Incremental Reachability Index

Laurent Bulteau
Université Gustave Eiffel, CNRS, LIGM, F-77454 Marne-la-Vallée, France

Pierre-Yves David
Octobus, France

Florian Horn
Université Paris Cité, CNRS, IRIF, F-75013 Paris, France

Euxane Tran-Girard
Université Gustave Eiffel, CNRS, LIGM, F-77454 Marne-la-Vallée, France

Abstract

We study the reachability problem in append-only DAGs: given two nodes u and v, is there a path from u to v? While the problem is linear in general, it can be answered faster by using a precomputed index, which gives a compressed representation of the transitive closure of the graph.

Index algorithms are evaluated on three dimensions: the *query time* that the algorithm needs to answer whether there is a path from one node to another, the *memory* that the index uses per node, and the *indexing time* that is required to update the index when a node is added to the graph.

In this paper, we combine Jagadish's static index [9] with Felsner's online chain-decomposition algorithm [7] to create an *incremental* index: data associated with a node is immutable, guaranteeing that queries are answered properly even if new nodes are inserted while the query is processed.

Its query time is constant, but its index size is heavily dependent on the graph width, and as such is not competitive with recent indexing algorithms (2-hop, tree-chain, ...). We also propose a version of that incremental algorithm with a much lighter index. In the most compressed version, the query time becomes $O(\log n)$. However, constant-time queries can be retained depending on the desired time/memory trade-off.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Directed acyclic graphs, reachability, append-only, index

Digital Object Identifier 10.4230/LIPIcs.SEA.2025.9

Related Version *Previous Version*: `https://hal.science/hal-04929088`

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.14792894`

1 Introduction

The reachability problem in Directed Acyclic Graphs consists in deciding, given two nodes u and v, whether there exists a path from u to v. It can be solved in linear time with a depth-first search, visiting either all the ancestors of v or all the descendants of u before one can answer negatively. A common way to lower that complexity is to use a precomputed index that allows future queries to be answered much faster. Several solutions have been proposed for this problem relying on tree cover with interval labeling [1, 15], approximate transitive closure [14, 12], and 2-hop [5, 16, 11]. The relative quality of such algorithms is usually evaluated on three dimensions: the *query time* is the amount of time it takes to process a single reachability query; the *memory* is the amount of information stored in the index; the *indexing time* is the amount of time it takes to compute the index of a new node.

In this paper, we consider append-only DAGs, which can only grow by adding children, *i.e.* in topological order. This constraint on graph updates is implied for Merkle Graphs, in which each node holds immutable content; it is identified by a hash of its contents, including its parent identifiers. We propagate this constraint to the index by considering *incremental*

algorithms, where the local index of the nodes do not change when new nodes are added. Formally, the index should be an append-only array, i.e. such that new entry insertions do not affect older entries. A small mutable *manifest* is acceptable, containing additional data (e.g., graph size, ...). An index satisfying such constraints allows for a number of useful features. **Incremental computation:** the index can be accessed by multiple *readers*, concurrently with a *writer* inserting new nodes. In an incremental index the writer can freely compute index entries for new nodes and append them to the end of the index array, then she only needs to lock the manifest while it is updated to its new value. In particular if the manifest is not necessary to answer queries (only for index updates, as is the case in our algorithm), then no lock at all is necessary. This allows the index to be partially computed and asynchronously updated. **Cache management:** the index is stored on disk and one or several clients may request values of $I[u]$ and keep responses in cache; with an incremental index these caches remain valid even after inserting a node. Cache fragments can be shared through a content-addressable storage. **Transactional updates:** in the case of a rollback, i.e. if the last k inserted nodes need to be removed from the graph, only the manifest needs to be reverted to the previous point in time. The index, and cached entries or checksums if any, remain automatically valid for all remaining nodes.

An existing algorithm by Jagadish [9] offers a fast index, with constant-time queries, based on a chain decomposition of the graph. Its chain decomposition is only computed when the graph is complete, but it can be adapted to use the *online* decomposition by Felsner [7] to become incremental as long as nodes are inserted in topological order. The index size for this algorithm is, however, heavily dependent on the graph width, i.e., the largest set of pairwise unreachable nodes. In this form, it is not competitive with the many recent algorithms. We are not aware of any other incremental reachability index (recent improvements [4, 10] on [9] optimize the chain decomposition once the graph is complete, so they are not incremental).

Formalism for DAGs and Chains.

▶ **Definition 1** (*Directed acyclic graphs*). *A* directed acyclic graph (DAG) *is a directed graph $G = (V, E)$ without cycles. If there is an arc $u \to v$ in E, we say that u is a parent of v and v is a child of u. If there is a path $u \to^* v$ (possibly with $u = v$), we say that u is an ancestor of v, v is a descendant of u, and that u, v are comparable. A node without parents is a source and a node without children is a sink.*

In the scope of this paper, we assume that the nodes of G are inserted one by one, in a topological order, that is, all nodes are sinks when they are inserted. Thus, each node is identified with its insertion rank (i.e. $V = \{1, \dots |V|\}$). The topological order further implies that $u < v$ for any arc $u \to v$ and that $u \leq v$ for any pair of ancestor/descendant $u \to^* v$.

▶ **Definition 2** (*Chains* and *anti-chains*). *A* chain *of G is a set of pairwise-comparable nodes. An* anti-chain *is a set of pairwise-non-comparable nodes. A chain decomposition of G is a partition of V into chains. The width of G, denoted $w(G)$, is the size of its largest anti-chain.*

▶ **Theorem 3** (Dilworth [6]). *The minimum number of chains in a chain decomposition of G is $w(G)$.*

The optimal chain decomposition of a graph can be computed in polynomial time [8]. However, in the incremental setting we consider here, a chain decomposition must be extended with each new node without editing previous chain assignments. This corresponds to the *on-line chain decomposition* problem described by Felsner et al. [7, 3]: in this setting, any algorithm yields a chain decomposition of size $\Omega(w(G)^2)$ in the worst case. Felsner also gives

an algorithm producing a chain decomposition of size $W \leq \frac{1}{2}w(G)(w(G)+1) = O(w(G)^2)$; see [3, Theorem 3.5] for a very concise description of this algorithm. The insertion time is dominated by the cost of W reachability queries performed in G (all using the inserted node). Beside the chain decomposition itself, this algorithm maintains a size-$O(W)$ manifest containing the last inserted vertices in each chain (and is only required for node insertions). Note that in our experiments (for graphs generated with our algorithm), we observe a linear dependency between W and $w(G)$, see Figure 6 in appendix.

Our contribution. Used with the incremental chain decomposition by Felsner [7], the chain-based reachability index by Jagadish [9] offers an incremental index with $O(1)$ query time and $O(W)$ memory per node. We propose an improved version of this algorithm based on two *anchor* strategies, yielding a much lighter index while retaining incrementality. The query time becomes $O(\log n)$ in the most-compressed version and can be reduced to $O(1)$ depending on the desired time/memory trade-off. The theoretical memory upper bound remains $O(W)$ in the worst case.

We run experimental evaluations of our algorithm both on generated and real-life graphs. Compared to Jagadish [9], we obtain similar query times and reduce the index size by a factor close to 10. With these encouraging results, we compare our algorithm against a selection of non-incremental reachability index. We obtain in particular similar query time as our implementation of 2-hop, with competitive index size in real-life graphs and generated graphs of width up to $1.5\sqrt{n}$.

Outline of the paper. Section 2 presents the chain index obtained by combining Jagadish's index [9] with Felsner's online chain decomposition [7]; we then build upon this algorithm in Section 3 to present our anchor-based index. We then describe the set-up for the experimental evaluation in Section 4, and the conclusions in Section 5. Additional detailed results are given in appendix.

2 Simple Chain Algorithm [9, 7]

Throughout this algorithm, we maintain a chain decomposition with a list of chain identifiers, and we use the term *chain* both for the chain *identifier* and the actual *set of vertices*. We store, for each node u, [the identifier of] the chain containing u. We write `Chains` for the set of chain identifiers.

▶ **Definition 4.** *Given a vertex v and a chain C, the top vertex of a chain C under v is $t_v[C] = \max\{u \mid u \in C \text{ and } u \to^* v\}$. We write $t_v[C] = -1$ if no such u exists. Given a chain decomposition $\{C_1, \dots, C_w\}$ of G, the top set of vertex v is the set of pairs $\{(C_i, t_v[C_i]) \mid 1 \leq i \leq w\}$.*

The core observation for the algorithm is the following: for any $u \in C$, u is an ancestor of v if and only if $u \leq t_v[C]$ (recall that node identifiers form a topological order on G). Indeed, the forward direction is clear since we took the maximal ancestor of v within chain C. The converse direction is obtained by transitivity: $t_v[C]$ is an ancestor of v and, since u and $t_v[C]$ are in the same chain C, u and $t_v[C]$ are comparable and u is an ancestor of $t_v[C]$ since $u \leq t_v[C]$. The top vertex $t_v[C]$ is thus sufficient to answer reachability queries (u, v) in constant time for any vertex in $u \in C$.

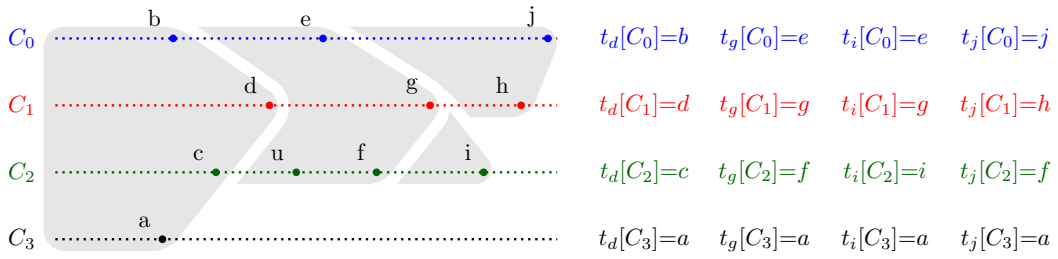

Figure 1 Left: A graph with a decomposition into 4 chains (C_0, C_1, C_2, C_3). Arcs are oriented left-to-right and are not drawn, only ancestor sets of a selection of nodes are depicted: the shaded area containing d represents all ancestors of d, the area containing g represents ancestors of g that are not ancestors of d, etc. Right: the top vertices for each chain and each $v \in$ {d,g,i,j}. For instance, we have u$\rightarrow^*$j since $u \in C_2$ and $u < f = t_j[C_2]$.

Description of the index. Since we rely on the chain decomposition by Felsner [7], we need to store the `Chains` list in the manifest. Furthermore, for each node v, we store:

- the identifier of the chain containing v in G;
- the set $\{(C, t_v[C]) \mid C \in \texttt{Chains} \text{ and } t_v[C] \neq -1\}$.

The manifest size is thus W, and each index entry has size $1 + 2|t_v| = O(W)$. The sets t_v can be stored either as sorted lists (for optimal storage) or as hashmaps (for optimal look-up). We pick the latter for the theoretical analysis, since the asymptotic memory requirement is identical. In practice, one can store the sorted lists on disk, use them directly for isolated reachability queries, or build a hashmap when queries are expected to be repeated for the same vertex.

Answering queries. In order to test whether u is an ancestor of v, we look up the index C of the chain containing u, and return *true* if $u \leq t_v[C]$. Using a hashmap for t_v, we get constant-time reachability queries.

Node insertion. Upon insertion of node v, we first compute the *combined top vertices* of the parents of v, defined as $ct_v[C] = \max\{t_p[C] \mid p \rightarrow v\}$. With this array we can answer reachability queries for v in constant time. Indeed, for a node u in chain C, u is an ancestor of v if $u = v$ or if $u \leq ct_v[C]$ since any ancestor of v (beside v itself) is an ancestor of some parent p of v. We can then assign a chain C^* to v in $O(W)$ with Felsner's algorithm [7], and compute t_v as follows: $t_v[C^*] = v$, and $t_v[C] = ct_v[C]$ for $C \neq C^*$.

The complexity for inserting v is thus dominated by the computation of ct_v in $O(W\delta(v))$. Note that for high-degree graphs ($\delta(v) > W$), this complexity can be reduced to $O(W^2 + \delta(v))$ by considering only the top-most parent p in each chain.

3 Our Anchor-based Algorithms

We can now present our anchor-based approach, aiming at reducing the index size to a minimum while keeping logarithmic queries. We present first the index itself, followed by two strategies for picking anchors, and finally some possible extensions for specific use-cases.

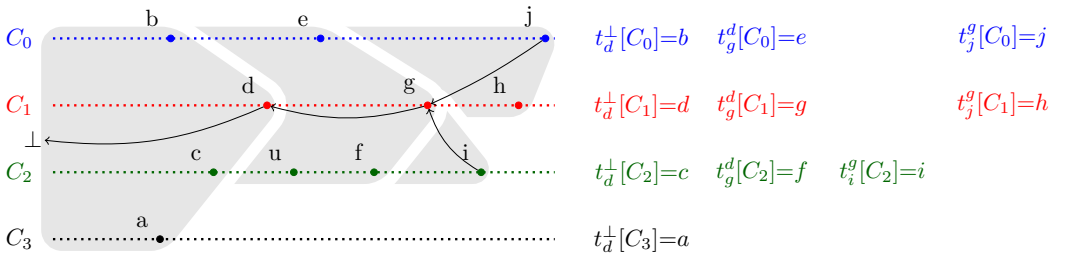

Figure 2 Left: The graph from Figure 1, with additional *anchor* information drawn as bended arcs (e.g. $\mathsf{anc}(g) = d$). The shaded areas are thus exactly the restricted ancestor sets of d, g, i and j. For instance, $\mathsf{AL}(j) = [j, g, d]$ and $\mathsf{ad}(j) = 3$. Right: the restricted top sets of the same vertices, which are subsets of the ones depicted in Figure 1, with much less redundancies. Here we have u$\rightarrow^*$j since $u \in C_2$ and $u < f = t_g^d[C_2]$ with $g = \mathsf{anc}(j)$.

3.1 Index construction

We introduce the notion of *anchor* of a node v, denoted $\mathsf{anc}(v)$. It can either be an ancestor of v other than v itself, or $\mathsf{anc}(v) = \perp$ when there is no suitable ancestor. As required by the incremental setting, the anchor is picked upon node insertion and cannot be changed afterwards. We discuss in the next section several anchor-picking strategies. We define the *Anchor-List* as follows: $\mathsf{AL}(\perp) = []$ and $\mathsf{AL}(v) = [v] + \mathsf{AL}(\mathsf{anc}(v))$ for any node v. The *anchor-depth* of v, denoted $\mathsf{ad}(v)$ is the length of $\mathsf{AL}(v)$, and $\mathsf{ad}(G) = \max(\mathsf{ad}(v) \mid v \in V)$. Then, the anchor is used to shrink the index, as we define the following *restricted top set* (see also Figure 2).

▶ **Definition 5.** *Given a vertex v, a chain C and an anchor a, the* restricted ancestor set *of v with anchor a is $\mathcal{R}_v^a = \{u \mid u \rightarrow^* v \text{ and } u \not\rightarrow^* a\}$ if $a \neq \perp$, $\mathcal{R}_v^a = \{u \mid u \rightarrow^* v\}$ otherwise. The restricted top vertex of chain C under v with anchor a is $t_v^a[C] = \max\{u \mid u \in C, u \in \mathcal{R}_v^a\}$ (-1 if no such u exists)*

Given a chain decomposition $\{C_1, \ldots, C_w\}$ of G, the restricted top set of vertex v with respect to anchor $a = \mathsf{anc}(v)$ is the set of pairs $t_v^a = \{(C_i, t_v^a[C_i]) \mid 1 \leq i \leq w, t_v^a[C_i] \neq -1\}$.

Description of the index. For each node v, we store the identifier of the chain containing v in G, the anchor $\mathsf{anc}(v)$, and the restricted top set $t_v^{\mathsf{anc}(v)}$.

Note that the worst-case index size per node remains $O(W)$. However, if the restricted ancestor set is small, then we also have a useful bound: $|t_v^{\mathsf{anc}(v)}| \leq |\mathcal{R}_v^{\mathsf{anc}(v)}|$.

Answering queries. In order to test whether u is an ancestor of v, we look-up the index C of the chain containing u and compute $t_v[C]$ using the following recurrence (as before, u is an ancestor of v if and only if $u < t_v[C]$):

$$t_v[C] = \begin{cases} t_v^{\mathsf{anc}(v)}[C] & \text{if } t_v^{\mathsf{anc}(v)}[C] \neq -1 \\ -1 & \text{if } t_v^{\mathsf{anc}(v)}[C] = -1 \text{ and } \mathsf{anc}(v) = \perp \\ t_{\mathsf{anc}(v)}[C] & \text{otherwise} \end{cases}$$

The recurrence takes up to $\mathsf{ad}(v)$ iterations since in the worst case the whole anchor list of v needs to be visited, so the running time becomes $O(\mathsf{ad}(v))$.

In case of successive queries with the same vertex v, values $t_v[C]$ can be cached using $O(W)$ temporary memory. Over N tests, only $\min(N, W)$ different values of $t_v[C]$ need to be computed, yielding an amortized time of $O(1 + \min(1, \frac{W}{N})\mathsf{ad}(v))$ per test, thus converging to $O(1)$. Such a cache can also speed up consecutive tests with different vertices, since it is common for different vertices to share the same anchor.

Node insertion. As in the simple chain algorithm, upon inserting v, we assign a chain to v, compute the top sets t_p of the parents of v and combine them into the top set t_v. The bottleneck here is the computation of the sets t_p, since each has size W and each top vertex takes time $O(\mathsf{ad}(p))$, so we need $O(W\delta(v)\mathsf{ad}(G))$ in total.

We then compute the anchor of v (using t_v for constant-time ancestor tests with respect to v if necessary). Finally, we have $t_v^{\mathsf{anc}(v)}[C] = -1$ if $t_v[C] \to^* \mathsf{anc}(v)$ and $t_v^{\mathsf{anc}(v)}[C] = t_v[C]$ otherwise. This step requires $N = W$ ancestor tests with $\mathsf{anc}(v)$, which are thus performed in time $O(W\mathsf{ad}(G))$.

3.2 Picking Anchors

An ideal anchor-picking strategy would satisfy the following criteria:

- memory optimization: minimize the number of different chains in the restricted ancestor set of v (so this set should be as small as possible)
- query time optimization: minimize the length of the anchor chain (so the restricted ancestor sets should be large enough)

We introduce two strategies, both of them aiming at logarithmic-size anchor chains with near-constant average canonical set size. We further introduce parameterizations of both strategies that help improve the query time for the first and index size for the second.

3.2.1 Record-based anchors

In this strategy, we assign a fixed *random* score to each node (chosen independently and uniformly at random in, say, $[0, 1]$). The overall invariant is that the score of the anchor should be higher than the score of the node. In simplified settings (see Lemma 7) below, this guarantees a $\log(n)$ bound on the anchor chain length with high probability. Within this constraint, we aim at minimizing the number of chains in $\mathcal{R}_v^a$.

▶ **Anchor Strategy 1** (With maximum depth $D \in \mathbb{N} \cup \{+\infty\}$)**.** *For any v, let $S = \bigcup_{p \to v}\{a \mid a \in \mathsf{AL}(p)$ and $\mathsf{score}(a) > \mathsf{score}(v)\}$. If S is not empty, let $a \in S$ minimizing $|t_v^a|$. Choose $\mathsf{anc}(v) = a$ if $\mathsf{ad}(a) < D$. In any other case, choose $\mathsf{anc}(v) = \bot$.*

This strategy is intended as a sub-linear approximation of the ideal strategy picking an anchor with higher score than v and minimal size of $|t_v^a|$. Intuitively, for most nodes, the anchor is selected among the parents and the restricted ancestor set is thus minimized (and the restricted top set has often size 1 or 2). For nodes with high scores, the restricted ancestor set is larger, but $\mathsf{anc}(u)$ is much closer to the sources of the DAG, and has itself a higher score, so the anchor list should have a bounded length.

▶ **Proposition 6.** *For any constant D, the query time with Strategy 1 is $O(1)$.*

Proof. By construction, $\mathsf{ad}(v) \leq D$ for all v, so the query time under this strategy is at most $O(D) = O(1)$. ◀

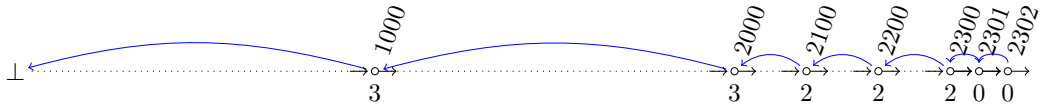

Figure 3 Example of Anchor Strategy 2 on a simple left-to-right path of length 2302. Anchors are depicted with bended blue arcs, and the base-10 power of each node is given below (e.g. node 2000 has power 3 since $2000 \geq 2.10^3 > 1999$).

▶ **Lemma 7.** *Consider Strategy 1 with $D = +\infty$. Let A be the set of ancestors of v. If all vertices in A have degree at most one, then* $\mathsf{ad}(v) \leq \log(|A|)$ *with high probability.*

Proof. Write $A = (a_1, a_2, \dots a_{|A|})$ with $a_1 = v$ and such that there exists a path $a_{|A|} \to \dots \to a_2 \to a_1$. Then by the anchor strategy, we have $a_i \in \mathsf{AL}(v)$ if and only if $\mathsf{score}(a_i) > \mathsf{score}(a_j)$ for all $j < i$. Since the scores are taken independently at random, each a_i has probability $1/i$ to have the maximum score over $\{a_1, \dots, a_i\}$. Thus, a_i is in $\mathsf{AL}(v)$ with probability $1/i$, hence the number of vertices in $\mathsf{AL}(v)$ is asymptotically $O(\log n)$. ◀

3.2.2 Power-based anchors

In this strategy, we do not use random scores, but instead keep track of the number of ancestors (hereafter called rank) of nodes as they are inserted. Whenever the rank passes a larger power of some integer B, the node is more likely to have a large restricted ancestor set and to be an anchor for future nodes; however in most cases the nodes have a restricted ancestor set of size $\leq B$.

▶ **Definition 8.** *Let v be a node. Write* $\mathrm{rank}(v)$ *for the number of ancestors of v, and* $\mathrm{lp}(v)$ *for the parent p of v maximizing* $\mathrm{rank}(p)$ *(ties are broken arbitrarily, we write* $\mathrm{lp}(v) = \bot$ *for source nodes; we further write* $\mathrm{rank}(\bot) = 0$*). Given an integer B, the* base-B *power of v, denoted $\pi_B(v)$ is the largest integer π such that there exists an integer x with* $\mathrm{rank}(v) \geq xB^\pi > \mathrm{rank}(\mathrm{lp}(v))$.

▶ **Anchor Strategy 2** (With base B)**.** *For any v, define* $\mathsf{anc}(v)$ *as the first node a of* $\mathsf{AL}(\mathrm{lp}(v))$ *such that $\pi_B(v) \leq \pi_B(a)$.*

Consider for example the path graph sketched in Figure 3. The rank of each node is depicted above each drawn node, so there are 2302 nodes (we use $\mathrm{rank}(v)$ as a vertex identifier as well, corresponding to an insertion order starting at 1). The anchor list of node 2302 is depicted: $\mathsf{AL}[2302] = (2301, 2300, 2200, 2100, 2000, 1000)$. In this graph, the maximum $\mathsf{ad}(v)$ is reached for $v = 1999$ ($\mathsf{ad}(v) = 19$). Note that most nodes (those that are not multiple of 10) have a trivial restricted ancestor set (containing the single node v), so the restricted set of chain tops is a singleton for them, even if the graph is somehow decomposed into multiple chains.

Under this strategy, we can prove a logarithmic (deterministic) upper bound on the anchor depth of each node, as well as an upper bound on the restricted ancestor set of all nodes with small power.

▶ **Lemma 9.** *For any v we have* $|\mathcal{R}_v^{\mathsf{anc}(v)}| \leq B^{\pi_B(v)+1}$ *and* $\mathsf{ad}(v) \leq B\lfloor \log(\mathrm{rank}(v)) \rfloor$.

Proof. Let $(v_0 = v, v_1, v_2, \dots, v_\ell = \bot)$ be the list of nodes defined with $v_{i+1} = \mathrm{lp}(v_i)$ until a source is reached.

Let p, q be the smallest indices > 0 such that $\pi_B(v_p) \geq \pi_B(v)$ and $\pi_B(v_q) > \pi_B(v)$ ($p = \ell$ and/or $q = \ell$ if no such index exists). Clearly $p \leq q$. Also, note that $\mathsf{anc}(v) = v_p$: indeed, v_p is necessarily the anchor of v_{p-1}, then it is in the anchor list of each v_i for i from $p-1$ to 1, so it is picked as the anchor of v.

Let $x = \lfloor \mathrm{rank}(v)/B^{\pi_B(v)+1} \rfloor$, we show by induction that $xB^{\pi_B(v)+1} \leq \mathrm{rank}(v_i) < (x+1)B^{\pi_B(v)+1}$ for all $0 \leq i \leq q$. This is by definition of x for $i = 0$. For any $0 < i < q$, note that we have $\pi_B(v_i) \leq \pi_B(v)$. Thus, $xB^{\pi_B(v)+1} \leq \mathrm{rank}(v_{i+1}) < (x+1)B^{\pi_B(v)+1}$ by definition of $\pi_B(v_i)$.

In particular for $v_i = \mathsf{anc}(v) = v_p$, $xB^{\pi_B(v)+1} \leq \mathrm{rank}(\mathsf{anc}(v)) \leq \mathrm{rank}(v) < (x+1)B^{\pi_B(v)+1}$, so $|\mathcal{R}_v^{\mathsf{anc}(v)}| = \mathrm{rank}(v) - \mathrm{rank}(\mathsf{anc}(v)) \leq B^{\pi_B(v)+1}$.

Consider now v_p, and assume that $\pi_B(v_p) = \pi_B(v)$, then $v_p \neq v_q$, and the anchor of v_p is necessarily some vertex in $(v_{p+1}, \dots v_q)$. Repeating this process, we have $\mathsf{AL}(v) = (v, v_{p_0} = v_p, v_{p_1}, \dots) + \mathsf{AL}(v_q)$, where each v_{p_i} satisfies $\pi_B(v_{p_i}) = \pi_B(v)$. Thus, for each i there is an integer $x - B \leq y_i < x$ such that $\mathrm{rank}(v_{p_i+1}) < y_i B^{\pi_B(v)} \leq \mathrm{rank}(v_{p_i})$. Thus, there are at most B such vertices v_{p_i}, and $\mathsf{ad}(v) \leq B + \mathsf{ad}(v_q)$.

We can now show $\mathsf{ad}(v) \leq B\lfloor \log_B(\mathrm{rank}(v)) \rfloor$. More precisely, we show by induction that $\mathsf{ad}(v) \leq B(\lfloor \log_B(\mathrm{rank}(v)) \rfloor - \pi_B(v))$. Indeed, $\pi_B(v) \leq \log_B(\mathrm{rank}(v))$ for any v by definition. For a vertex v with $\pi_B(v) = \lfloor \log_B(\mathrm{rank}(v)) \rfloor$, $\mathrm{rank}(\mathrm{lp}(v)) < B^{\pi_B(v)}$, and necessarily $\mathsf{anc}(v) = \perp$, so $\mathsf{ad}(v) = 0$. Now for any other v, define v_q as above (with $\mathrm{rank}(v_q) < \mathrm{rank}(v)$ and $\pi_B(v_q) > \pi_B(v)$; by induction

$$\begin{aligned}
\mathsf{ad}(v_q) &\leq B(\lfloor \log_B(\mathrm{rank}(v_q)) \rfloor - \pi_B(v_q)) \\
&\leq B(\lfloor \log_B(\mathrm{rank}(v)) \rfloor - (\pi_B(v) + 1)) \\
&= B(\lfloor \log_B(\mathrm{rank}(v)) \rfloor - \pi_B(v)) - B \\
\text{so } \mathsf{ad}(v) &\leq \mathsf{ad}(v_q) + B \\
&\leq B(\lfloor \log_B(\mathrm{rank}(v)) \rfloor - \pi_B(v)).
\end{aligned}$$

◀

3.3 Possible extensions

We present two possible extensions to this algorithm for specific use cases, although not covered by the experimental evaluation.

Distributed node creations. If multiple agents may insert nodes in parallel, the main difficulty is to maintain a correct chain decomposition. Once this difficulty is overcome, each agent may compute the anchor and restricted top vertices for each node they insert, then publish the index entries along with the rest of the node data. This can in particular be useful in VCS settings where commits are decentralized.

A simple solution to maintain a chain decomposition is to ensure that any chain is extended by at most one agent. For instance, if Alice needs to insert a node u and has no available chain for u, then a new chain is spawned for u, and only Alice is "allowed" to insert new nodes in this chain. Felsner's algorithm [7] can be run by each agent independently, each with their own disjoint set of chain heads.

2-way insertions. In some settings, it can be desirable to insert nodes either as sources or sinks. For instance, if a server holds a long list of events, and a client needs to retrieve in priority the most recent events. Then the client will progressively download events starting from the most recent, but new events may also occur during this process, and need to be inserted on the other side of the graph.

A possible approach for this scenario would be to maintain 2 disjoint graphs, each with its own chain decomposition to perform reachability queries within each side. For queries spanning both sides of the graph, we further store, for each node, the *lowest* vertex in each chain of the other side at the time this vertex is created (or, at least, those lowest vertices that are not already stored in the anchor list). Given a reachability query between u and v, we look for the chain of v within the anchor list of u, and for the chain of u within the anchor list of v. Overall, the index size remains linear in the graph width, and query time should be multiplied by 2 for such wide queries.

4 Experimental Evaluation

We implemented the chain algorithm with both anchor strategies, as well as a selection of literature algorithms. We use a corpus of 1 000 randomly generated graphs, as well as 97 real-world DAGs. For each graph, we draw 5 000 random queries. Then, we build the index on each graph and perform all queries successively.

All algorithms were implemented in Python using standard data structures, in order to obtain homogeneous running times. Furthermore, no disk access is performed during the timed parts of the benchmark, as all operations are performed in RAM. We measure 3 quantities for each algorithm in each graph:

Memory. Index size was evaluated as the number of integers (mostly node identifiers) stored for each algorithm, counting both index and manifest.

Query time. The positive (resp. negative) query time is the average time to answer a positive (resp. negative) query. The overall *query time* is the mean of these two quantities. Since some algorithms perform better for one kind than the other, this helps have more homogeneous results, independent of whether we tested more or less positive queries.

Indexing time. The average time (in ms) per node needed to build the index for the whole graph (index construction / graph size).

All benchmark scripts are available at `https://doi.org/10.5281/zenodo.14792894` for replication. All plots are drawn in log scale. We now describe more precisely the graphs used in the benchmark, as well as the implemented algorithms.

4.1 Random Graph Benchmark

We generated 1 000 random graphs with a tight control on their size, width, and average degree. Concretely, we have 3 parameters:

- N: size of the graph (number of nodes)
- k: width of the graph ($k \leq N$)
- p: probability to add a new arc to a node

We use the following generation algorithm:

Start with k degree-0 nodes, remember each as a chain head.

For each new node u (repeat $N - k$ times), add an arc from a random chain head v, and replace v by u in the set of chain heads.

With probability p, add an arc from a random node $< u$ to u and repeat this step.

Randomly shuffle the list of in-neighbors (parents) of the node.

Overall, we produce N nodes in topological order, with width exactly k (the first k nodes form an anti-chain, and the nodes admit a partition into k paths), and expected average degree $1/(1-p)$. Note that some arcs may be drawn multiple times. The overall benchmark is obtained by generating graphs for all combinations of the following parameters:

$N \in \{5\,000, 10\,000, 15\,000, \ldots, 100\,000\}$
$k \in \{100, 200, 300, \ldots, 1\,000\}$
$p \in \{0.30, 0.60, 0.80, 0.90, 0.95\}$

4.2 Tests on the CARDS dataset

We also ran algorithms on real-life graphs selected in the CARDS dataset [13]. The whole dataset contains dependency graphs from various sources (package dependencies, git/mercurial repositories, etc.). For repositories, we only used the 20 largest graphs from each category, giving 97 graphs overall. We further truncated each graph to 100 000 nodes for more homogeneous running times.

4.3 Competing algorithms

Since we have no knowledge of existing incremental reachability indexes, we selected two algorithms with sub-linear insertion time allowing, at least, for fast update of the graph.

Hub Labeling (2-hop). In the 2-hop algorithm(s) [5], the index contains two lists of *hubs* for each node: one among its descendants and one among its ancestors. The central guarantee is that any pair of related nodes have a common hub. The query time and size is dominated by the size of the hub lists.

Append-only insertions are the worst-case for this index, as each ancestor of a node becomes a hub, and the index size becomes quadratic. We thus chose to compare with a *dynamic* algorithm by Zhu et al. [17] that allows graph insertions and deletions without recomputing the index from scratch (but does edit hub lists of earlier nodes). More precisely, we implemented the index building routine as described by Akiba et al [2], using the node ordering presented in Zhu et al. [17]; we did not count the additional memory needed to prepare for random node insertion/deletion. We also implemented a random ordering strategy, to evaluate the impact of the ordering strategy on the index performances.

BFL. In this algorithm [12], we pick a number A of *semaphores*. We store for each node (using length-A bit-vectors) the sets of semaphores present in the ancestors and descendants. Comparing 2 such bit vectors allows, for some pairs of nodes, to answer reachability queries in constant time (positively if a semaphore appears as ancestor of one node and descendant of the other, negatively if the ancestor semaphores of one are not included in the ancestors of the others, or similarly for descendants). When this comparison is inconclusive, a DFS is run from an endpoint testing each node along the way, until either a path is found after a positive test or all branches have been pruned by negative tests.

5 Results

We first compare in a unified setting all presented algorithms. Out of these, we select 4 *finalists* for which we analyse running-times and memory in more details. Finally, we run two selected algorithms on CARDS graphs [13] to ensure that the conclusions of the benchmark carry over to real-life graphs.

Comparing all algorithms. The plots in Figure 4 show the performances of the algorithms presented here. Each graph is assigned a *dimension*, defined as $N + 100 * k$. All graphs with the same dimension are grouped into a single point for each algorithm, with the query time

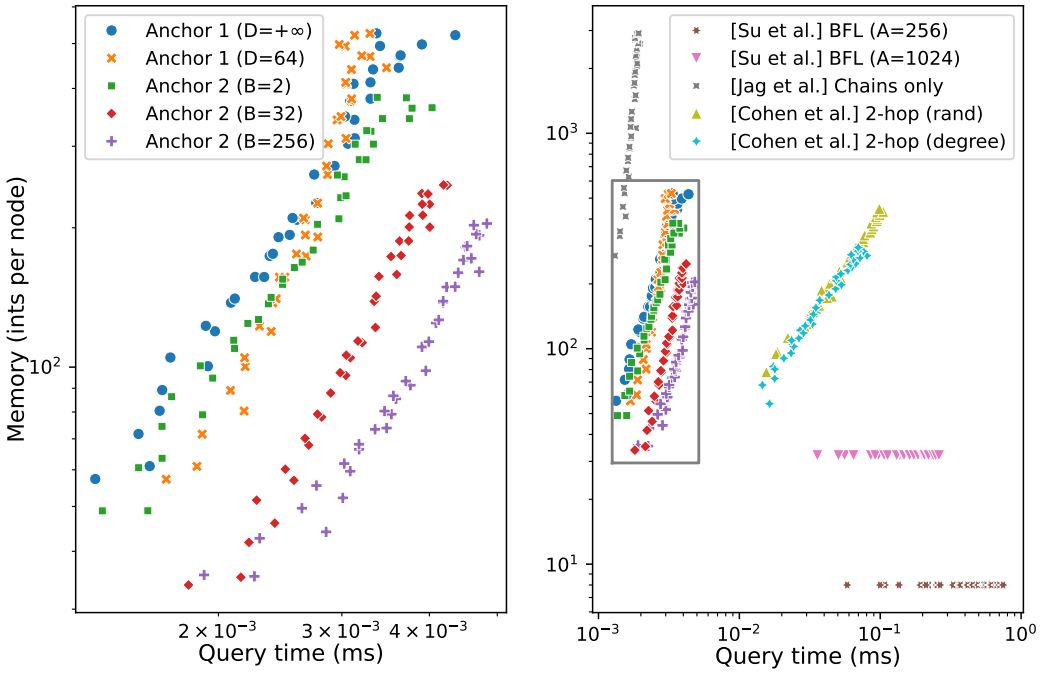

Figure 4 Memory and query time for each algorithm as the graph dimension grows.

as X coordinate and memory as y coordinate. The choice of the dimension allows to simplify the scatter plots, while keeping a good view of upper- and lower-bounds for each algorithms. Indeed, our anchor-based algorithms are mostly dependent on k, while other algorithms are mostly dependent on the graph size N. The right plot shows all algorithms, and the left plot gives a zoom-in on our anchor-based algorithms. Finally, Table 1 gives numeric results over the largest graphs of the data-set, corresponding to the top-right-most points of the plots.

Overall, each algorithm family behaves rather uniformly. As expected, chain-based approaches give the best (near-constant) query times, with a high memory cost for the original chains-only approach [9]. Algorithm BFL has prohibitive running times, even with up to 1024 semaphores. Considering 2-hop, the degree heuristic indeed reduces significantly the worst-case time and memory.

Detailed comparisons. We choose the following four algorithms for more extensive performance analysis. Our Anchor algorithms with strategy 1 ($D = 64$) and strategy 2 ($B = 256$) are the two "extreme" algorithms in the memory/time trade-off. We also include the original chain algorithm [9], as well as the 2-hop algorithm with degree heuristic, giving the best performances among literature algorithms on our benchmark.

Figure 5 helps compare these algorithms for each criteria as a function of each graph parameter. As mentioned already, the memory requirements for the original chain algorithm are prohibitive, even for the smallest graph width, however its query and indexing times remain optimal for all dimensions. The memory requirements for our anchor-based heuristics are comparable with those of 2-hop in this favorable setting with bounded graph width, and queries remain an order of magnitude faster. Considering the degree of the graph, it can be seen that it mainly affects the indexing time (since all algorithms need to visit each arc at least once); however, it does not have a significant impact on the index performances.

Table 1 Average performances of all compared algorithms over the 30 largest graphs in the benchmark ($N \geq 90\,000$, $k \geq 900$).

Algorithm	Query (ms)	Memory (ints/ node)	Indexing (ms per node)
[Jagadish] Chains only	0.002	2770.697	1.614
Anchor 1 (D=$+\infty$)	0.004	498.900	8.085
Anchor 1 (D=64)	0.003	498.602	7.778
Anchor 2 (B=2)	0.004	363.158	2.737
Anchor 2 (B=32)	0.004	236.599	4.035
Anchor 2 (B=256)	0.005	193.859	5.568
[Cohen et al.] 2-hop (degree)	0.070	283.423	36.269
[Cohen et al.] 2-hop (rand)	0.099	428.571	82.426
[Su et al.] BFL (A=1024)	0.261	32.000	0.008
[Su et al.] BFL (A=256)	0.744	8.000	0.007

On CARDS dataset. Figure 7 shows the performances of our Anchor algorithm (with the lowest memory needs, i.e. Strategy 2 with $B = 256$) compared with 2-Hop. Unsurprisingly, the worst-case query times appear for 2-hop. Considering memory, there appear to be many very simple graphs (e.g. with isolated nodes or small paths): on such graphs, the anchor algorithm's memory is dominated by constant-size overheads (chain assignments, anchors, powers, etc.) rather than chain tops themselves, while 2-hop has no such overhead and also maintains very short hub lists. Detailed results are given in Table 2 in appendix.

Conclusions. We can see that strategy 2 (with a large enough value of B) has comparable or lower memory needs than 2-hop for all values of k up to $1.5\sqrt{N}$. Compared to the original chain algorithm, we maintain bounded query and indexing times (within a factor 2 of the original), but improve by a factor 10 the memory requirements.

Performance with a real-life dataset indicates that our proof-of-concept algorithm performs similarly to our implementation of 2-hop, within the same order of magnitude.

Clearly, our benchmark does not cover all existing algorithms for the reachability problem, nor does it include the most efficient ones. The generated graphs are also very constrained, in order to precisely see the limits of each algorithm. The overarching goal is to ensure that performances are within the same order of magnitude as standard algorithms, while introducing the incremental feature required in many applications. A more precise benchmark would need to simulate real-life operations such as rollbacks, cache management, concurrent access, etc. where the incremental feature alone can save orders of magnitude in index maintenance. We hope our work can help introduce efficient reachability indexes in real-world applications, instead of a slow DFS, to help them scale up to larger graphs.

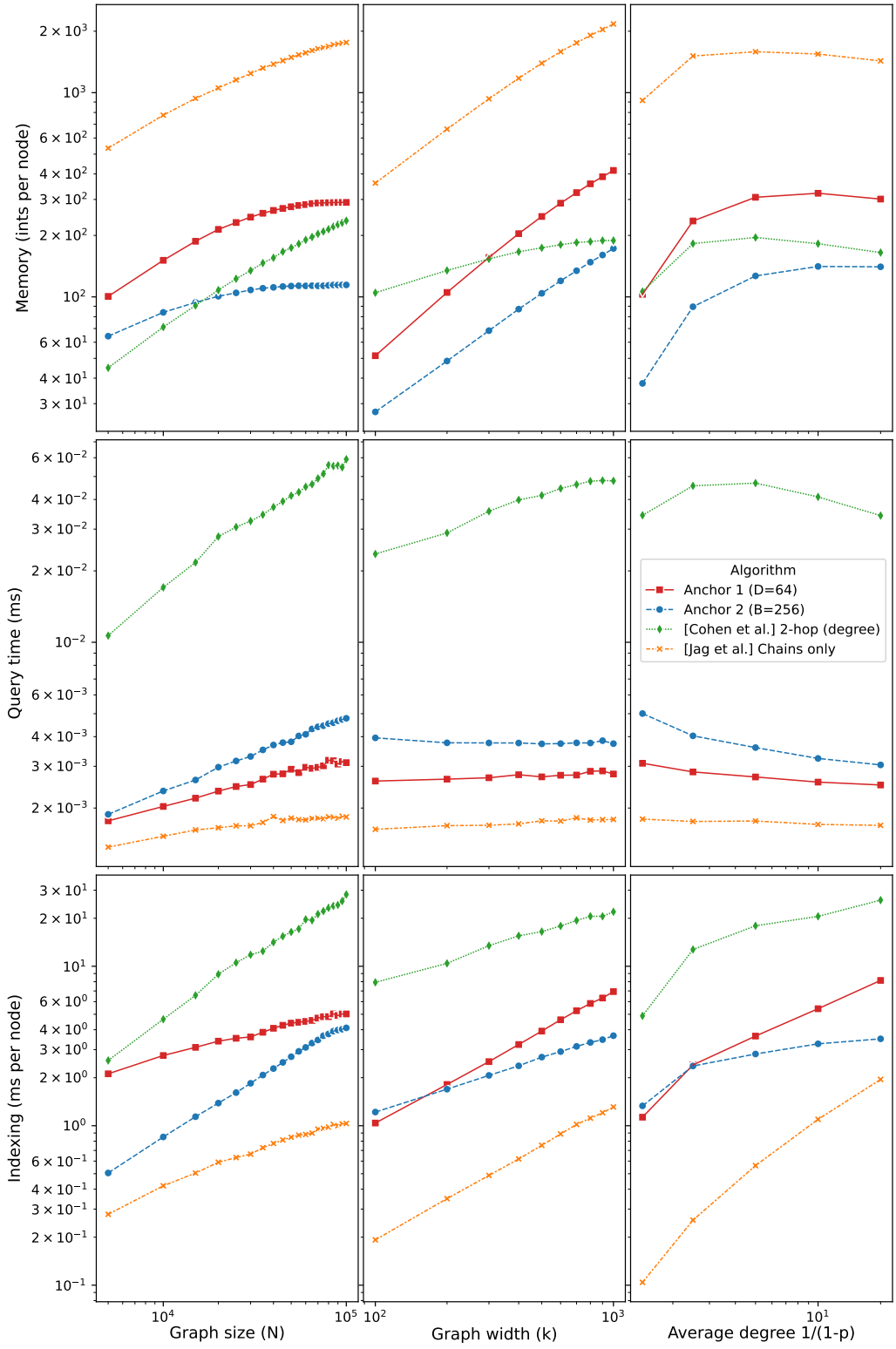

Figure 5 Plots comparing performances for a selection of 4 algorithms, across each criterion (rows) with respect to each graph parameter (columns). Each point in the first (resp. second, third) column of plots is the average over all 50 (resp. 100, 200) graphs with the given size (resp. width, degree).

References

1. 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.
2. Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 349–360, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2463676.2465315.
3. Bartlomiej Bosek, Stefan Felsner, Kamil Kloch, Tomasz Krawczyk, Grzegorz Matecki, and Piotr Micek. On-line chain partitions of orders: A survey. *Order*, 29(1):49–73, 2012. doi:10.1007/S11083-011-9197-1.
4. 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. doi:10.1109/ICDE.2008.4497498.
5. 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. doi:10.1137/S0097539702403098.
6. R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161–166, 1950. URL: http://www.jstor.org/stable/1969503.
7. Stefan Felsner. On-line chain partitions of orders. *Theor. Comput. Sci.*, 175(2):283–292, 1997. doi:10.1016/S0304-3975(96)00204-6.
8. Delbert Ray Fulkerson. Note on dilworth's decomposition theorem for partially ordered sets. In *Proceedings of the American Mathematical Society 7.4*, pages 701–702. AMS, 1956. URL: https://api.semanticscholar.org/CorpusID:119499686.
9. H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, December 1990. doi:10.1145/99935.99944.
10. Giorgos Kritikakis and Ioannis G. Tollis. Fast reachability using DAG decomposition. In Loukas Georgiadis, editor, *21st International Symposium on Experimental Algorithms, SEA 2023, July 24-26, 2023, Barcelona, Spain*, volume 265 of *LIPIcs*, pages 2:1–2:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.SEA.2023.2.
11. 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. doi:10.1007/978-3-030-73197-7_52.
12. 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. doi:10.1109/TKDE.2016.2631160.
13. Euxane Tran-Girard, Laurent Bulteau, and Pierre-Yves David. CARDS: A collection of package, revision, and miscellaneous dependency graphs. In *22nd International Conference on Mining Software Repositories (MSR)*, Ottawa, Canada, 2025. To appear. Download at https://doi.org/10.5281/zenodo.14245890.
14. 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. doi:10.14778/2732977.2732992.
15. 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. arXiv:1301.0977.
16. 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. doi:10.1145/2588555.2612181.
17. 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*, SIGMOD '14, pages 1323–1334, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2588555.2612181.

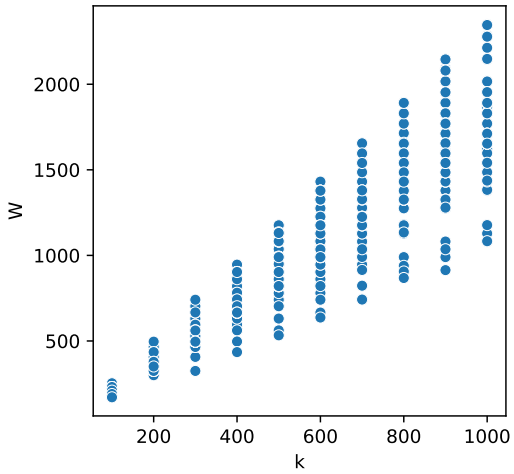

Figure 6 Number of chains produced by Felsner's algorithm (W) [7] compared to the actual width of the graph (k) for generated random graphs. Although $W = O(k^2)$ in the worst case, in our setting we have $W = 1.85k$ on average, and $W \leq 2.53k$ for all graphs.

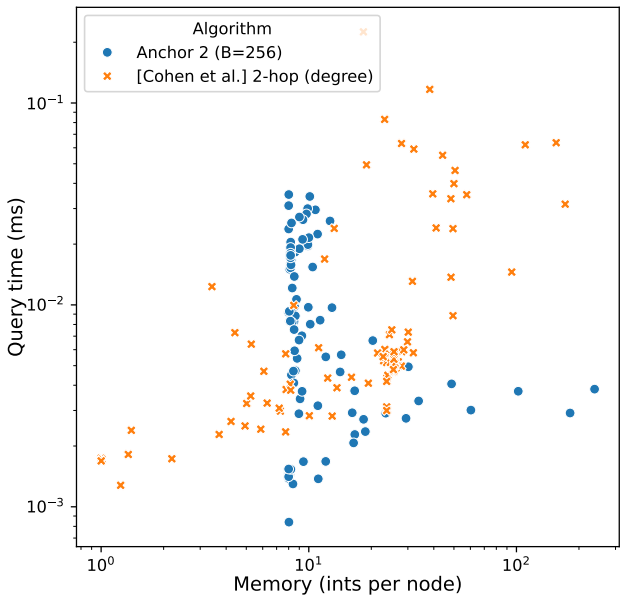

Figure 7 Performances of Anchor and 2-hop on 97 truncated graphs from the CARDS dataset [13]. Each dot represents one algorithm execution on one graph of the dataset. Unsurprisingly, the worst-case query time appear for 2-hop.

Table 2 Details of the CARDS dataset [13]. There are 18 families split into 4 main types: VCS graphs, package dependencies, library dependencies, and miscellaneous. We limited to 20 graphs per family and 100 000 nodes per graph. Each result is an average over all graphs of the family. The reachability ratio is the proportion of pairs of nodes $\{u, v\}$ such that $u \to^* v$. The width is the one computed by Felsner's algorithm [7].

Family	Graph count	Size	Average degree	Reachability ratio	Width W	Memory (ints/node) 2-hop	Memory (ints/node) Anchor	Query (ms) 2-hop	Query (ms) Anchor	Indexing (ms/node) 2-hop	Indexing (ms/node) Anchor
Mercurial	20	100000	1.05	93.04%	151	26	**8**	**0.005**	0.017	**0.284**	1.289
Git	20	100000	1.15	60.25%	16005	26	**10**	**0.005**	0.022	**0.323**	39.712
Debian	1	63701	4.35	0.25%	52726	32	**23**	0.059	**0.003**	7.663	**1.16**
Homebrew	4	7157	1.36	0.19%	6812	**7**	10	0.011	**0.002**	0.309	**0.109**
Rust	4	100000	3.0	0.97%	93507	23	**15**	0.025	**0.004**	7.28	**3.075**
Snap	3	54123	8.51	22.83%	40276	90	**53**	0.016	**0.003**	11.046	**7.021**
Archlinux	5	100000	7.22	3.44%	91826	31	**16**	0.041	**0.006**	18.905	**17.359**
Freebsd	3	36068	31.72	0.29%	31862	52	**17**	0.036	**0.002**	7.158	**0.809**
Nix	2	100000	2.77	30.43%	47750	22	**13**	0.027	**0.015**	**3.454**	15.293
NpmJS	3	100000	6.61	12.05%	78818	36	**20**	0.028	**0.006**	**6.637**	41.936
Perl	1	35361	9.49	8.3%	29810	41	**11**	0.024	**0.003**	10.289	**3.711**
Maven	7	100000	1.09	0.02%	95898	**7**	11	0.045	**0.002**	0.964	**0.507**
R	1	21056	0.37	0.0%	20516	**2**	8	0.002	**0.001**	**0.068**	0.112
Matrix	20	49582	0.87	6.63%	33213	9	**9**	**0.004**	0.009	**0.158**	1.854
DBLP	1	100000	4.23	3.74%	56743	**155**	238	0.064	**0.004**	23.186	**2.934**
Openalex	1	100000	2.05	0.86%	82966	**49**	60	0.024	**0.003**	3.654	**0.938**
LegiFrance	1	100000	1.53	0.08%	89734	**11**	14	0.006	**0.005**	**0.264**	0.911
Caselaw	1	100000	3.18	1.33%	70531	**110**	181	0.062	**0.003**	16.305	**1.479**
All	97	80525	3.02	34.44%	35712	25	**17**	0.015	**0.011**	**2.985**	11.57