The Problem of Discovery in Version Control Systems

Laurent Bulteau, Pierre-Yves David, Florian Horn

▶ **To cite this version:**

Laurent Bulteau, Pierre-Yves David, Florian Horn. The Problem of Discovery in Version Control Systems. 2022. hal-03830513v1

HAL Id: hal-03830513

https://hal.science/hal-03830513v1

Preprint submitted on 2 Nov 2022 (v1), last revised 28 Nov 2023 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

The Problem of Discovery in Version Control Systems

Laurent Bulteau[1,2], Pierre-Yves David[3], and Florian Horn[2,4]

[1] Laboratoire d'Informatique Gaspard Monge, Université Gustave Eiffel
[2] Centre National de la Recherche Scientifique
[3] Octobus France
[4] 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 and establish some universal upper and lower bounds on its communication complexity. We also present and analyze some algorithms that are used in extant VCSs, such as Mercurial and Git.

1 Introduction

Version Control Systems (VCSs) are tools that help developers keep track of 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* [11] and *Revision Control System* [12], 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* [8] through *Subversion* [2] 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 programmer could write code simultaneously.

The current generation of VCSs, represented by tools like *Git* [13] and *Mercurial* [10], 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, *ie* an acyclic graph in which each node includes the hash(es) of its parent(s).

This allow 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 require 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 [5, 9, 1, 3], although to the best of our knowledge, this article is the first to study this 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 Sections 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 a node is the smallest set that contains it and the ancestors of its parents. The set of its *descendants* is the smallest set that contains the node itself and the descendants of its children. 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 algorithm, we sometimes use the terms *repository* and *revision* instead of graph and node.

A *chain* of a graph G is a sequence of nodes $v_0 \to^+ v_1 \to^+ \ldots \to^+ v_n$. An *antichain* is a set of vertices 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*.

Theorem 1. *[6] The size of the largest antichain in a DAG is equal to the minimal number of chains required to cover the graph.*

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 $G_c = G_\ell[V_\ell] = G_r[V_r]$ 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).

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, 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 and our theoretical analysis, as they add a lot of complexity for little theoretical gain. Note however that this step is important in practice, as we show in an analysis of variants in Table 3.

3 Discovery Algorithms

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[5] as `undecided`, with children preceding their parents. Then local asks remote which revisions they know over a channel with capacity 32. 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.

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

```
Data: local, remote
common ← ∅
undecided ← sort(local.nodes)
while undecided ≠ ∅ do
    sample ← undecided[0:32]
    present = remote.known(sample)
    for node ∈ sample do
        if node ∈ present then
            common.add(ancestors(node))
            undecided.remove(ancestors(node))
            timeout = 0
        else
            undecided.remove(node)
            timeout += 1
            if timeout = 256 then return common
return common
```

Algorithm 1: Git: default discovery

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 means that the skipping discovery returns an under-approximation of the set of `common` revisions. 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

Until 2010, Mercurial used a tree discovery algorithm (Algorithm 3). In contrast with the gitaxian approach, tree discovery always answer 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. First, local asks remote from its heads (with `remote.get_heads`). Then, it looks for the linear sections that contain a border node. Finally, it uses binary search inside each border section to look for the exact border.

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 4)

```
Data: local, remote
common ← ∅
undecided ← local.revs
candidates ← ∅
for h in local.heads do
  candidates.heappush((h,0))
while candidates is not empty do
  sample ← ∅
  for i in range(32) do
    sample.add(candidates.heappop())
  present = remote.known(sample)
  for (rev,i) ∈ sample do
    if rev ∈ present then
      common.add(ancestors(node))
      undecided.remove(ancestors(node))
      candidates.remove(ancestors(node))
      timeout = 0
    else
      timeout += 1
      if timeout = 256 then return common
      for a in ancestors(rev,⌊i/2 + 1⌋) do
        candidates.heappush(a,⌊3i/2 + 1⌋))
return common
```

Algorithm 2: Git: skipping discovery

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`.

In each iteration of the `while` loop, the algorithm samples a non-empty subset of the undecided nodes 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 an 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;
- a node is samplable if its height or depth 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.

Mercurial discovery went through some policy changes over the years. We summarize the most significant here:

14164:cb98fed52495 Introduces Set Discovery; the size of the sample is fixed at 200;

```
Data: local, remote
sections, seen, border_sections, split_sections ← ∅, ∅, ∅, ∅
for node in remote.get_heads() do
    request.add(node)
while request ≠ ∅ do
    request ← ∅
    for section in remote.get_sections(request) do
        if section.base ∈ local then
            border_sections.add(section)
        else if section.base ∉ seen then
            seen.add(section.base)
            request.add(section.base)
while border_sections ≠ ∅ do
    split_sections = remote.split(border_sections)
    border_sections ← ∅
    for (mid,lower,upper) ∈ split_sections do
        if lower == ∅ then
            border.add(mid)
        else if mid ∈ local then
            border_sections.add(lower)
        else
            border_sections.add(upper)
return border
```

Algorithm 3: Mercurial: tree discovery

67554:dbd0fcca6dfc The size of the sample grows over time (+5% at each round)

91746:2b1b8f3e6510 The actual size of the sample for each round is dynamically adjusted with the minimum of number of roots and number of heads, if it is larger than the current target sample size.

4 Complexity bounds for the discovery problem

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 $V_\ell \cap V_r$.

In all generality, the total query size in the problem of discovery is linear:

Lemma 1. *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.*

```
Data: local, remote
common ← ∅
missing ← ∅
undecided ← local.nodes
while undecided is not empty do
    sample ← sample_set(undecided)
    present = remote.known(sample) for node ∈ sample do
        if node ∈ present then
            common.add(ancestors(node))
            undecided.remove(ancestors(node))
        else
            missing.add(descendants(node))
            undecided.remove(descendants(node))
return common
```

Algorithm 4: Mercurial set discovery

Proof. Let $G_{\ell,n}$ consists 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.

However, the graphs involved in this lower bound are not very interesting from a practical point of view. There is also a universal lower bound for any graph of size n:

Lemma 2. *For any fixed local graph G_ℓ with n nodes,* Graph Discovery *needs a total query size of at least $\log_2(n)$ nodes in the worst case over all graphs G_r.*

Proof. Let G_ℓ be a graph of size n. We consider the cases in which the graph G_r has the same topology as G_ℓ, and each individual node can only be equal to the remote node in the same position. As our algorithms are name-agnostic, `local` and `remote` have exactly the same information, so we can assume that all computation is done by `local`. Furthermore, there is no difference between `local` requesting the id of remote's node at position p and `local` asking `remote` whether they know the id of its own node at position p. It follows that `remote`'s contribution to the communication is a string of binary answers to queries by local. As there are at lease n possible outcomes for the discovery problem of G, there must be at least $\log_2(n)$ different queries in the worst case.

This lower bound can be realized in the case of a graph consisting in a single path of length n, by conducting a dichotomy search over the path (see Figure 1).

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.

Theorem 2. *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 $w \log_2(h)$.*

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

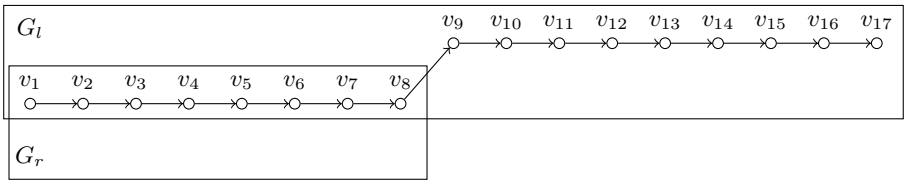

Fig. 1: Single-branch case

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 2 that there needs to be $\log_2(h)$ 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 $w \log_2(h)$ total queries in the worst case.

We describe an algorithm realizing discovery in $w \log_2(h)$ total query size over $\log_2(h)$ round-trips as Algorithm 5. 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`.

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.

Remark 2. Computing a minimal chain covering is quadratic in time, which makes the Chain Discovery algorithm intractable in practice. However, Cáceres et al. [4] 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 [7].

5 Experimental Results

We ran the Git and Mercurial 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 pairs of local subgraphs by choosing a revision at

```
Data: local, remote
common ← ∅
chains ← chain_cover(local.nodes)
while chains ≠ ∅ do
    request ← ∅
    for chain ∈ chains do
        request.add(middle(chain))
    present ← remote.known(request)
    for chain ∈ chains do
        if chain = ∅ then
            chains.remove(chain)
        else if middle(chain) ∈ present then
            common.add(chain.split_to_base())
            chain ← chain.split_to_tip()
        else
            chain ← chain.split_to_base()
return common
```

Algorithm 5: chain discovery

random and keeping only its ancestors. The results that we obtained are presented in Table 1. 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 Table 1), but comes at the cost of a potentially large number of missed common revisions (visible in Table 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, 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).

A contrario, 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.

Even so, the worst cases for the total queries are significantly worse for tree discovery than for set discovery. In mozilla-discovery and mozilla-try, these difference is significant enough that set discovery has a lower average number of queries, even though the median is much smaller for tree-discovery.

Note that this progress did not occur all at once. In Table 3, we show a series of hand-picked situation that illustrate how the worst cases improved on the successive versions of set discovery. The first column does not correspond to an actual implementation of set discovery in Mercurial: it is an illustration of what happens if heads are not exchanged at the beginning of the algorithm.

	avg	med	<10	10-10^2	10^2-10^3	10^3-10^4	10^4-10^5	> 10^5
pypy: total queries								
tree discovery	823	6	6840	575	804	1731	50	
set discovery	327	294	1	1039	8924	87		
default discovery	381	401	0	6	9987	7		
skipping discovery	163	155	4	4441	5533	22		
pypy: round-trips								
tree discovery	107	7	6823	980	2118	79		
set discovery	4.0	4	10000					
default discovery	3	3	10000					
skipping discovery	2	2	10000					
netbeans: total queries								
tree discovery	248	15	4577	3299	1570	513	41	
set discovery	318	281	4	66	9647			
default discovery	359	401	4	19	9977			
skipping discovery	1134	832.5	19	2136	2983	4862		
netbeans: round-trips								
tree discovery	94	9	5425	3334	955	286		
set discovery	3.6	3	10000					
default discovery	3	3	10000					
skipping discovery	7	6	6046	3954				
mozilla-unified: total queries								
tree discovery	1183	147	3191	988	4826	572	423	
set discovery	488	496	2	77	9340	11		
default discovery	403	401	1	8	9991			
skipping discovery	223	215	5	1473	8520	2		
mozilla-unified: round-trips								
tree discovery	211	78	3491	4045	1875	589		
set discovery	4.5	4	10000					
default discovery	3	3	10000					
skipping discovery	3	3	10000					
mozilla-try: total queries								
tree discovery	7823	93	4713	308	1018	2435	1350	176
set discovery	1675	413	0	238	7605	101	334	8
default discovery	340	401	0	7	9993			
skipping discovery	275	226	5	278	9646	71		
mozilla-try: round-trips								
tree discovery	2107	25	4797	912	2062	1661	568	
set discovery	4.3	4	10000					
default discovery	3	3	10000					
skipping discovery	3	3	9989	11				

Table 1: Queries and round-trips for Mercurial and Git algorithms.

	avg	p50	p90	p99	pmax
pypy (total 105948 revisions)					
default discovery	16010	18444	36991	60049	73679
skipping discovery	1218	223	2440	22995	57398
netbeans (total 306930 revisions)					
default discovery	75276	0	264248	292058	302701
skipping discovery	17180	1	79325	263840	281679
mozilla-unified (total 624961 revisions)					
default discovery	156234	159211	330634	481583	583708
skipping discovery	29321	1797	102811	270164	407565
mozilla-try (total 1671467 revisions)					
default discovery	156065	200661	350447	426468	467416
skipping discovery	18528	424	44446	300333	373813

Table 2: Number of errors (missed common nodes) for Git algorithms.

The situations were generated using a variety of methods, aiming at finding cases as hard as possible for the current version of the set discovery algorithm. The local and remote graphs where selected using one of the following methods:

- Choose a revision at random, and select all of its ancestors;
- Choose an antichain at random, and select all of its ancestors;
- Choose a head, remove it, and repeat that process a large number of times.

The result show a significant improvement in the number of round-trips needed to solve these complex cases. However the number of total queries can remains quite high.

No exchange		**Initial**		**Growing**		**Dynamic**	
total queries	round trips	total queries	round trips	total queries	round trips	total queries	round trips
228374	1142	295846	676	297546	75	324614	7
14710	74	12974	43	13271	25	14722	7
9251	47	7696	27	7933	19	8688	7
352650	1764	448757	1441	453629	92	514506	9
75871	380	235492	374	236566	64	253305	9
177959	890	179276	889	182575	82	211372	10

Table 3: Set discovery improvements over different variants, for six hand-picked instances. In **No exchange**, the heads are not exchanged as a first step of the algorithm. Algorithms **Initial**, **Growing** and **Dynamic** correspond to three different policies that have been used over time for the size of the sample set (see end of Section 3).

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 performances 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. Furthermore, in these cases, the initial exchange of heads can be counter-productive, see for example the last two lines of Table 3.

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. Babenhauserheide, A.: Automatic coordinated rebase with changeset evolution and mercurial (2020), https://blog.disy.net/hg-evolution/
2. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: Version control with subversion - next generation open source version control. O'Reilly (2004), http://www.oreilly.de/catalog/0596004486/index.html
3. Courtiel, J., Dorbec, P., Lecoq, R.: Theoretical analysis of git bisect. Tech. rep. (Nov 2021), hal.archives-ouvertes.fr/hal-03431454
4. Cáceres, M., Cairo, M., Mumey, B., Rizzi, R., Tomescu, A.I.: Sparsifying, Shrinking and Splicing for Minimum Path Cover in Parameterized Linear Time, pp. 359–376. https://doi.org/10.1137/1.9781611977073.18, https://epubs.siam.org/doi/abs/10.1137/1.9781611977073.18
5. De Rosso, S.P., Jackson, D.: What's wrong with git?: a conceptual design analysis. In: Hosking, A.L., Eugster, P.T., Hirschfeld, R. (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. pp. 37–52. ACM (2013). https://doi.org/10.1145/2509578.2509584
6. Dillworth, R.P.: A decomposition theorem for partially ordered sets. Annals of Mathematics **51**(1), 161–166 (1950). https://doi.org/10.2307/1969503
7. Felsner, S.: On-line chain partitions of orders. Theoretical Computer Science **175**(2), 283–292 (1997). https://doi.org/https://doi.org/10.1016/S0304-3975(96)00204-6, https://www.sciencedirect.com/science/article/pii/S0304397596002046
8. Grune, D.: Concurrent versions system, a method for independent cooperation. Tech. Rep. 113, Vrije Universiteit Amsterdam (1986)
9. Levenberg, J.: Why google stores billions of lines of code in a single repository. Commun. ACM **59**(7), 78–87 (2016). https://doi.org/10.1145/2854146

10. Mackall, O.: Towards a better scm: Revlog and mercurial. Tech. rep. (2005), http://selenic.com/mercurial
11. Rochkind, M.J.: The source code control system. IEEE Trans. Software Eng. **1**(4), 364–370 (1975). https://doi.org/10.1109/TSE.1975.6312866
12. Tichy, W.F.: Design, implementation, and evaluation of a revision control system. In: Ohno, Y., Basili, V.R., Enomoto, H., Kobayashi, K., Yeh, R.T. (eds.) Proceedings of the 6th International Conference on Software Engineering. pp. 58–67. IEEE Computer Society (1982), http://dl.acm.org/citation.cfm?id=807748
13. Torvalds, L.: Git - tree history storage tool. Tech. rep.