Dichotomy for reachability and label synchronisation in large version-controlled repositories

Laurent Bulteau, Pierre-Yves David, Florian Horn, Euxane Tran-Girard

To cite this version:

Laurent Bulteau, Pierre-Yves David, Florian Horn, Euxane Tran-Girard. Dichotomy for reachability and label synchronisation in large version-controlled repositories. 2024. hal-04778558

HAL Id: hal-04778558

https://hal.science/hal-04778558v1

Preprint submitted on 12 Nov 2024

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

Dichotomy for reachability and label synchronisation in large version-controlled repositories

Laurent Bulteau[1][0000−0003−1645−9345], Pierre-Yves David[2][0009−0005−3618−3560], Florian Horn[3][0000−0001−8872−4705], and Euxane Tran-Girard[1,2][0009−0003−4190−7151]

[1] LIGM, CNRS, Université Gustave Eiffel, France
{laurent.bulteau,euxane.trangirard}@univ-eiffel.fr
[2] Octobus, France pierre-yves.david@octobus.fr
[3] IRIF, CNRS, Université Paris Cité, France florian.horn@irif.fr

Abstract. DAGs are commonly used for modelling successive versions of a project in systems such as Git or Mercurial, where each version (node) is based on one or two former versions (with arcs from each node to its former versions). Some specificities make these graphs unique: they grow in "append-only" mode, i.e. nodes are created with in-degree 0, so the out-degree of existing nodes remains constant. Also, several copies of the graph grow in parallel, with synchronisation steps when an agent sends their new nodes to other agents.

For the largest graphs, which can reach up to several million revisions, sub-linear algorithms become necessary for recurring tasks, creating a need for a pre-computed index that can grow and be shared along with the graph nodes. In this paper we focus on the *reachability* problem (deciding if a path exists between two nodes), as well as *label discovery*: if each agent has a *label* attached to each node, determine the set of nodes with distinct labels between two agents. Algorithms are to be designed not only in a dynamic setting, i.e. allowing for node insertions, but also with the multi-agent constraint that different agents, having received different nodes and common nodes in different orders, may need to share large sets of nodes among themselves. Very efficient reachability indices exist in static DAGs, but they often do not apply well to rapidly-growing graphs in a multi-agent model. We present a framework allowing for fast algorithms for both tasks, using a compact index (with a few bytes per node in practice) that can easily be updated when the graph grows and nodes are shared. At the core of our approach lies the notion of *ranges*, i.e. specific sets of nodes that admit a concise representation and can be partitioned (split) into smaller ranges.

An implementation of our algorithms is available at https://doi.org/10.5281/zenodo.10715742.

Keywords: Merkle Graph · Reachability · Version-control systems

1 Introduction

Version-control systems are designed to help multiple developers work in parallel on a common project, sharing their progress as seamlessly as possible. In modern systems such as Mercurial or Git, this is achieved by using local *revision graphs* on a shared *repository*: each time a developer wants to save their progress, they add a new *revision* —a snapshot of their current version of the project— to their own local revision graph, pointing to the previous revision(s) that served at the basis for this new version. These revisions are then shared among developers, often but not always through a server centralizing all data.

The data-structure for local revision graphs is a *Merkle DAG*, which is updated from the sources: new revisions are sources with outgoing arcs to the revision(s) from which they were derived. A global repository is a set of consistent revision graphs: if two revision graph in the same repository share a common node u, they also share the nodes upon which u is based and, recursively, all reachable nodes from u.

Larger projects can involve hundreds of developers over many years with many sub-projects being developed in parallel. Such repositories can grow to millions of nodes, quickly leading to performance issues for complex operations even when the underlying algorithms are linear in time. In this paper, we propose a dichotomy framework which allows us to *split* sets of nodes to recursively explore subsets without having to enumerate the whole graph. We explore two concrete applications of this principle: the *reachability* test, which asks whether there is a dependency path between two revisions; and *label discovery*, which asks on which nodes two agents have added different *labels*, which are pieces of data associated to nodes after their creation.

It is possible to get sub-linear complexity for reachability testing on a specific graph by using a precomputed index, such as 2-hop. This is not enough for our purposes as such an index could be obsolete as soon as a new node is inserted. Also, there is no guarantee in general that two agents reaching the same revision graph from different histories would get to the same index. Formally, we consider that an index is a function that gives a value for each revision in a graph, and we say that an index is *coherent* if the stored value for a specific node is constant over time and over all graphs in a given repository.

Example 1 (Node Identifier). The simplest identifier for a node is its *insertion number*. However it is not coherent since different agents may receive common nodes in different orders. One can define a coherent *hash identifier* with a perfect hash[4] of its content, including the hash identifiers of the nodes on which it is based. The consistency property of repositories ensures that each node has a unique hash identifier shared by all agents. In Mercurial, both identifier systems are maintained: the insertion numbers yield better local performances, and the hashes are used whenever nodes are exchanged between agents.

[4] Mercurial uses 160-bit hashes, so the chances of a collision are infinitesimal.

We also consider the label discovery problem as a test case for our dichotomy framework. The idea is to compute, exchange, and cache hash values for the labels of carefully chosen sets of nodes, so that only the modified sets need to be exchanged between two successive synchronisations. For this application the dichotomy algorithm must be able to pinpoint an edited label using as few cuts as possible, and conversely all cached sets containing any given node must be accessible easily in order to keep the cache up to date after label edits.

Previous Works Representing the causal history of revisions using a DAG structure has been suggested and formalised by Plaice and Wadge [14] and implemented in current distributed version control systems (DVCS) such as Mercurial [13] and Git [9] as Merkle graphs. Several problems specific to DVCS have been studied in the literature. The problem of finding optimal search strategies using the graph structure for the purpose of regression testing has been implemented in Git and Mercurial and studied by Bendík et al. [3] and Courtiel et al. [7] using a dichotomy approach (unrelated to this paper). The problem of graph discovery, i.e. finding new nodes between two instances of a revision graph for the purpose of synchronisation, has been formulated and studied by Bulteau et al. [4].

The problem of label discovery, i.e. finding the differences between two labellings of a revision graph for the purpose of synchronisation, has not, to our knowledge, been extensively studied before. An experimental implementation of a restricted form of label discovery has been proposed in the *Evolve* extension of Mercurial [8] for synchronising obsolescence markers attached to nodes. Label discovery can be seen as a graph generalization of the file synchronisation problem, cf. the rsync algorithm (Tridgell et al. [18]).

The problem of reachability (or path existence query) in unlabeled directed graphs (acyclic or not) has been extensively studied and iterated upon [10] [21]. The main indexing approaches rely on chain cover (Jagadish [11], Chen and Chen [5], both dynamic), tree cover with interval labelling (Agrawal et al. [2], Yildirim et al. [20] (dynamic)), approximate transitive closure (Wei et al. [19] (dynamic), Su et al. [17]), and 2-hop (Cohen et al. [6], Zhu et al. [22] (dynamic), Lyu et al. [12] (dynamic)). The dynamic methods allow index updates under arbitrary node addition and deletion, without coherence restrictions. In practice, reachability tests for Merkle graphs have been implemented using chain cover in the Matrix communication protocol [1], and as pruned DFS in Mercurial [15] and Git [16], opportunistically using topological order, level and date information.

Our results We propose several algorithms leading to efficient solutions for reachability testing and label discovery through dichotomy. At the core of our approach lies the notion of *range*, i.e. a set of nodes that can be described concisely. Ranges are sets that can be defined as the first k nodes seen in a specific traversal of the reachable nodes from some node u, the *stable-tail sort* of u. We prove sublinear bounds on the depth of the dichotomy search tree, as well as other sublinear guarantees in specific cases (merge-free graphs). We evaluate the performances of our algorithms on a large dataset of revision graphs, and in particular for the reachability problem in a (single-agent) dynamic setting, we

run comparisons against implementations of 2-Hop and BFL algorithms, where our algorithm is competitive on large graphs with the advantage of having a coherent index. Due to space constraints, proofs and detailed experimental results are deferred to the appendix.

2 Definitions, notations, and general properties

In this paper, we use the Python notation for lists to manipulate *ordered sets*: elements are written within brackets separated by commas; the cardinal of L is $|L|$; its $i+1$th element is $L[i]$; the concatenation of two disjoint lists K and L is $K+L$. We also use usual set notations: $x \in L$ means that x is an element of L and $L \setminus K$ is the ordered set obtained from L by removing each element of K.

Definition 1 (*Revision graph*, *Repository*). *A* revision graph *is a labeled directed acyclic graph* $G = (V, E, \mathrm{num})$ *with out-degree at most 2, where* num *is an anti-topological bijection* $V \to [1, \ldots, |V|]$*, i.e.* $\mathrm{num}(u) > \mathrm{num}(v)$ *for each arc* $u \to v$*. Nodes of* V *are sometimes called* revisions*. A node* u *is an in-neighbour of* v *and* v *is an out-neighbour of* u *if there is an arc* $u \to v$ *in* E*. A node* v *is reachable from* u *if there is a path* $u \to^* v$ *(possibly with* $u = v$*). A node without in-neighbours is a source and a node without out-neighbours is a sink. A node with two out-neighbours is a merge.*

A repository is a collection of revision graphs where common nodes have the same out-neighbours: if G *and* H *are two revision graphs with common node* u *and* $u \to v \in E_G$*, then* $u \to v \in E_H$*.*

Remark 1. A source of confusion in the study of VCSs is that the dependency order is opposite to the topological order: new nodes are sources with transitions to existing nodes rather than sinks with transitions from existing nodes. Graph theory uses the parent-child relationship according to the topological order (if $u \to v$, then u is the parent of v), while VCS studies use the terms *parent* and *child* according to the dependency order (if $u \to v$, then u is the child of v). To avoid confusion, we do not use these terms, hence *in-* and *out-neighbours*.

We first introduce low-level concepts for nodes that give the necessary foundations for our dichotomy algorithms. These concepts are presented in Figure 1.

Definition 2 (Node properties). *For any node* u*, the reachable set* of u*, denoted* $\overrightarrow{\mathcal{R}}(u)$*, is the set of nodes that are reachable from* u*; the* rank *of* u*, denoted* $\mathrm{rank}(u)$*, is the size of its reachable set:* $\mathrm{rank}(u) = |\overrightarrow{\mathcal{R}}(u)|$*. Among out-neighbours of* u*, the tail neighbour, denoted* $\mathrm{t}(u)$*, is the one with larger rank (or lower hash identifier, cf. Example 1, in case of ties) and the exclusive neighbour is the other one, if any, and is denoted* $\mathrm{x}(u)$*. The tail path* of u*, denoted* $\mathrm{tail}(u)$*, is defined recursively as* $[u]$ *if* u *is a sink and* $[u] + \mathrm{tail}(\mathrm{t}(u))$ *if it has at least one out-neighbour.*

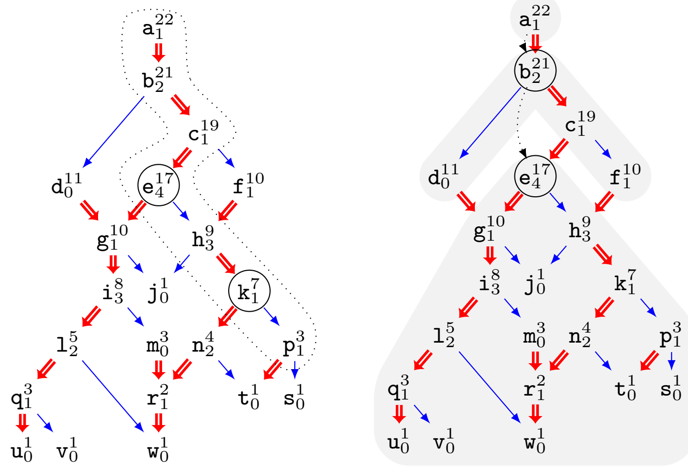


Fig. 1. A revision graph with 22 nodes. Arcs to the tail neighbour are doubled (red), others (blue) are to the exclusive neighbour. Super- and sub-scripts are respectively the rank and power of each node. Left: the switch list from $\mathtt{a}$ to $\mathtt{p}$ is $[\mathtt{e}, \mathtt{k}]$ (the corresponding path is outlined). Right: following dotted arcs, we have $\mathrm{anc}(\mathtt{a}) = \mathtt{b}$, $\mathrm{anc}(\mathtt{b}) = \mathtt{e}$, $\mathrm{anc}(\mathtt{e}) = \perp$. This yields the partition of $\overrightarrow{\mathcal{R}}(\mathtt{a})$ into canonical sets $\mathcal{C}(\mathtt{a}) = \{\mathtt{a}\}$, $\mathcal{C}(\mathtt{b}) = \{\mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{f}\}$ and $\mathcal{C}(\mathtt{e}) = \overrightarrow{\mathcal{R}}(\mathtt{e})$ (see also Split Rule 1).

Revision graphs are growing by adding new sources to the graph, while existing nodes and arcs remain unchanged. In particular, the out-neighbours (tail and exclusive) of a node, its hash id and rank stay constant after its creation. These values are computed and stored when the node is first created (*commit*).

We now define the *anchor* of a node, a notion that is central to our dichotomy algorithm, using a log-based *power* function.

Definition 3 (*Power, anchor, canonical set*). *The* power of a node u, *denoted* $\pi(u)$, *is the largest bit that changes between the binary representations of* $\mathrm{rank}(u)$ *and* $\mathrm{rank}(\mathrm{t}(u))$. *Formally, by convention, if* u *is a sink,* $\pi(u) = 0$. *Otherwise,* $\pi(u)$ *is the largest integer such that there exists an integer* x *with:*

$$\mathrm{rank}(u) \geq x \cdot 2^{\pi(u)} > \mathrm{rank}(\mathrm{t}(u))$$

The anchor of a node u, *denoted* $\mathrm{anc}(u)$, *is the first node* v *of* $\mathrm{tail}(u) \setminus \{u\}$ *such that* $\pi(v) \geq \pi(u)$; *if no such node exists, we write* $\mathrm{anc}(u) = \perp$.

The canonical set of a node u *is the set* $\mathcal{C}(u) = \overrightarrow{\mathcal{R}}(u) \setminus \overrightarrow{\mathcal{R}}(\mathrm{anc}(u))$ *(using* $\overrightarrow{\mathcal{R}}(\perp) = \emptyset$*).*

The definition of power ensures that the canonical set is very small for most nodes, but grows exponentially as we follow chains of anchors (see Figure 1).

Finally, we introduce the notion of switch list, which helps us bound the complexity of our enumeration algorithm and the depth of the dichotomy.

Definition 4 (*Switch list*). *Let u, v be a pair of nodes with $v \in \overrightarrow{\mathcal{R}}(u)$. The* switch list *from u to v, denoted $\sigma(u,v)$ is defined as follows:*

- *if $u = v$, $\sigma(u,v) = []$;*
- *if $v \in \overrightarrow{\mathcal{R}}(\mathrm{t}(u))$, $\sigma(u,v) = \sigma(\mathrm{t}(u), v)$;*
- *otherwise, $\sigma(u,v) = [u] + \sigma(\mathrm{x}(u), v)$.*

The switch distance *from u to v, denoted $\mathrm{d_S}(u,v)$ is the length of $\sigma(u,v)$.*

The switch list can be seen as a characteristic of a specific path from u to v: it always goes to the tail neighbour of the current vertex until it reaches the first switch; in that case, it goes to the exclusive neighbour and discards that switch. The fact that the tail neighbour has the larger rank allows us to bound the switch distance between any two nodes.

Lemma 1. *For any nodes u and v, $v \in \overrightarrow{\mathcal{R}}(u)$, the switch distance between u and v is at most $\sqrt{2\,\mathrm{rank}(u)}$.*

3 Stable-tail sort and ranges

In order to allow dichotomy algorithms on sets of nodes, we introduce linear orderings over nodes. The first constraint is to make them coherent (i.e. different agents have the same view on common nodes), and as *stable* as possible, i.e. node insertions in different revision graphs should not completely shuffle the permutation. We thus introduce the *stable-tail sort* below, defined as a coherent topological ordering of the reachable set of any node (in any repository, all revision graphs having the same node u have the same $\mathrm{STS}(u)$). See Figure 2 for an illustration.

Definition 5 (STS : Stable-Tail Sort). *For a sink u we define $\mathrm{STS}(u) = [u]$. If u has one out-neighbour $\mathrm{t}(u)$, then $\mathrm{STS}(u) = [u] + \mathrm{STS}(\mathrm{t}(u))$. If u is a merge then we write $\mathrm{excl}(u) = \mathrm{STS}(\mathrm{x}(u)) \setminus \overrightarrow{\mathcal{R}}(\mathrm{t}(u))$ and $\mathrm{STS}(u) = [u] + \mathrm{excl}(u) + \mathrm{STS}(\mathrm{t}(u))$. The lists $\mathrm{excl}(u)$ and $\mathrm{STS}(\mathrm{t}(u))$ are respectively called the* exclusive *and* tail *parts of $\mathrm{STS}(u)$.*

Remark 2. A node's Stable-Tail Sort of its reachable set is *not* the projection of a graph-wide order on $\overrightarrow{\mathcal{R}}(u)$.

Intuitively the STS is a variation of the depth-first traversal of the reachable set, where instead of skipping nodes that have already been seen, one skips nodes that *will be seen again*. Indeed, switching the exclusive and tail parts in Definition 5 would define exactly the depth-first traversal. Our objective is, for any two nodes, to have a large common suffix. In particular for merges with a small exclusive part, a large suffix comes from $\mathrm{STS}(\mathrm{t}(u))$, which in turn has a large common suffix with its own tail neighbour, etc.

Although we precisely aim at avoiding any exhaustive enumeration of subsets of nodes, such enumeration is necessary in some cases, especially for short

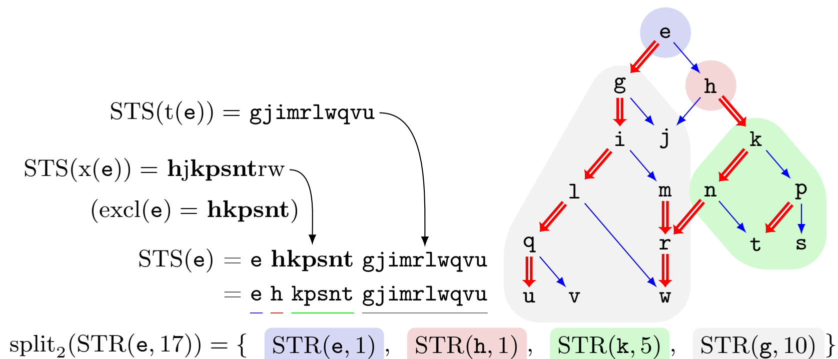


Fig. 2. Top: construction of $\mathrm{STS}(e)$ from its out-neighbours: the whole $\mathrm{STS}(\mathtt{g})$ is used as a suffix, while the exclusive part (bold nodes) is listed according to $\mathrm{STS}(\mathtt{h})$. Bottom: when splitting the corresponding range, the exclusive part is greedily partitioned into $\mathrm{STR}(h, 1) \cup \mathrm{STR}(k, 5)$.

prefixes of $\mathrm{STS}(u)$, and needs to be as efficient as possible. A straightforward implementation would actually use a depth-first-search, remembering visited nodes and entering a node only if all its in-neighbours have been visited. However, the in-neighbourhood of a node is not stored explicitly in memory (it can be computed in $O(n)$, but this would be before the enumeration starts). Similarly, one could also run a regular DFS starting with the tail neighbour and then swap the exclusive and tail parts for each merge node, but again, this strategy would require to visit the whole set $\overrightarrow{\mathcal{R}}(u)$ before starting the actual enumeration.

For a more efficient enumeration of prefixes of the STS of a node, and also have a faster access to any node by index, we pre-compute the *leaps* of u: all intervals of nodes in $\mathrm{STS}(\mathrm{x}(u))$ that also appear in $\overrightarrow{\mathcal{R}}(\mathrm{t}(u))$. Leaps can be stored concisely as pairs of integers. Then, $\mathrm{excl}(u)$ can be obtained by (1) enumerating $\mathrm{STS}(\mathrm{x}(u))$, (2) removing the positions contained in a leap of u and (3) applying a cut-off at the correct length: $|\mathrm{excl}(u)| = \mathrm{rank}(u) - \mathrm{rank}(\mathrm{t}(u)) - 1$ (see Algorithm 1). This straightforward algorithm can further be turned into an efficient enumeration algorithm.

Theorem 1. *Let u be a node. There exists an enumeration algorithm for $\mathrm{STS}(u)$ using an index containing rank and leap data for all nodes, and taking time $O(1 + d_p(1 + \ell_p))$ for each position p, where (denoting $v = \mathrm{STS}(u)[p]$):*

- *$d_p = d_S(u, v)$ is the switch distance from u to v,*
- *ℓ_p is the maximum number of leaps among switches of $\sigma(u, v)$*

Although in the worst case ℓ_p can be linear and d_p can grow in $\sqrt{n}$, most merges have no leaps at all ($\mathrm{excl}(u)$ is a prefix of $\mathrm{STS}(\mathrm{x}(u))$), and enumerating each position takes less than 2 look-ups in most cases.

Using the Stable-Tail Sort, we can move on to defining *ranges* (i.e., distinguished subsets) of nodes, and subsequent splitting rules.

```
def STS(u):
    if u = ⊥ then
        return []
    X ← STS(x(u))
    foreach interval I in leaps(u) do
        remove elements with index in I from X
    X ← X[0 : rank(u) − rank(t(u)) − 1]
    return [u] + X + STS(t(u))
```

Algorithm 1: A recursive algorithm for STS, using rank and leap data to build the exclusive part $X = \mathrm{excl}(u)$. We write $\mathrm{x}(u), \mathrm{t}(u) = \perp$ if the corresponding neighbour is undefined.

Definition 6 (STR: Stable-Tail Range). *The* stable-tail range *(or* range *for short) with head u and length $k > 0$, denoted* $\mathrm{STR}(u, k)$ *is the set of nodes* $\{\mathrm{STS}(u)[i] \mid 0 \le i < k\}$. *A range is called* atomic *if $k = 1$,* full *if $k = |\overrightarrow{\mathcal{R}}(u)|$,* canonical *if $k = |\mathcal{C}(u)|$. It is* short *if $k \le |\mathcal{C}(u)|$, and* long *otherwise (in particular, atomic and canonical ranges are short, full ranges are long unless* $\mathrm{anc}(u) = \perp$*).*

We now present range-splitting algorithms, i.e. we aim at partitioning the nodes in a range into a (small) number of smaller ranges. We present below two distinct *splitting rules*, combined into $\mathrm{split}(S) = \mathrm{split}_1(S)$ if S is a long range, and $\mathrm{split}(S) = \mathrm{split}_2(S)$ if S is short.

Splitting Rule 1 (Long ranges)**.** *For any range S with head u, let* $\mathrm{split}_1(S) = \{S\}$ *if S is short, and* $\mathrm{split}_1(S) = \{\mathcal{C}(u)\} \cup \mathrm{split}_1(S \cap \overrightarrow{\mathcal{R}}(\mathrm{anc}(u)))$ *otherwise (cf Figure 1).*

The second splitting rule, stated below, uses the notion of *longest range prefix* of some list L, which can be defined as the longest common prefix between L and $\mathrm{STS}(L[0])$ (it is indeed a range since it is a prefix of some STS).

Splitting Rule 2 (Short ranges)**.** *For a range S with head u, let X be the prefix of* $\mathrm{excl}(u)$ *containing only nodes of S, and* $T = S \cap \overrightarrow{\mathcal{R}}(\mathrm{t}(u))$ *(in particular,* $X = \mathrm{excl}(u)$ *whenever T is not empty, and* $S = \{u\} \cup \mathrm{set}(X) \cup T$*). If X is not empty, we define successive ranges $X_1, \ldots, X_k$ such that X_i is the longest range prefix of* $X \setminus (X_1 \cup \ldots \cup X_{i-1})$.

Overall, $\mathrm{split}_2(S) = \{\{u\}, X_1, \ldots, X_k, T\}$ *(cf Figure 2).*

Intuitively, split_1 cuts long ranges at successive anchors to produce few ranges, all of them short (mostly even canonical). For short ranges, no anchor is available, and split_2 splits greedily to produce at least two ranges, short or long.

Definition 7 (Search-Tree and Dichotomy search). *The* Search Tree *for a range S, denoted T_S, is the tree with root labelled with range S and, if S is not atomic, the subtree T_{S_i} for each* $S_i \in \mathrm{split}(S)$. *For a node u, we write T_u for the tree* $T_{\overrightarrow{\mathcal{R}}(u)}$.

A dichotomy search *for a node $v \in S$ is the path from the root T_S to the leaf $T_{\mathrm{STR}(v,1)}$ (such a path always exists since the splitting function on non-atomic ranges always yields a partition into strictly smaller ranges, so the leaves of T_S are exactly the nodes $T_{\mathrm{STR}(v,1)}$ for $v \in S$).*

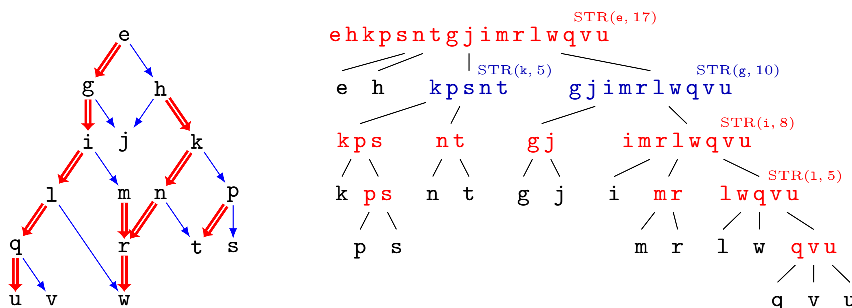


Fig. 3. Search tree (right) for the range $\overrightarrow{\mathcal{R}}(e) = \mathrm{STR}(\mathtt{e}, 17)$ from the revision graph given on the left. Atomic ranges are shown in black, long ranges in blue and short ranges in red. The range content is enumerated explicitly (in STS order), and the notation as $\mathrm{STR}(x, i)$ is given as superscript for the longer ones. Here $T_{\overrightarrow{\mathcal{R}}(\mathtt{e})}$ has size 27, depth 6 and degree 4.

Given a graph G, we are interested in the following metrics: the *worst-case depth* is the maximum depth (or height) of T_u for $u \in V$. The *worst-case degree* is the maximum number of sub-graphs of any node in these trees. The *total range count* is the overall number of distinct ranges seen in these trees. Both degree and depth need to be optimized, as the total time cost taken by a dichotomy search is typically the sum of all degrees along the path, which can be bounded in $O(\text{degree} \times \text{depth})$. The total range count needs also to be minimized, in order to optimize storage and caching: whenever meta-data needs to be stored for ranges in internal nodes of the tree (e.g. in label discovery, see Section 4.2), the number of cached ranges per node needs to be bounded on average in order to maintain a linear space requirement.

Theorem 2. *A graph with n nodes has worst-case depth $O(\sqrt{n}\log(n))$*

The worst-case degree can trivially be bounded by n. Function split_1 can be shown to have degree $\leq \log_2 n + 1$ but there is no theoretical bound for split_2.

4 Algorithms for Mercurial

4.1 Range-based reachability algorithm

The dichotomy framework defined in Section 3 can be used as a search-tree for reachability tests. We use an approximate, constant-time oracle for range

containment, i.e. a function that, given a node x and a range S returns `yes` ($x \in S$), `no` ($x \notin S$) or `maybe`. We run the oracle on range $S = \overrightarrow{\mathcal{R}}(x)$: we can have an answer if it returns `yes` or `no`. Otherwise, we continue recursively on each subrange in split(S) until either a `yes` is found, or no range remains (in which case the query is negative). For a range S with head u and candidate node x, the oracle applies the following rules in order, using pre-computed values for rank and minrank (where $\text{minrank}(u) = \min\{\text{rank}(v) \mid v \in \mathcal{C}(u)\}$) :

1. If $u = x$, return `yes`;
2. If $\text{rank}(x) \geq \text{rank}(u)$ or $\text{num}(x) \geq \text{num}(u)$, return `no` [5]
3. If S is short and $\text{rank}(x) < \text{minrank}(u)$, return `no`;
4. Otherwise, return `maybe`.

The complexity of this reachability algorithm depends heavily on the shape of the graph. In the worst case, a large fraction of canonical ranges have nodes with ranks spanning an interval containing rank(x), which would lead to a linear look-up time. We can however give a better bound on merge-free graphs.

Theorem 3. *The index of the range-based reachability algorithm is coherent. If G is a merge-free graph, then the index is linear and has query time $O(\log^2(n))$.*

4.2 Label discovery

In the label discovery problem, two agents have revision graphs with the same set of nodes V, but not necessarily the same revision order. Furthermore, each agent has a label function, denoted respectively ℓ_1 and ℓ_2, assigning a label of some discrete type to each node. The goal is to determine the set $\Delta \subseteq V$ of nodes where ℓ_1 is different than ℓ_2. We want to minimize the number of exchanges between the agents as well at the total amount of information exchanged, as a function of $|\Delta|$, the number of differences between the two agents.

For each agent i we write $\mathrm{L}_i(S)$ for a hash mixing all labels of all nodes in some range S[6]; such values are assumed to be pre-computed whenever needed. We compute the deviations by maintaining a set of candidate ranges. Initially, the candidate ranges are the reachable sets of the sources of the graph. Then, we process the ranges in the candidate set successively as follows:

- If $\mathrm{L}_1(S) = \mathrm{L}_2(S)$, nothing (there are no differences in S);
- otherwise, if S is atomic, we add the single revision in S to Δ;
- otherwise, we add each range in split(S) to the set of candidate ranges.

Theorem 4. *Consider a single-source revision graph with worst-case depth H ($H = O(\sqrt{n}\log n)$ by Theorem 2) and worst-case degree D. The number of round-trips of the algorithm is at most H, and the total number of exchanged values is at most $DH|\Delta|$.*

[5] the second condition uses the non-coherent num function, although we do not count it as part of the index since it is already part of the input graph

[6] we assume that the hash values are large enough to ignore collisions: $\mathrm{L}_1(S) = \mathrm{L}_2(S)$ if and only if $\ell_1(u) = \ell_2(u)$ for all nodes $u \in S$.

In practice, computing the hashes of many different ranges is time-consuming, and we obtain a better balance between network exchanges and computation time with the two following modifications:

- long ranges are not exchanged, they are immediately split instead;
- for a short range S with head u, $S \subseteq \mathcal{C}(u)$, exchange $\mathrm{L}(\mathcal{C}(u))$ instead of $\mathrm{L}(S)$

The second modification leads to transient false positives (some ranges may have different exchanged hash value although the labels are identical): they are eliminated in subsequent rounds, when the partitioned is refined to smaller ranges. This allows us to use an index containing $L(S)$ for canonical and atomic ranges only, so $\leq 2n$ hashes in total.

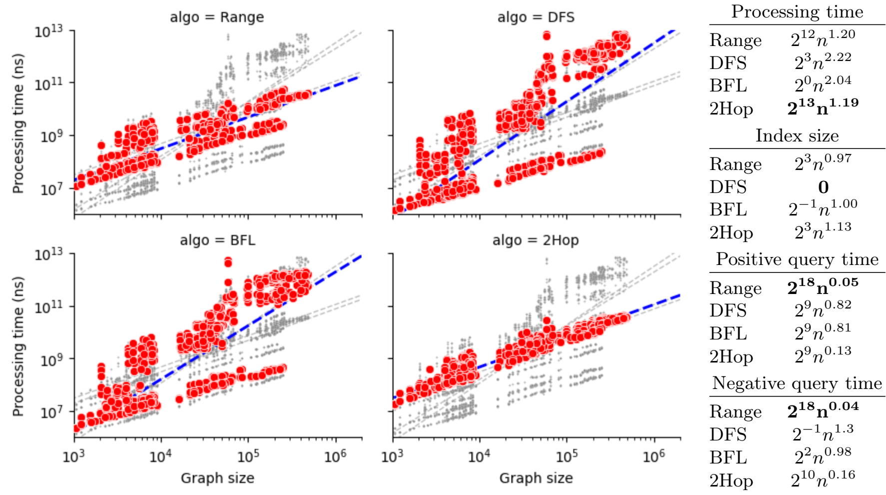
5 Experimental evaluation

We evaluate our algorithms in two phases. First, we run implementation-independent versions of our STS and STR algorithms in order to estimate, in real data, various statistics influencing our algorithms performances. Then, we compare running times and memory usage for our reachability algorithms with literature algorithms on largest graphs.

First, for our dichotomy algorithm, when visiting a search tree for $\overrightarrow{\mathcal{R}}(u)$, we have a typical height of $1.3\log(\mathrm{rank}(u))$, and each sub-range S has degree at most $2\log(|S|)$. Moreover, we use less than $2n$ distinct subranges in most graphs, allowing efficient caching for range-relative data. Reachability tests use an index of size typically $2n$ and answer with $2\log n$ oracle calls, although this value can be multiplied by 100 for the most complex graphs. Similarly, Label Discovery is performed in less than $2\log(n)$ round-trips on average. See Appendix D.1 for detailed statistics.

Reachability Algorithms. As noted in the Previous Works section, most known reachability algorithms use non-coherent indexes, which can be a major drawback in our context for external reasons (e.g. node insertion, but also data compression or cache management). We nonetheless compare our reachability algorithms with two literature algorithms that can be implemented to support node insertions in sublinear time: 2Hop (specifically the *TOL* algorithm [22]), and BFL [17]. (See Appendix D.3 for details).

The results are given in Figure 4. A simple DFS gives optimal results for many instances, but some instances yield prohibitively high costs. BFL behaves similarly. Our range algorithm and 2Hop give comparable asymptotic behavior for processing the graph. However, this test only evaluates node insertions on a single graph. When synchronising nodes between agents, the coherent STR index can be shared along with node data without any additional computation, which is a major advantage (although hard to quantify) for this algorithm in our setting.



Processing time	
Range	$2^{12}n^{1.20}$
DFS	$2^{3}n^{2.22}$
BFL	$2^{0}n^{2.04}$
2Hop	$\mathbf{2^{13}n^{1.19}}$
Index size	
Range	$2^{3}n^{0.97}$
DFS	$\mathbf{0}$
BFL	$2^{-1}n^{1.00}$
2Hop	$2^{3}n^{1.13}$
Positive query time	
Range	$\mathbf{2^{18}n^{0.05}}$
DFS	$2^{9}n^{0.82}$
BFL	$2^{9}n^{0.81}$
2Hop	$2^{9}n^{0.13}$
Negative query time	
Range	$\mathbf{2^{18}n^{0.04}}$
DFS	$2^{-1}n^{1.3}$
BFL	$2^{2}n^{0.98}$
2Hop	$2^{10}n^{0.16}$

Fig. 4. Processing time for tested reachability algorithms as a function of the graph size. Right: typical trends for time and memory over the whole benchmark, obtained using min-square linear regressions in log scale (drawn with dashed lines on the corresponding graphs for processing times).

6 Conclusion

We presented a dichotomy framework tailored for revision graphs, based on stable orderings of reachable sets for each nodes. This framework requires only a light-weight, coherent index, and allows to perform several tasks in sub-linear time. In particular we obtain a competitive algorithm for reachability queries, even against state-of-the-art algorithms that are not bound by the coherence constraints.

In future works we aim at extending this framework to other applications, as well as investigating the many problems related to Version-Control Systems that remain mostly unexplored from a theoretical viewpoint, starting with efficient data-structures to store, retrieve or exchange any set of nodes.

Bibliography

[1] Synapse internal docummentation: State resolution: The auth chain difference algorithm. https://matrix-org.github.io/synapse/v1.98/auth_chain_difference_algorithm.html#chain-cover-index, 2021.

[2] Rakesh Agrawal, Alexander Borgida, and Hosagrahar Visvesvaraya Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *ACM SIGMOD Record*, 18(2):253–262, 1989.

[3] Jaroslav Bendík, Nikola Benes, and Ivana Cerna. Finding regressions in projects under version control systems. *arXiv preprint arXiv:1708.06623*, 2017.

[4] Laurent Bulteau, Pierre-Yves David, and Florian Horn. The problem of discovery in version control systems. *Procedia Computer Science*, 223:209–216, 2023.

[5] Yangjun Chen and Yibin Chen. An efficient algorithm for answering graph reachability queries. In *2008 IEEE 24th International Conference on Data Engineering*, pages 893–902. IEEE, 2008.

[6] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5): 1338–1355, 2003.

[7] Julien Courtiel, Paul Dorbec, and Romain Lecoq. Theoretical analysis of git bisect. *Algorithmica*, pages 1–35, 2023.

[8] Pierre-Yves David. Hg evolve: Obsolescence marker discovery implementation. https://repo.mercurial-scm.org/evolve/file/11.1.1/hgext3rd/evolve/obsdiscovery.py, 2017.

[9] Junio C Hamano. Git–a stupid content tracker. *Proc. Ottawa Linux Symposium*, 1:385–394, 2006.

[10] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms–a quick reference guide. *ACM Journal of Experimental Algorithmics*, 27:1–45, 2022.

[11] HV1093244 Jagadish. A compression technique to materialize transitive closure. *ACM Transactions on Database Systems (TODS)*, 15(4):558–598, 1990.

[12] Qiuyi Lyu, Yuchen Li, Bingsheng He, and Bin Gong. Dbl: Efficient reachability queries on dynamic graphs. In *Database Systems for Advanced Applications: 26th International Conference, DASFAA 2021, Taipei, Taiwan, April 11–14, 2021, Proceedings, Part II 26*, pages 761–777. Springer, 2021.

[13] Olivia Mackall. Towards a better SCM: Revlog and mercurial. *Proc. Ottawa Linux Symposium*, 2:83–90, 2006.

[14] John Plaice and William W Wadge. A new approach to version control. *IEEE transactions on Software Engineering*, 19(3):268–276, 1993.

[15] Georges Racinet. Mercurial lazy ancestor iterator rust implementation. https://repo.mercurial-scm.org/hg/file/6.6.3/rust/hg-core/src/ancestors.rs#l123, 2018.

[16] Derrick Stolee. Git mailing list: [RFC] generation number v2. https://lore.kernel.org/git/6367e30a-1b3a-4fe9-611b-d931f51effef@gmail.com/, 2018.
[17] Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. Reachability querying: Can it be even faster? *IEEE Transactions on Knowledge and Data Engineering*, 29(3):683–697, 2016.
[18] Andrew Tridgell, Paul Mackerras, et al. The rsync algorithm. 1996.
[19] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. Reachability querying: An independent permutation labeling approach. *Proceedings of the VLDB Endowment*, 7(12):1191–1202, 2014.
[20] Hilmi Yildirim, Vineet Chaoji, and Mohammed J Zaki. Dagger: A scalable index for reachability queries in large dynamic graphs. *arXiv preprint arXiv:1301.0977*, 2013.
[21] Chao Zhang, Angela Bonifati, and M Tamer Özsu. Indexing techniques for graph reachability queries. *arXiv preprint arXiv:2311.03542*, 2023.
[22] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. Reachability queries on large dynamic graphs: a total order approach. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1323–1334, 2014.

A Material for Sec. 2, Definitions, notations, and general properties

The following proposition is an important observation following directly from the definition of power (Definition 3).

Proposition 1. *If* $\mathrm{anc}(u) \neq \bot$*, then* $\pi(\mathrm{anc}(u)) > \pi(u)$.

Proof. By the definition of power, for any node v, $\mathrm{rank}(v) \geq x_v 2^{\pi(v)}$ with x_v some integer. Note that x_v is odd, since otherwise we could write $\mathrm{rank}(v) \geq \frac{x_v}{2} 2^{\pi(v)+1}$, and $\pi(v)$ would not be maximal. Similarly, we have

$$\mathrm{rank}(v) < (x_v + 1)2^{\pi(v)}$$

since otherwise we could write $\mathrm{rank}(v) \geq \frac{x_v+1}{2} 2^{\pi(v)+1}$.

Consider a node $v \in \mathrm{tail}(u) \setminus \{u\} = \mathrm{tail}(\mathrm{t}(u))$ with $\pi(v) = \pi(u) = p$. We show that v cannot be $\mathrm{anc}(u)$. From the definition of π, we have $\mathrm{rank}(u) \geq x_u 2^p$, $\mathrm{rank}(\mathrm{t}(u)) < x_u 2^p$ and $\mathrm{rank}(v) \geq x_v 2^p$. Since $v \in \overrightarrow{\mathcal{R}}(\mathrm{t}(u))$, we have $\mathrm{rank}(v) < x_u 2^p$ so $x_v < x_u$. Since both x_v, x_u are odd, there is an integer x' such that $x_v < 2x' < x_u$. Write $X = \{w \mid w \in \mathrm{tail}(u), \mathrm{rank}(w) \geq x' 2^{p+1}\}$. Set X is a prefix of $\mathrm{tail}(u)$ (since the rank is decreasing along the tail path). We have $u \in X$ (using $x_u > 2x'$) and $\mathrm{t}(u) \in X$ (using $\pi(u) = p$) but not v (using $x_v < 2x'$ and $\mathrm{rank}(v) < (x_v + 1)2^{\pi(v)}$ by the previous paragraph).

Let w be the node in X with minimum rank. Then $\mathrm{t}(w) \in \mathrm{tail}(u)$ and $\mathrm{t}(w) \notin X$, so $\mathrm{rank}(\mathrm{t}(w)) < x' 2^{p+1}$ and $\pi(w) \geq p + 1$. Overall, w appears before v in $\mathrm{tail}(u)$ and it has a strictly larger power: v cannot be the anchor of u, and $\pi(\mathrm{anc}(u)) > \pi(u)$. □

Definition 8 (*Switch-Successor List*). *For* $v \in \overrightarrow{\mathcal{R}}(u)$*, with switch list* $\sigma(u, v) = [s_1, \dots, s_k]$*, the* switch-successor list *of* v *wrt.* u *is the list of nodes* $[\mathrm{x}(v), \mathrm{t}(v), \mathrm{t}(s_k), \dots, \mathrm{t}(s_1)]$ *(from which non-existing nodes are removed).*

*Proof (**Proof of Lemma 1**).* For a merge node s, we write

Let $\sigma(u, v) = [s_1, s_2, \dots, s_k]$ be the switch list from u to v, with $k = d_S(u, v)$. We write $p_i = \mathrm{t}(s_i)$, and define sets $X_i = \overrightarrow{\mathcal{R}}(\mathrm{x}(s)) \setminus \overrightarrow{\mathcal{R}}(p_i)$ and $P_i = \overrightarrow{\mathcal{R}}(p_i) \setminus \overrightarrow{\mathcal{R}}(\mathrm{x}(s_i))$. Note that by definition of $\mathrm{x}(s_i)$ and $\mathrm{t}(s_i)$, we have $|P_i| \geq |X_i|$.

By definition of switch list, for each $1 \leq i \leq k$, we have $v \in X_i$ and $s_j \in X_i$ for $i < j \leq k$, so $|X_i| \geq k - i + 1$, which gives $|P_i| \geq k - i + 1$.

For $1 \leq i < j \leq k$, nodes in P_j are reachable from $\mathrm{x}(s_i)$ (they are reachable from p_j, which is an out-neighbour of s_j, which is itself reachable from $\mathrm{x}(s_i)$). Thus, sets P_i and P_j are pairwise disjoint for each pair $i < j$, and they are all reachable from u. Furthermore, nodes s_j and v are not part of any set P_i and

are also reachable from u. We thus get a lower-bound on the size of $\overrightarrow{\mathcal{R}}(u)$:

$$\begin{aligned}
\operatorname{rank}(u) &\geq \sum_{i=1}^{k} |P_i| + |\{s_1, \ldots, s_k, v\}| \\
&\geq k + 1 + \sum_{i=1}^{k} ((k-i)+1) \\
&= \binom{k+2}{2} \\
&\geq \frac{k^2}{2}, \text{ and} \\
d_S(u,v) &= k \\
&\leq \sqrt{2 \operatorname{rank}(u)}
\end{aligned}$$

We note that the bound $\operatorname{rank}(u) \geq \binom{d_S(u,v)+2}{2}$ is tight: for any $k > 0$, one can build a graph with a pair of nodes u, v satisfying $d_S(u,v) = k$ and $\operatorname{rank}(u) = \binom{k+2}{2}$, as can be seen in Figure 5.

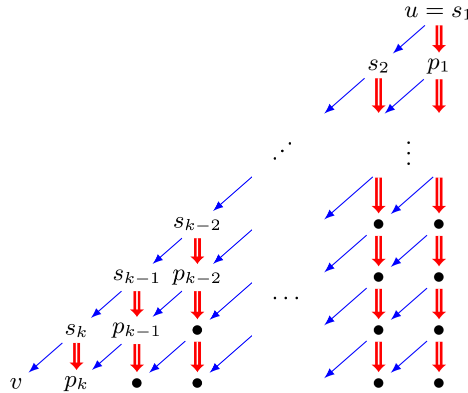


Fig. 5. Worst case for the number of switches between two revisions: the graph has $\binom{k+2}{2}$ vertices for a pair of nodes with switch distance k. We use notations u, v, s_i, p_i from the proof of Lemma 1; tail neighbour is directly below each node, and each exclusive neighbour is in diagonal to the left (the rank is equal for all pairs of neighbours, and we assume that the tie-breaking rule gives the desired property). Thus, the switch list is indeed $\sigma(u,v) = [s_1, \ldots, s_k]$ and $d_S(u,v) = k$. In other words, $d_S(u,v) = \Omega(\sqrt{\operatorname{rank}(u)})$ in this example.

B Material for Sec. 3, Stable-tail sort and ranges

B.1 Proof of Theorem 1

Before proving Theorem 1, we first give several useful observations for Stable-Tail Sorts, including the formal definition of a *leap* (Definition 9).

Proposition 2. $\mathrm{STS}(u)$ *is a topological sort of* $\overrightarrow{\mathcal{R}}(u)$

Proof. By induction, this is clear for out-degree 0 and 1 nodes. For a degree-2 node u and any arc $x \to y$ with $x \in \overrightarrow{\mathcal{R}}(u)$, either $x = u$ (in which case x is the first node, so before y), either x and y are both in the exclusive or both in the tail parts (in which case x is before y by induction), either they are in different parts but then only y can be in the tail part (if $x \in \overrightarrow{\mathcal{R}}(\mathrm{t}(u))$, then $y \in \overrightarrow{\mathcal{R}}(\mathrm{t}(u))$), in which case x is before y by construction. □

Proposition 3. *For any u and $v \in \mathrm{tail}(u)$,* $\mathrm{STS}(v)$ *is a suffix of* $\mathrm{STS}(u)$*.*

Proof. By induction on u. This is trivial for $u = v$, and follows directly from the definition for $v = \mathrm{t}(u)$. For any other $v \in \mathrm{tail}(u)$, we have $v \in \mathrm{tail}(\mathrm{t}(u))$, so $\mathrm{STS}(v)$ is a suffix of $\mathrm{STS}(\mathrm{t}(u))$ and of $\mathrm{STS}(u)$. □

Proposition 4. *Let $v \in \overrightarrow{\mathcal{R}}(u)$ with switch-successor list $[w_1, \ldots, w_k]$ wrt. u (cf Definition 8), and let Z be the suffix of* $\mathrm{STS}(u)$ *starting at v. Then Z can be written as a concatenation $Z = [v] + Z_{w_1} + \ldots + Z_{w_k}$ where each Z_{w_i}, if non empty, is a subsequence of* $\mathrm{STS}(w_i)$ *starting with w_i. Moreover, sets Z_{w_i} and $\overrightarrow{\mathcal{R}}(w_j)$ are disjoint for any $i < j$.*

Proof. By induction on v, starting with $v = u$. $\mathrm{STS}(u) = [u] + Z_{\mathrm{x}(u)} + Z_{\mathrm{t}(u)}$ where $Z_{\mathrm{x}(u)} = \mathrm{excl}(u)$ and $Z_{\mathrm{t}(u)} = \mathrm{STS}(\mathrm{t}(u))$. Also, $Z_{\mathrm{x}(u)}$ is disjoint from $\overrightarrow{\mathcal{R}}(\mathrm{t}(u))$.

For any $v \in \overrightarrow{\mathcal{R}}(\mathrm{t}(u))$, the suffix T of $\mathrm{STS}(u)$ starting at v is identical to the suffix of $\mathrm{STS}(\mathrm{t}(u))$ starting at v, and $\sigma(u, v) = \sigma(\mathrm{t}(u), v)$, so the same decomposition of Z holds for $\mathrm{t}(u)$ and u.

Otherwise, $v \in \overrightarrow{\mathcal{R}}(\mathrm{x}(u)) \setminus \overrightarrow{\mathcal{R}}(\mathrm{t}(u))$. Let Z' be the suffix of $\mathrm{STS}(\mathrm{x}(u))$ starting at v. By construction $Z = (Z' \setminus \overrightarrow{\mathcal{R}}(\mathrm{t}(u))) + \mathrm{STS}(\mathrm{t}(u))$. We have $\sigma(u, v) = [u] + \sigma(\mathrm{x}(u), v)$, so, writing $[w_1, \ldots, w_k]$ for the switch-successor list of v wrt. $\mathrm{x}(u)$, the switch-successor list of v wrt. u is $[w_1, \ldots, w_k, \mathrm{t}(u)]$. By induction $Z' = [v] + Z'_{w_1} + \ldots + Z'_{w_k}$. Write, for each $i \leq k$, $Z_{w_i} = Z'_{w_i} \setminus \overrightarrow{\mathcal{R}}(\mathrm{t}(u))$. So Z_{w_i} is indeed a subsequence of $\mathrm{STS}(w_i)$. Moreover, if Z_{w_i} is not empty, then $\overrightarrow{\mathcal{R}}(w_i) \not\subseteq \overrightarrow{\mathcal{R}}(\mathrm{t}(u))$ and $w_i \notin \overrightarrow{\mathcal{R}}(\mathrm{t}(u))$. So since Z_{w_i} starts with w_i, Z'_{w_i} also starts with w_i. Overall $Z = [v] + Z_{w_1} + \ldots + Z_{w_k} + \mathrm{STS}(\mathrm{t}(u))$, which gives a correct decomposition. Disjunction follows by induction, except for Z_{w_i} and $\overrightarrow{\mathcal{R}}(\mathrm{t}(u))$ that are disjoint by definition of Z_{w_i}. □

Definition 9 (leap). *We say that u has an ℓ*-leap at position p *(with $\ell \geq 1$ and $2 \leq p \leq |\mathrm{excl}(u)|$) if, for some integer q, we have* $\mathrm{excl}(u)[p-2] = \mathrm{STS}(\mathrm{x}(u))[q]$ *and* $\mathrm{excl}(u)[p-1] = \mathrm{STS}(\mathrm{x}(u))[q+1+\ell]$

We use elements $\text{excl}(u)[p-2]$ and $\text{excl}(u)[p-1]$ since they correspond respectively to $\text{STS}(u)[p-1]$ and $\text{STS}(u)[p]$, and the leap modifies the "expected value" of $\text{STS}(u)[p]$, that would have been $\text{STS}(\text{x}(u))[q+1]$ otherwise.

*Proof (**Proof of Theorem 1**).*
The algorithm is based on the following observation: for any two successive nodes v, w in $\text{STS}(u)$, i.e. with $v = \text{STS}(u)[p]$ and $w = \text{STS}(u)[p+1]$, we have $w \in \{\text{x}(v), \text{t}(v), \text{t}(s_1), \ldots, \text{t}(s_k)\}$ with the switch list $\sigma(u, v) = [s_1, \ldots, s_k]$. This is a direct corollary of Proposition 4. Thus, in addition to the current node v, the algorithm maintains the current switch list $\sigma(u, v)$, as well as the current position p_i of v in $\text{STS}(p_i)$ for each *level* i.

In order to compute the next element w, we update p_i (to a new value denoted p'_i) for each level. If $\text{STS}(s_i)[p'_i] = \text{t}(s_i)$, then the next element is $\text{t}(s_i)$. This condition is easily verified using the rank, since for any node x, $\text{t}(x)$ is at index $\text{rank}(x) - \text{rank}(\text{t}(x))$ in $\text{STS}(x)$. Otherwise, $w \in \text{excl}(s_i)$, and we move on to the next level with $s_{i+1} \in \text{STS}(\text{x}(s_i))$. By definition of leaps, $p'_{i+1} = p_{i+1} + (p'_i - p_i + L)$, where L is the sum of all leap lengths for leaps of s_i with positions between p_i (excluded) and p'_i (included). We repeat this step for all levels (i.e., all switches s_i), until either $w = \text{t}(s_i)$ for some i, or we reach the end of the switch list: then the computed value of p'_{k+1} corresponds to the index of w in $\text{STS}(v)$. This index is then sufficient to distinguish between $\text{x}(v)$ and $\text{t}(v)$. Finally, we update the switch list as follows: if $w = \text{t}(s_i)$ for some i, then s_i (and all subsequent nodes in the switch list) is no longer a switch to w, and $\sigma(u, w) = [s_1, \ldots s_{i-1}]$. Conversely, if $w = \text{x}(v)$, then v becomes an additional switch in the list, and $\sigma(u, w) = \sigma(u, v) + [v]$. Finally, if $w = \text{t}(v)$, then the switch list remains unchanged.

Remark 3. With small modifications, the above algorithm can be adapted to automatically skip subsets of the form $\overrightarrow{\mathcal{R}}(v)$ for any arbitrary node v (use a reachability query for each generated node, and truncate the pile whenever a node in $\overrightarrow{\mathcal{R}}(v)$ should have been output). We can moreover evaluate the number of skipped nodes, which allows to compute the rank and the leaps of any new merge node u in roughly $|\text{excl}(u)|$ steps.

B.2 Proof of Theorem 2

In order to bound the depth of our dichotomy trees, we first need to make several observations on STS and STR. The overall strategy is to prove that the pair $(\Sigma(S), \Pi(S))$, defined below, decreases lexicographically between a range and its subrange when split.

Definition 10. *For a range S with head u, we say that S has* power value $\Pi(S) = \max(\pi(v) \mid v \in \text{tail}(u) \cap S)$, *and* switch value $\Sigma(S) = \max(d_S(u, v) \mid v \in S)$.

Proposition 5. *Consider a range S with head u and $v \in S \cap \text{tail}(u)$, $v \neq u$. Then sets $S_1 = S \setminus \overrightarrow{\mathcal{R}}(v)$ and $S_2 = S \cap \overrightarrow{\mathcal{R}}(v)$ are both ranges with heads u and v*

respectively, and $\mathrm{STS}(u)[|S_1|] = v$*. Moreover,* $\Pi(S_i) \leq \Pi(S)$ *and* $\Sigma(S_i) \leq \Sigma(S)$ *for each* $i \in \{1, 2\}$*.*

Proof. Let k such that $S = \mathrm{STS}(u, k)$, and p such that $\mathrm{STS}(u)[p] = v$. Since $v \in S$, we have $p < k$. With Proposition 3, $\mathrm{STS}(v)$ is the suffix of $\mathrm{STS}(u)$ starting at p, so $\mathrm{STS}(u)[q]$ is in $\overrightarrow{\mathcal{R}}(v)$ iff $q \geq p$. Thus, $S_1 = \mathrm{STR}(u, p)$ and $S_2 = \mathrm{STR}(v, k - p)$.

Moreover, $\Pi(S_i) \leq \Pi(S)$ since $\mathrm{tail}(v)$ is a subsequence (actually, a suffix) of $\mathrm{tail}(u)$. For the switch value, this is clear for S_1 (since only a subset is considered in the max). For S_2, we need to first note that, for $w \in S_2$, $d_S(u, w) = d_S(v, w)$. Indeed, since $w \in \overrightarrow{\mathcal{R}}(v)$, by the recursive definition of σ) we have $\sigma(u, w) = \sigma(v, w)$, hence the first switch (and thus, all subsequent switches) is on the same node for both paths.

□

Proposition 6. *The canonical range with head* u *is exactly* $\mathcal{C}(u)$*, and, given a range* S*, the following are equivalent:*

- *range* S *is short*
- $S \subseteq \mathcal{C}(u)$
- $\mathrm{anc}(u) \notin S$.

Proof. This is a direct consequence of Proposition 5 with $v = \mathrm{anc}(u)$. Indeed, $\mathcal{C}(u) = \overrightarrow{\mathcal{R}}(u) \setminus \overrightarrow{\mathcal{R}}(v)$ is a range with head u of size $|\mathcal{C}(u)|$. Also, $\mathrm{STS}(u)[|\mathcal{C}(u)|] = \mathrm{anc}(u)$ so $\mathrm{STR}(u, k)$ contains $\mathrm{anc}(u)$ iff $k > |\mathcal{C}(u)|$. □

Proposition 7. *For a long range* S *with head* u*,* $\mathrm{split}_1(S)$ *is a set of ranges* $\{S_1, \ldots, S_k\}$ *with heads respectively* $h_1, \ldots, h_k$ *such that*

1. $k \geq 2$ *and sets* S_i *form a partition of* S*.*
2. *Each* S_i *is a short range*
3. $\Pi(S_i) \leq \Pi(S)$ *and* $\Sigma(S_i) \leq \Sigma(S)$
4. $k \leq \log_2(\mathrm{rank}(u)) + 1$

Proof. Sets S_i are ranges by Proposition 5, since they are obtained by splitting S with sets of the form $\overrightarrow{\mathcal{R}}(v)$ with $v \in \mathrm{tail}(u)$ and they are short by Proposition 6. We have $k \geq 2$ since S is long. Proposition 5 also implies that $\Pi(S_i) \leq \Pi(S)$, and $\Sigma(S_i) \leq \Sigma(S)$.

Finally, with Proposition 1, the power of successive heads increases strictly at each step ($\pi(h_i) < \pi(h_{i+1})$). Starting at 0 in the worst case, since the maximum power in $\overrightarrow{\mathcal{R}}(u)$ is at most $\log_2(\mathrm{rank}(u))$, k is bounded by $\log_2(\mathrm{rank}(u)) + 1$.

□

Proposition 8. *For a non-atomic range* S *with head* u*, let* $\mathrm{split}_2(S) = \{\{u\}, X_1, \ldots, X_k, T\}$

- $\{u\}$ *and each* X_i *are ranges.* T *is a range if it is not empty.*
- $\{\{u\}, X_1, \ldots, X_k, T\}$ *is a partition of* S *with at least 2 non-empty sets.*
- *For each* $i \leq k$ $\Sigma(X_i) < \Sigma(S)$

– *If T is not empty, then $\Pi(T) \leq \Pi(S)$. The inequality is strict if S is short.*
– *If T is not empty, then $\Sigma(T) \leq \Sigma(S)$*

Proof. Set $\{u\}$ is the atomic range $\mathrm{STR}(u, 1)$. Sets X_i are ranges by the definition. If T is not empty, then $\mathrm{t}(u) \in S$, and T is a range by Proposition 5.

A first partition of S is $\{u\} \cup (\mathrm{set}(\mathrm{excl}(u)) \cap S) \cup (\overrightarrow{\mathcal{R}}(\mathrm{t}(u)) \cap S)$. The partition proposed here is a refinement where $\mathrm{set}(\mathrm{excl}(u)) \cap S$ is further partitioned into $X_1, \ldots, X_k$. It cannot contain a single non-empty set, since this set would then be $\{u\}$: this is not possible since $|S| > 1$.

Consider X_i with head h for some $i \leq k$. In order to show $\Sigma(X_i) < \Sigma(S)$, we aim at proving that $\sigma(u, h)$ is a prefix of $\sigma(u, v)$ for any $v \in X_i$. Let s be any swtich of $\sigma(u, h)$. First note that for any $v \in X_i$, we have $v \neq \mathrm{t}(s)$. Indeed, otherwise h would be in the exclusive part of s with $\mathrm{t}(s) \in \overrightarrow{\mathcal{R}}(h)$, hence $\mathrm{rank}(\mathrm{x}(s)) \geq \mathrm{rank}(h) > \mathrm{rank}(\mathrm{t}(s))$: a contradiction. Thus, by Proposition 4, X_i may not intersect $\overrightarrow{\mathcal{R}}(\mathrm{t}(s))$ (since otherwise it would also contain $\mathrm{t}(s)$), so $v \in \mathrm{excl}(s)$, and s is also a switch in $\sigma(u, v)$. More precisely, $\sigma(u, v) = \sigma(u, h) + \sigma(h, v)$ and $d_S(u, v) = d_S(u, h) + d_S(h, v)$. Since $h \in \mathrm{excl}(u)$, $\sigma(u, h)$ contains at least u and $d_S(u, h) \geq 1$. Thus, $d_S(h, v) < d_S(u, v)$ and overall, $\Sigma(X_i) < \Sigma(S)$.

Assume now that T is not empty. We have $\Pi(T) \leq \Pi(S)$ and $\Sigma(T) \leq \Sigma(S)$ by Proposition 5. Furthermore, if S is short, then S does not contain $\mathrm{anc}(u)$, so all nodes in $(\mathrm{tail}(u) \setminus \{u\}) \cap S$ have power at most $\pi(u) - 1$ (since otherwise they would be anchors for u). Thus, $\Pi(T) \leq \pi(u) - 1 < \pi(u) = \Pi(S)$. □

Lemma 2. *For a range S with switch value $\Sigma(S)$ and head rank r, the tree T_S has height at most $2\Sigma(S)\log(r) + 3$.*

Proof. Consider the list of ranges $[S_1, \ldots S_k]$ in a maximum-length dichotomy search from S to some node v, and write h_i for the head of range S_i (with $h_k = v$). With propositions 7 and 8, we have the following properties:

– $\Sigma(S_{i+1}) \leq \Sigma(S_i)$
– If S_i is long, then S_{i+1} is short and $\Pi(S_{i+1}) \leq \Pi(S_i)$
– If S_i is long, then $\Pi(S_{i+1}) < \Pi(S_i)$ or $\Sigma(S_{i+1}) < \Sigma(S_i)$

Combining two consecutive steps, if S_i is short, then some S' among S_{i+1}, S_{i+2} is also short, and either $\Sigma(S') < \Sigma(S')$; either $\Sigma(S') = \Sigma(S_i)$ and $\Pi(S') < \Pi(S_i)$. Overall the pair $(\Sigma(S_i), \Pi(S_i))$ decreases strictly for the lexicographical order over one or two splits. Moreover $\Pi(S_i)$ is upper bounded by $\max(\pi(v) \mid v \in S) \leq \lfloor \log_2(r) \rfloor$. Counting up to one extra step to reach the first short range, and up to two more steps to reach the last atomic range $S_k = \mathrm{STR}(v, 1)$, we obtain a total of $2\Sigma(S)\log(r) + 3$ in the worst case. □

*Proof (**Proof of Theorem 2**).* This is a direct application of Lemmas 1 and 2, using $\mathrm{rank}(u) \leq n$. □

C Material for Sec. 4, Algorithms for Mercurial

Lemma 3. *A revision graph of size n with no merge node has worst-case degree $\leq \max(\log(n), 2)$ and worst-case depth $\leq 2\log(n) + 2$.*

Proof. We use n as an upper bound for $\mathrm{rank}(u)$ for any u. Function split_1 has degree at most $\log(n)$, and split_2 has degree 2 for a non-merge node. For the depth, note that a dichotomy search never visits 2 long ranges consecutively, and that the value $\Pi(S)$ decreases strictly (by Propositions 7 and 8) between successive short ranges. Since $0 \leq \Pi(S) \leq \log(|V|)$, we get a depth of $2\log(n)+2$ overall. □

*Proof (**Proof of Theorem 3**).* First note that the required index for the oracle are rank and minrank, and split further requires tail and exclusive parents and anchors of all nodes, as well as leap data: such an index is indeed coherent. In the case of merge-free graphs there are no leaps, so the index is linear as well.

We show that the algorithm always keeps a single candidate range after a split+pruning step. This is clear for Rule 2: since all exclusive parts are empty, $\mathrm{split}_2(S) = \{\{u\}, S \setminus \{u\}\}$. Since $\{u\}$ is atomic it never yields *maybe* so it is never split any further. Thus, only $S \setminus \{u\}$ remains after applying this Rule.

For Rule 1, we use the following property of merge-free graphs: $\mathrm{minrank}(u) > \mathrm{rank}(\mathrm{anc}(u))$. Indeed, all nodes in $\mathcal{C}(u)$ are also in $\mathrm{tail}(u)$, so they are comparable with $\mathrm{anc}(u)$. Since they are not descendants of $\mathrm{anc}(u)$, then they must have a strictly greater rank. It follows that if the pruning oracle returns `maybe` for some range S_i in $\mathrm{split}(S)$, then it returns `no` for each S_j with $\mathrm{num}(j) < \mathrm{num}(i)$ (by criterion 3) or $\mathrm{num}(j) > \mathrm{num}(i)$ (by criterion 2).

The running time thus follows from the degree and depth of the search-tree: since both are in $O(\log n)$ by Lemma 3, we get a $O((\log n)^2)$ running time with an index containing the rank, minrank and leaps for each node. □

*Proof (**Proof of Theorem 4**).* For each node $v \in \Delta$, the hash of all ranges in the dichotomy search for v from $\overrightarrow{\mathcal{R}}(h)$ are exchanged into successive rounds, yielding the upper bound of H round-trips. Furthermore, all siblings of these ranges in the search tree are also exchanged, yielding the upper bound of DH exchanged values per node with distinct labels.

D Material for Sec. 5, Experimental evaluation

D.1 Graph and STS statistics

We compute statistics on two sets of repositories (cf Table 6). The first (denoted DB_1) is an extraction of 18350 repositories from the Bitbucket public Mercurial archive[oct20], having at least 16 nodes, and with forks (smaller "subgraphs") excluded, for a total of 6976003 nodes. The second (denoted DB_2) is a set of 11 hand-picked repositories from large open-source projects (Mozilla, Netbeans, PyPy, Mercurial, Evolve, OpenJDK, NetBSD), for a total of 7374214 nodes. For

Value	Normalization	DB$_1$ avg	DB$_1$ 5%	DB$_1$ 95%	DB$_2$ avg	DB$_2$ 5%	DB$_2$ 95%
Graph statistics							
Graph size n		380	17	650	6.7E5	6.9E3	2.8E6
Number of merges	$/n$	0.03	0.00	0.14	0.10	0.00	0.30
Reachability ratio		0.96	0.78	1.00	0.82	0.43	1.00
Graph width	$/n$	0.05	0.01	0.12	0.02	0.00	0.10
STS statistics							
Number of leaps per merge		0.02	0.00	0.10	0.11	0.01	0.29
Cost of STS over excl(u)	$/\lvert\text{excl}(u)\rvert$	1.17	1.00	1.85	1.09	1.00	1.50
Common tail misses	$/\lvert X_{uv}\rvert$	6.7E-3	0.00	0.00	1.8E-2	0.00	5.2E-2
Breakpoints	$/\lvert X_{uv}\rvert$	4.3E-3	0.00	0.00	3.5E-4	0.00	1.5E-3
STR and dichotomy statistics							
Ranges of interest	$/n$	1.51	1.25	1.75	1.87	1.74	2.01
$\lvert\text{split}_1(S)\rvert$ for long S	$/\log_2(\lvert S\rvert)$	0.68	0.33	1.26	0.68	0.33	1.26
$\lvert\text{split}_2(S)\rvert$ for short S	$/\log_2(\lvert S\rvert)$	1.36	0.40	2.00	1.32	0.43	2.00
Height of search tree of $\overrightarrow{\mathcal{R}}(u)$	$/\text{lr}(u)$	1.31	0.13	3.25	2.20	0.10	5.64
Worst-cost dichotomy search	$/\text{lr}(u)^2$	0.75	0.75	0.75	1.34	0.73	3.93
Reachability and label discovery							
STR reachability: oracle calls	$/\log_2(n)$	1.58	1.15	2.18	54.55	2.27	260
STR index size	$/n$	2.00	2.00	2.01	2.02	2.00	2.07
LD round-trips (1 edit)	$/\log_2(n)$	1.26	1.09	1.47	1.09	1.01	1.15
LD exchanged values (1 edit)	$/\log_2(n)$	3.22	2.65	4.13	12.5	4.61	71.34
LD round-trips (100 edits)	$/\log_2(n)$	1.92	1.76	2.10	1.84	1.62	2.22
LD exchanged values (100 edits)	$/\log_2(n)$	53.6	27.1	116	321	135	864

Fig. 6. Statistics on graphs of both databases (Bitbucket-Mercurial and hand-picked), with average, 5th and 95th percentiles. We write $\text{lr}(u)$ for $\log_2(\text{rank}(u)) + 1$. Other notations are detailed in the corresponding paragraphs. See also appendix D.2 for detailed results for DB$_2$.

each graph, we compute the number of merges, the reachability ratio (number of arcs in the transitive closure $/\binom{n}{2}$), and the width (longest antichain, obtained by a greedy algorithm). See Figure 6 for the resulting values.

STS Statistics. The complexity of STS is measured using the quantity $1 + d_p + \ell_p$ defined in Theorem 1: this quantity is computed over all positions p corresponding to a point in the exclusive part of all merges of the graph. The main goal of STS, given any two nodes u and v, writing $X_{uv} = \overrightarrow{\mathcal{R}}(u) \cap \overrightarrow{\mathcal{R}}(v)$ for the nodes reachable both from u and v, is to have a common suffix (*common tail*) between STS(u) and STS(v) containing almost all nodes of X_{uv}. The *common tail misses* are nodes of X_{uv} not in the common tail, and *breakpoints* are nodes of X_{uv} that are followed by different nodes in each STS. Values are averages for 100 random pairs.

STR and dichotomy statistics. We build the search-trees from all ranges of the form $\overrightarrow{\mathcal{R}}(u)$. All non-atomic ranges appearing in any such tree are called *range of interest*, we compute the average degree of split over all ranges S of interest. Finally we evaluate the cost of a dichotomy search as the sum of the

degrees of the ranges along the path. The largest possible cost is retained over the whole graph.

Reachability and label discovery. We pick random pairs of nodes (u, v) to perform reachability queries, and evaluate our algorithm on these tests (we retain the average for each graph over 100 runs). More precisely, we evaluate the number of oracle calls, and the index size as a number of stored integers. For label discovery, we simulate our label discovery protocol with a single random label difference: we count the number of round-trips and the total number of exchanged values (and retain the average over 100 runs). We also evaluate the number of exchanged values with 100 random editions, with 10554 repositories excluded due to not having enough candidates.

Results. Revision graphs are particular in that they are mostly linear, with high reachability ratio and relatively low graph width (50% of our dataset graphs contain a single path). The first goal of Stable-Tail Sorts is to have a *stable* ordering of reachable nodes: this is attained since, for any two nodes, the respective STSs have a large common suffix (representing typically more than 99% of the nodes reachable from both). The index needed to retrieve efficiently the STS (leap and rank tables) takes in practice less than 2 integers per node on average, which is largely within acceptable bounds.

Regarding our dichotomy algorithm, on average, any range S is split in less than $1.4 \log_2(|S|)$ ranges, and we reach any singleton after up to $1.8 \log_2(|S|)$ splitting steps. This gives the desired framework in which we can run sublinear dichotomy searches.

D.2 Detailed STS statistics for hand-picked large graphs

Value	Normalization	evolve	mercurial	mozilla-central	mozilla-try	mozilla-unified	netbeans	netbsd-pkgsrc	netbsd-src	netbsd-xsrc	openjdk	pypy
Graph statistics												
Graph size n		6k	50k	463k	5M	624k	316k	388k	364k	7k	54k	106k
Number of merges	$/n$	0.14	0.05	0.05	0.18	0.04	0.33	0.00	0.00	0.00	0.26	0.09
Reachability ratio		0.92	1.00	0.97	0.30	0.90	0.97	0.91	0.80	0.57	0.99	0.71
Graph width	$/n$	0.01	0.00	0.00	0.18	0.00	0.00	0.00	0.00	0.01	0.01	0.03
STS statistics												
Number of leaps per merge		0.27	0.04	0.10	0.02	0.10	0.11	NA	0.00	NA	0.07	0.30
Cost of STS over $\mathrm{excl}(u)$	$/\lvert\mathrm{excl}(u)\rvert$	3.14	1.06	1.32	1.03	1.35	1.45	NA	1.00	NA	1.34	1.35
Common tail misses	$/\lvert X_{uv}\rvert$	0.06	0.00	0.05		0.04	0.01	0.00	0.00	0.00	0.01	0.04
Breakpoints	$/\lvert X_{uv}\rvert$	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00
STR and dichotomy statistics												
Ranges of interest	$/n$	1.91	1.80	1.84	1.99	1.82	2.03	1.75	1.75	1.74	2.00	1.93
$\lvert\mathrm{split}_1(S)\rvert$ for long S	$/\log_2(\lvert S\rvert)$	0.66	0.67	0.66	0.69	0.66	0.64	0.66	0.66	0.68	0.64	0.67
$\lvert\mathrm{split}_2(S)\rvert$ for short S	$/\log_2(\lvert S\rvert)$	1.24	1.37	1.33	1.31	1.34	1.27	1.39	1.39	1.39	1.28	1.30
Height of search tree of $\overrightarrow{\mathcal{R}}(u)$	$/\mathrm{lr}(u)$	1.17	1.17	2.28	2.34	2.34	2.26	1.18	1.18	1.21	1.33	1.32
Worst-cost dichotomy search	$/\mathrm{lr}(u)^2$	0.75	0.75	0.76	6.70	0.87	1.17	0.75	0.75	0.75	0.71	0.77
Reachability and label discovery												
STR reachability: oracle calls	$/\log_2(n)$	4.47	7.54	19.1	497	14.6	24.6	2.66	3.18	1.88	11.8	13.2
STR index size	$/n$	2.07	2.00	2.01	2.01	2.01	2.07	2.00	2.00	2.00	2.04	2.05
LD round-trips (1 edit)	$/\log_2(n)$	1.14	1.09	1.06	1.00	1.11	1.11	1.06	1.09	1.13	1.09	1.09
LD exchanged values (1 edit)	$/\log_2(n)$	4.55	5.77	6.57	74.5	8.12	6.80	6.14	6.39	4.89	5.57	8.68
LD round-trips (100 edits)	$/\log_2(n)$	1.91	1.66	1.80	1.67	2.22	2.06	1.62	1.66	1.79	1.85	2.03
LD exchanged values (100 edits)	$/\log_2(n)$	150	215	321	867	365	307	279	271	134	236	389

Table 1. Detailed statistics (see Table 6) for large hand-picked open-source repositories. Depending on the row, each value is either the actual value for the whole graph, an average over all nodes or pairs of nodes, or an average over a random sample. Blank values could not be computed before timing out.

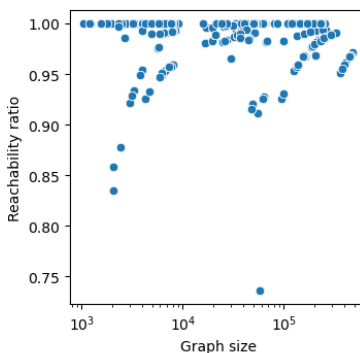


Fig. 7. Reachability ratio of subgraphs generated over DB_2.

D.3 Benchmark setting for reachability algorithms

Algorithms were implemented in Python using standard data structures (lists, sets, ...). [7]

The goal is to evaluate the time needed to insert a node, perform some reachability queries (both between recently inserted nodes and between uniformly random nodes), and the memory consumption of the index. We create any number of artificial revision graphs from the same underlying "true" repository (graphs in DB_2) by picking a random node x in the graph, and extracting the subgraph G_x induced by $\overrightarrow{\mathcal{R}}(x)$, along with a randomised anti-topological *insertion order* num (cf. Appendix D.5).

We then let each algorithm *process* each graph G_x: insert all nodes in the given insertion order and run the reachability queries required to compute the rank of each node as they are inserted (i.e., for merge nodes, we run a DFS from the exclusive neighbour and prune out nodes that are reachable from the tail neighbour). We thus measure the processing time for each algorithm and each instance G_x, as well as the memory needed to store the index : this is estimated as the number of integers stored in the index, independently of the underlying data structure (e.g. a set or a list of length n are both counted as n integers, independently of the actual number of bytes that Python is using to store them). Once the graph is processed, we further run reachability queries on randomly selected pairs of nodes in G_x (with typically much larger distance than during the processing phase), allowing to compute an average time per test. We split these random tests into *positive* and *negative* tests, since some algorithms

[7] A lower-level language with fine-tuned data-structures would certainly give better performances, but since this type of optimization is not within the scope of the paper, we elected to test all algorithms in this "uniform" setting. See Appendix D.4 for more algorithmic details

perform better than others on each kind, and the ratio of positive and negative tests for random pairs depends highly on the underlying repository.

D.4 Implemented algorithms

Beside our canonical-range-based reachability algorithms, we implemented several alternatives. We identified three families (naive, BFL, 2Hop), and tested several variants in each family. The best in each case was selected for the final comparison presented in Section 5.

Naive algorithms The first obvious possibilities for reachability queries are index-free algorithms DFS and BFS. For a query from u to v, we start the algorithm from u, pruning the search tree whenever we reach a node u' with $\mathrm{num}(u') < \mathrm{num}(v)$. We further tested a rank-based pruning step (prune if $\mathrm{rank}(u') \leq \mathrm{rank}(v)$ and $u \neq v$). This additional step comes at the cost of an additional comparison per node, and help prune the search tree. However, we do not count the rank computation in the running time of the algorithm (we assume it is computed in any case in our setting).

We also implemented a *full-index* approach, where the whole transitive closure is stored explicitly in memory, allowing constant-time reachability queries. As can be expected, the memory requirement in this setting is quadratic.

Overall, the algorithm giving best performances in this family (see Figure 8) was DFS without the rank-based pruning step.

BFL In this algorithm we pick $k = 16$ *landmarks*, and register for each node u the set of landmarks appearing either in $\overrightarrow{\mathcal{R}}(u)$ or $\boldsymbol{\mathcal{R}}(u)$ (with $\boldsymbol{\mathcal{R}}(u) = \{v \mid u \in \overrightarrow{\mathcal{R}}(v)\}$). Both subsets are stored as bit-vectors on single integers, allowing constant-time bit-wise operations. For a reachability query from u to v, we first test if there is a landmark in $\overrightarrow{\mathcal{R}}(u) \cap \boldsymbol{\mathcal{R}}(v)$, in which case we can directly answer positively. Otherwise, we run a DFS from u, and for each $u' \in \overrightarrow{\mathcal{R}}(u)$, we have an additional pruning test: if there is a landmark ℓ in $\overrightarrow{\mathcal{R}}(u') \setminus \overrightarrow{\mathcal{R}}(v)$ or in $\boldsymbol{\mathcal{R}}(v) \setminus \boldsymbol{\mathcal{R}}(u')$, then $v \notin \overrightarrow{\mathcal{R}}(u')$ and the search tree can be pruned at u'.

This approach works best if landmarks are evenly distributed within the graph, however all bit vectors have to be recomputed whenever landmarks are displaced. So in practice, we mostly keep landmarks unchanged when we insert new nodes, and compute bit-vectors for reachable landmarks as the *OR* of the corresponding bit-vectors of its out-neighbours. We reset landmarks whenever the graph size doubles, which leads to constant insertion time on average. The full set of tests includes variants with more landmarks (32 or 64), or a larger *reset ratio* (4 instead of 2). As for DFS, we further tested a rank-based pruning step, again without clear advantage. See Figure 9.

2-Hop The algorithm we implemented for 2-Hop is based on the dynamic algorithm presented by Zhu et al. [ZLWX14]. The first step is to associate a *score* $s(u)$ to each node (more details on how the score is selected are given below).

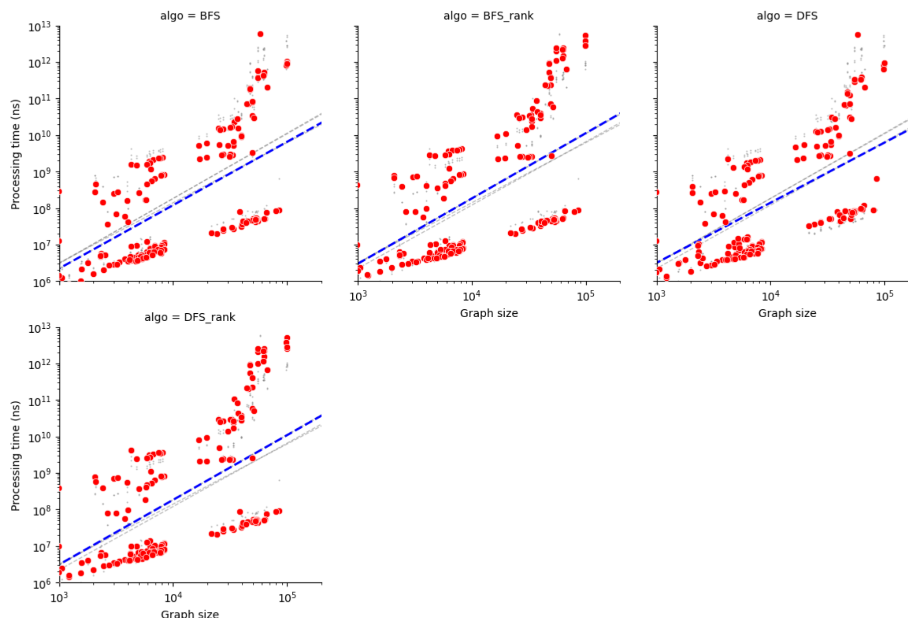


Fig. 8. Processing time as a function of the graph size for algorithms variants in the no-index family.

We maintain two lists of nodes $L_{\text{in}}(u)$, $L_{\text{out}}(u)$ for each node, where $v \in L_{\text{in}}(u)$ (resp. $v \in L_{\text{out}}(v)$) iff (i) $s(v) > s(u)$, (ii) $v \in \overrightarrow{\mathcal{R}}(u)$ (resp. $u \in \overrightarrow{\mathcal{R}}(v)$), and (iii) there is no w on any path between u and v with $s(w) > s(v)$. The key property of these lists is that there is a path from u to v iff the sets $(\{u\} \cup Lout(u))$ and $(\{v\} \cap Lin(v))$ intersect.

The insert operation is non-trivial. For a new node u, since u is a source in our setting we have $L_{\text{in}}(u) = \emptyset$. For $L_{\text{out}}(u)$, write N for the out-neighbours of u: we start with the candidate set $C = \bigcup_{u' \in N} L_{\text{out}}(u') \cup \{u'\}$. Clearly $L_{\text{out}}(u) \subseteq C$, but we need to filter out those nodes that do not satisfy conditions (i) and (iii). Note that for (iii), only nodes $w \in L_{\text{out}}(u)$ need to be tested. The main difficulty is then to update $L_{\text{in}}(v)$ for existing nodes v in the graph, since u may need to be inserted in many such lists. It can be noted that for any such v, we have $v \in \overrightarrow{\mathcal{R}}(u')$ for some $u' \in N$, so either $v \in C$, or there is $w \in C$ with $w \in L_{\text{in}}(v)$ and $s(w) < s(u)$. Thus, we maintain for each w the inverse list $I_{\text{in}}(w) = \{v \mid w \in L_{\text{in}}(v)\}$, and enumerate $\bigcup_{w \in C, s(w) < s(u)} I_{\text{in}}(w)$. All nodes in this set are candidate nodes v, and it remains to verify condition (iii) before inserting u to $L_{\text{in}}(v)$ (which can be done by looking-up nodes in $L_{\text{in}}(v) \cap L_{\text{out}}(u)$).

Here the lists L_{out} are coherent, but L_{in} and I_{in} may grow whenever a node is inserted, unless the score of new nodes is always minimal.

For optimal performances, the score should normally take into account both in-degree and out-degree of each node. On the other hand, editing the score of

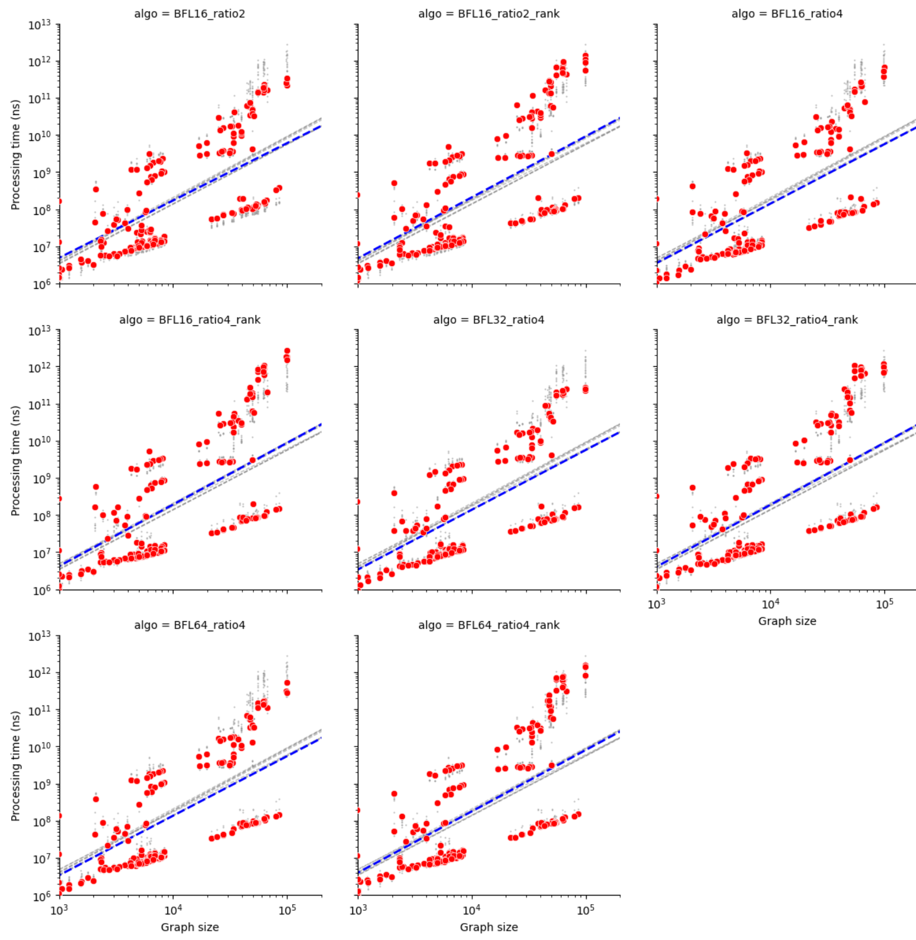


Fig. 9. Processing time as a function of the graph size for algorithms variants in the BFL family.

existing nodes may lead to arbitrarily large changes in the rest of the index. We thus tested different versions (see Figure 10):

- (V0) the score of inserted nodes is always minimal, and new scores are assigned periodically[8] (leading to a full reset of the index);
- (V1) the score of a node is random and constant;
- (V2) the score is based on its out-degree (which remains unchanged) plus a random component for tie-breaking.

Note that in V0 we do not need to store the inverse lists, since a node is never added to any L_{in} at insertion time (only during full resets).

As a side note, the insertion algorithm is different from the one presented in [ZLWX14], not only because adding sources entails a few simplification, but also because of a couple of oversights in the original pseudo-code (using the notations from the paper), cf Algorithm 2 p.1327:

- the main loop of the algorithm does not include the case $u = v$, so v cannot be added to any list L_{in} when $L'_{\text{in}}(v)$ is empty (which is the case in particular for sources). This can easily be fixed by looping over $L'_{\text{in}}(v) \cap \{v\}$.
- The inner loop considers the set $L'_{\text{out}}(v) \cup \{v\}$, but it should instead loop over the whole set C_{out} defined in Algorithm 1. A simple example is the case where the new node v has the largest score over the whole graph: in this case, $L'_{\text{out}}(v) = \emptyset$, so the second loop is empty, but instead v should be added to $L_{\text{in}}(x)$ for all $x \in \overrightarrow{\mathcal{R}}(v)$.

D.5 Creating random anti-topological orders

The order in which nodes are inserted in the graph (i.e. the num function) may have an important impact on the performance of algorithms. The *baseline* algorithm we implemented works as follows: pick a random source of the graph, place it last in the order, and continue in the remaining subgraph.

The main drawback with this approach is that two parallel branches can be overly interleaved. For instance, if the graph consists of two completely independent paths advancing in parallel, the probability to have a long interval of nodes from the same path being inserted consecutively decreases exponentially with the length of the interval. This can be realistic in some cases (e.g. for a server receiving code from several developers currently working in parallel on different sub-projects), but not always (e.g., each developer would normally not add the other branch into her own graph until both branches merge, leading to two well-separated intervals of nodes). So beside this baseline permutation, we also generate *unshuffled* permutations. To this end, we pick random intervals $I = [u_i, \ldots, u_j]$ in the baseline order, and reorder such intervals into $I \cap \boldsymbol{\mathcal{R}}(u_j) + I \setminus \boldsymbol{\mathcal{R}}(u_j)$ (i.e., all nodes that can reach u_j first, then the rest, keeping the same relative order within each part).

[8] we tested either with fixed *ratios* (reset scores when the graph size is multiplied by a given factor) or *deltas* (reset scores when a fixed number of nodes is added)

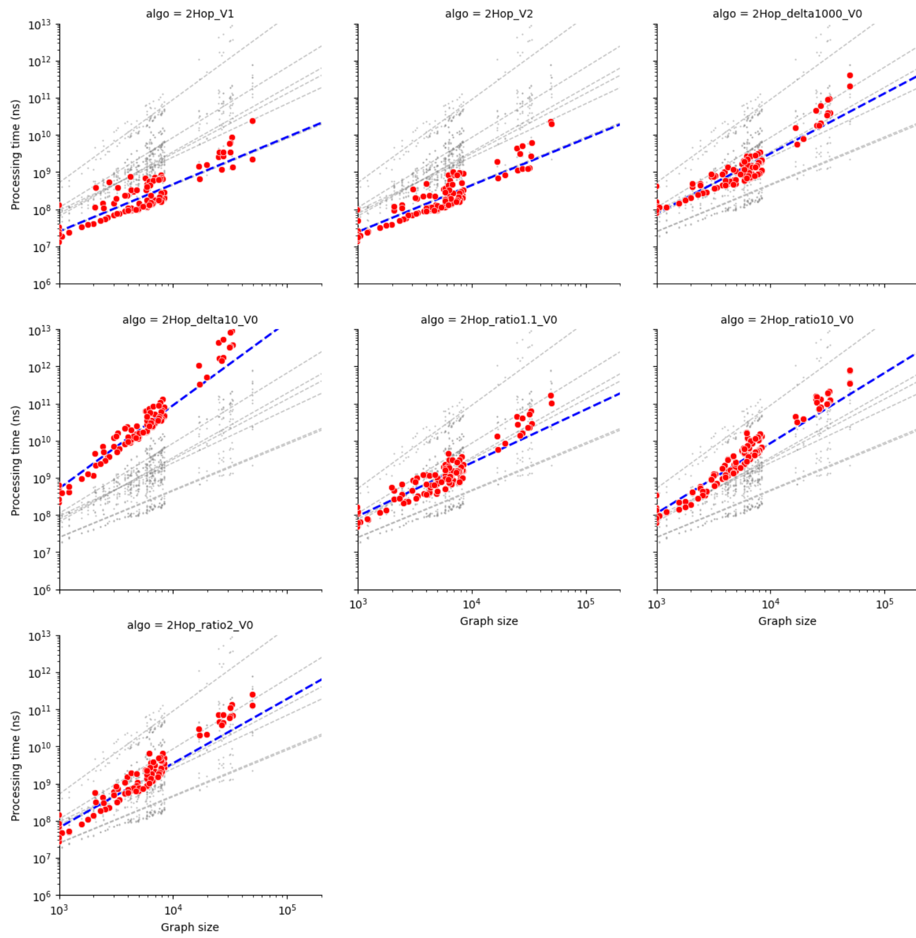


Fig. 10. Processing time as a function of the graph size for algorithms variants in the 2Hop family.

The final benchmark group is composed both of baseline permutations and of unshuffled permutations with different window sizes (10, 100, 1000).

We evaluate the window size influence over an algorithm as follows: for each subgraph in our dataset, we compare the processing times between all pairs of tested window sizes, compare the ratio, and average over all pairs of window sizes and all subgraphs. We notice that with larger unshuffling windows our range algorithm as well as 2Hop are slightly slower (up to 8% increase on average for 2Hop). On the other hand, DFS and BFL gain respectively 10% and 15% in their running time when we increase the window size.

Algorithm	Range	DFS	BFL	2Hop
$\frac{\text{Time with larger window}}{\text{Time with shorter window}}$	1.026	0.903	0.851	1.082

D.6 Detailed results for memory and individual queries

See Figures 11, 12 and 13 for detailed scatter-plots for memory usage, positive queries and negative queries respectively.

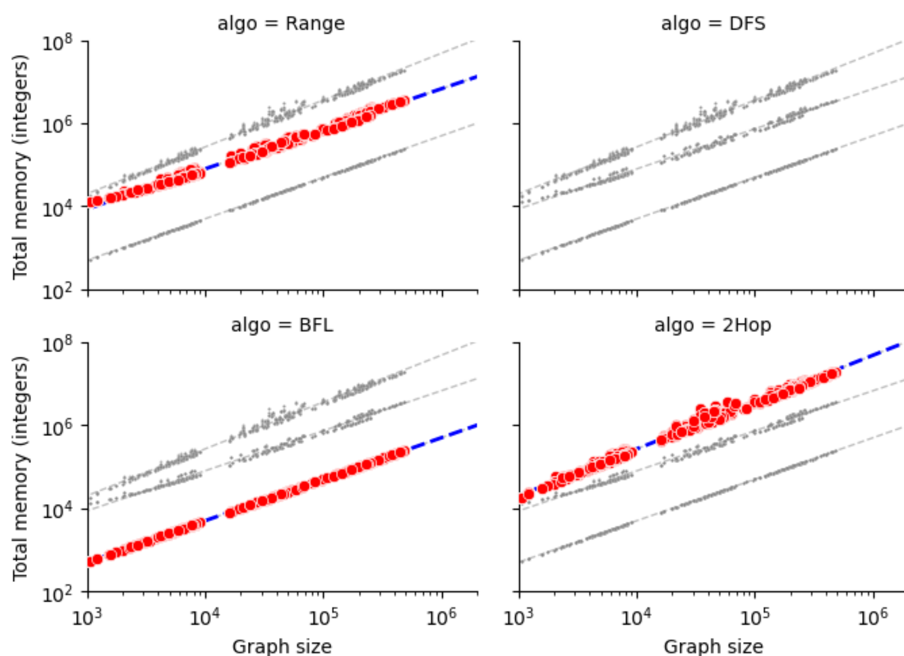


Fig. 11. Memory needed for each algorithm as a function of the instance size. Note that DFS does not need any index, so it is not depicted here. Beside the index, our algorithms use caches for storing split results between successive calls. These caches are included in the total depicted here, even though they can be cleared at any time.

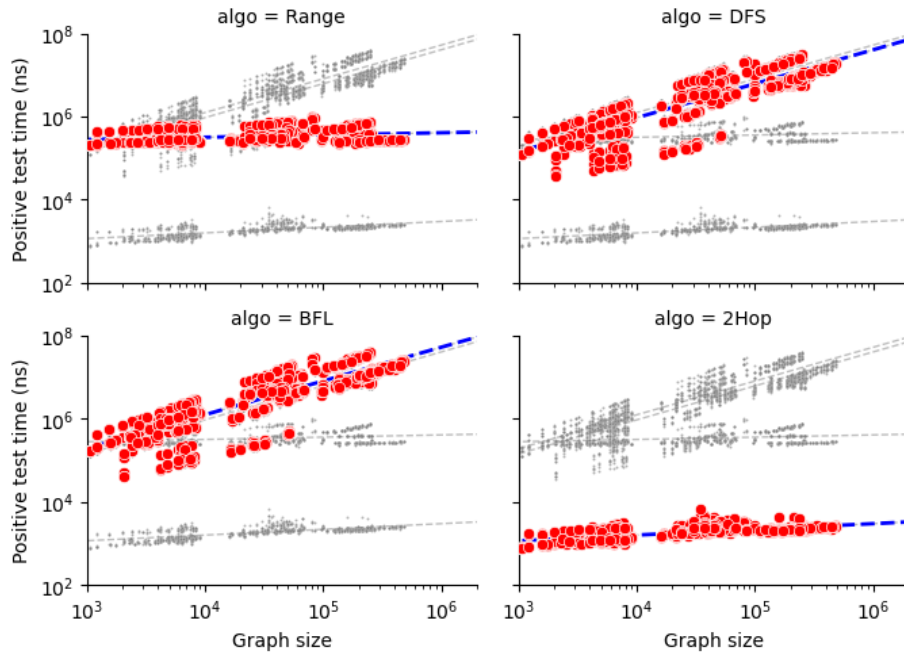


Fig. 12. Average time to perform a positive reachability query for a random pair of nodes in the graph.

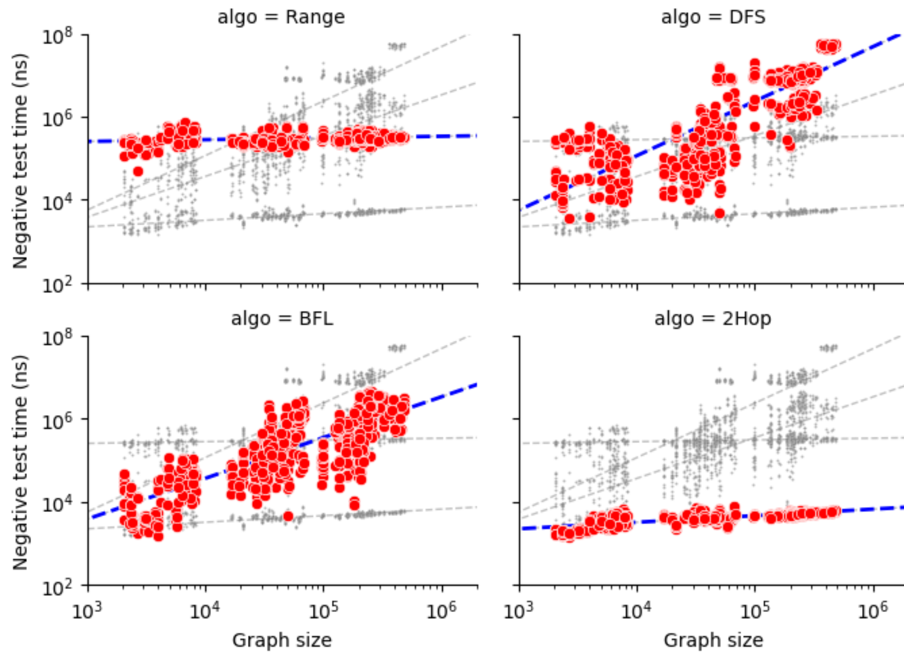


Fig. 13. Average time to perform a negative reachability query for a random pair of nodes in the graph.

Bibliography

[oct20] Bitbucket public mercurial archive. https://bitbucket-archive.softwareheritage.org/, 2020.

[ZLWX14] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. Reachability queries on large dynamic graphs: a total order approach. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1323–1334, 2014.