minimal chain cover of the local nodes and then searching for the border in each chain by dichotomy. Note that a chain may skip over some revisions that are covered by other chains. It may even skip over the border, in which case one half of the chain will eventually belong to `common` while the other will eventually belong to `missing`. □

```
    Data: local, remote
 1  common ← ∅
 2  chains ← chain_cover(local.nodes)
 3  while chains ≠ ∅ do
 4  |  request ← ∅
 5  |  for chain ∈ chains do
 6  |  |  request.add(middle(chain))
 7  |  present ← remote.known(request)
 8  |  for chain ∈ chains do
 9  |  |  if chain = ∅ then
10  |  |  |  chains.remove(chain)
11  |  |  else if middle(chain) ∈ present then
12  |  |  |  common.add(chain.split_to_base())
13  |  |  |  chain ← chain.split_to_tip()
14  |  |  else
15  |  |  |  chain ← chain.split_to_base()
16  return common
```

Algorithm 4: chain discovery

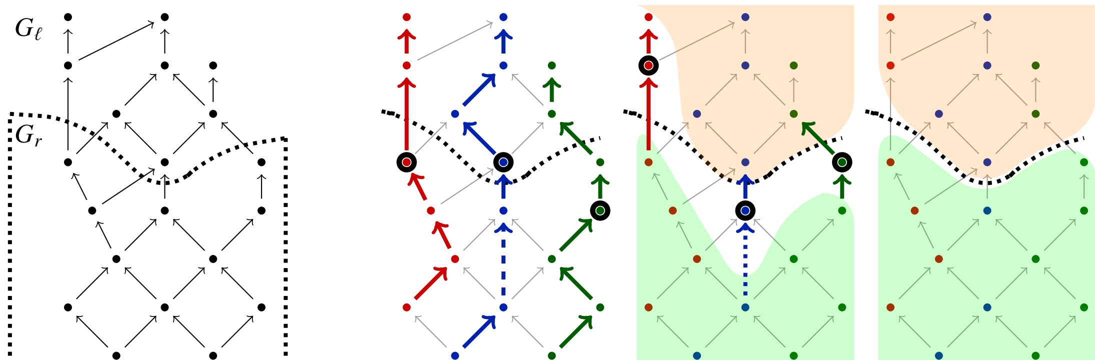

Fig. 1: The chain discovery algorithm. Left: a Merkle graph with an unknown (dotted) border. Right: a chain partition is computed (non-adjacent vertices consecutive in the blue chain are linked by a dotted arc), and middle vertices of each chain are probed successively until the known and unknown vertex sets (bottom and top shaded areas, respectively) cover the whole graph.

This chain discovery can be seen as an instance of set discovery, with a different way of sampling `undecided` nodes. In particular, there might be cross-chain inferences where a result on one chain gives information on another. However, the chain cover should *not* be computed anew in each loop, as the height of `undecided` may not be divided by two when we split the chains.

Computing a minimal chain covering is quadratic in time, which makes the Chain Discovery algorithm intractable in practice. However, Cáceres et al. [12] recently proposed a *parameterized* linear-time algorithm (i.e. linear for constant width) for the similar minimum path cover problem, which should be competitive for small width graphs. One may also maintain a chain cover problem through on-line insertions of the nodes. Since the nodes are inserted in topological order, each new node can be assigned a chain (without editing past nodes), using at most $O(w^2)$ chains for width-w graphs [13].

5. Experimental Results

We ran the Git, Mercurial and chain algorithms on large repositories to see how they behaved in practice. Our study cases were created from copies of pypy, netbeans, mozilla-unified, and mozilla-try repositories, to see account for different topologies. For each of them, we started from a specific repository state and then created 10 000 different

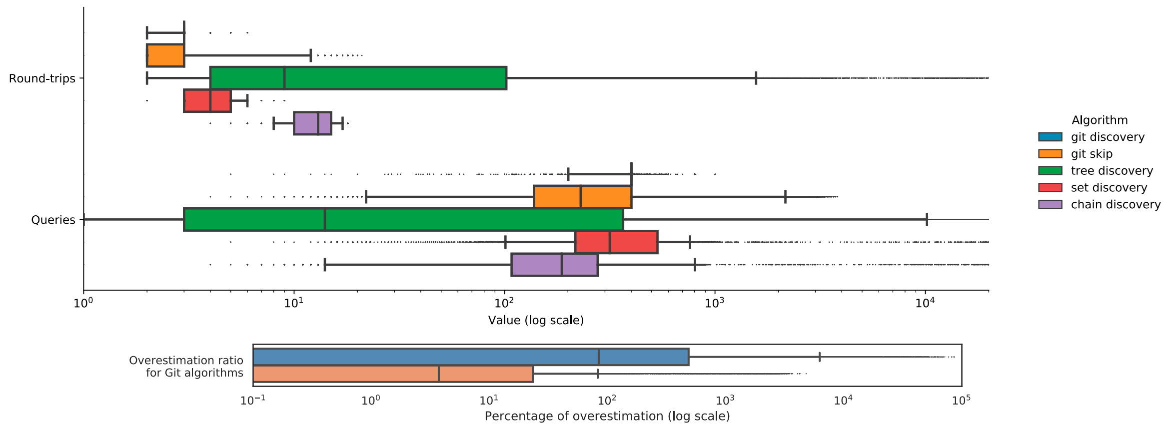

Fig. 2: Distributions of the number of round-trips, total query size, and over-estimation ratio (for Git algorithms) for each algorithm. Boxes delimit the second and third quartiles, and whiskers extend to the 5-95 percentiles. Overall, the current algorithms for Git and Mercurial (git skip and set discovery) have similar performances with typically 1–10 round trips and 1 000–10 000 queried nodes. Git yields an overestimation of typically 1–20%.

pairs of local subgraphs by choosing a revision at random and keeping only its ancestors. Our results are presented in Figure 2. We extended Git's sample size from 32 to 200 to get a fairer comparison with Mercurial's results.

Git's algorithms make a trade-off between speed and accuracy, giving up when cases become complicated. This helps keep the number of round-trips under control (visible in the top plot of Figure 2), but comes at the cost of a potentially large number of missed common revisions (visible in the bottom plot of Figure 2). This can represent a very large excess network use if already known revisions are later sent to the remote server. The skipping variant alleviates some of these problems, reducing the overestimation ratio to less than 50% in most of the cases, but the issue remains significant.

Mercurial's set discovery also has very low number of round-trips while tree discovery regularly used thousands (tens of thousands in the case of mozilla-try).

However, set discovery uses more total queries than tree discovery for pypy and netbeans. This is a direct consequence of set discovery maximising the information fetched in each round-trip: most of the apparent advantage of tree-discovery comes from "wasted" capacity, as overhead costs mean that a request for a few revisions is not much less costly than a request for 200 revisions.

Finally, we tested a naive implementation of the chain discovery algorithm, using a heuristic for chain decomposition. This algorithm obtains very competitive results in terms of number of queries, but at the cost of additional round-trips, as well as a time-consuming chain decomposition in the first place. We expect that larger queries (sampling more nodes per chain) can give a better trade-off between round-trips and total queries.

As a final statistic, we tested former versions of Mercurial on hand-picked difficult cases. The current algorithm adjusts dynamically the size of of the sample based on the number of heads and roots. Compared to fixed-size samples, this policy yields a 92% round-trip gain, while a simple growing policy of +5% per round already gives a 71% gain. On the other hand, larger samples lead to more queries per round trips, and overall there is an increase in total query size in both cases (+1.5% with the simple version, +13% in the dynamic version).

6. Conclusion

With the third generation of VCSs, the problem of discovery has emerged as a significant issue for efficient handling of larger repositories.

Mercurial's latest algorithm, set discovery, has very good performances, with a number of round-trips in the single digits. However, it should be noted that one reason for its performance is that requests can be arbitrarily large. In

pathological cases with many heads and a wide boundary, these requests can include several thousands revisions ids. This cannot always be done over http protocols, leading to a commensurate explosion on the number of round-trips.

It would be interesting to investigate new methods to reduce the cost of discovery for repositories with large width, especially when there are many heads. Towards this goal, the constraint that only nodes can be exchanged could be lifted: one can allow, for example, the agents to share hashes of larger sets of nodes, in order to check whether all nodes in some set are known without sending each node one by one. Such an approach would be particularly useful in cases with a large common subgraph between the local and remote agents.

References

[1] M. J. Rochkind, The source code control system, IEEE Trans. Software Eng. 1 (4) (1975) 364–370. doi:10.1109/TSE.1975.6312866.
[2] W. F. Tichy, Design, implementation, and evaluation of a revision control system, in: Y. Ohno, V. R. Basili, H. Enomoto, K. Kobayashi, R. T. Yeh (Eds.), Proceedings of the 6th International Conference on Software Engineering, IEEE Computer Society, 1982, pp. 58–67. URL http://dl.acm.org/citation.cfm?id=807748
[3] D. Grune, Concurrent versions system, a method for independent cooperation, Tech. Rep. 113, Vrije Universiteit Amsterdam (1986).
[4] B. Collins-Sussman, B. W. Fitzpatrick, C. M. Pilato, Version control with Subversion - next generation open source version control, O'Reilly, 2004. URL http://www.oreilly.de/catalog/0596004486/index.html
[5] L. Torvalds, Initial revision of "GIT", the information manager from hell, Tech. rep. (2005). URL https://github.com/git/git/commit/e83c5163316f89bfbde7d9ab23ca2e25604af290
[6] O. Mackall, Towards a better SCM: Revlog and Mercurial, Tech. rep. (2005). URL http://selenic.com/mercurial
[7] S. P. De Rosso, D. Jackson, What's wrong with GIT? A conceptual design analysis, in: A. L. Hosking, P. T. Eugster, R. Hirschfeld (Eds.), ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013, ACM, 2013, pp. 37–52. doi:10.1145/2509578.2509584.
[8] J. Levenberg, Why Google stores billions of lines of code in a single repository, Commun. ACM 59 (7) (2016) 78–87. doi:10.1145/2854146.
[9] A. Babenhauserheide, Automatic coordinated rebase with changeset evolution and Mercurial (2020). URL https://blog.disy.net/hg-evolution/
[10] J. Courtiel, P. Dorbec, R. Lecoq, Theoretical analysis of GIT bisect, in: A. Castañeda, F. Rodríguez-Henríquez (Eds.), LATIN 2022: Theoretical Informatics, Springer International Publishing, Cham, 2022, pp. 157–171.
[11] R. P. Dillworth, A decomposition theorem for partially ordered sets, Annals of Mathematics 51 (1) (1950) 161–166. doi:10.2307/1969503.
[12] M. Cáceres, M. Cairo, B. Mumey, R. Rizzi, A. I. Tomescu, Sparsifying, Shrinking and Splicing for Minimum Path Cover in Parameterized Linear Time, pp. 359–376. doi:10.1137/1.9781611977073.18.
[13] S. Felsner, On-line chain partitions of orders, Theoretical Computer Science 175 (2) (1997) 283–292. doi:https://doi.org/10.1016/S0304-3975(96)00204-6.