O'Reach: Even Faster Reachability in Large Graphs

Kathrin Hanauer
University of Vienna, Faculty of Computer Science, Austria

Christian Schulz
Heidelberg University, Germany

Jonathan Trummer
University of Vienna, Faculty of Computer Science, Austria

Abstract

One of the most fundamental problems in computer science is the *reachability problem*: Given a directed graph and two vertices s and t, can s *reach* t via a path? We revisit existing techniques and combine them with new approaches to support a large portion of *reachability queries* in constant time using a linear-sized *reachability index*. Our new algorithm `O'Reach` can be easily combined with previously developed solutions for the problem or run standalone.

In a detailed experimental study, we compare a variety of algorithms with respect to their index-building and query times as well as their memory footprint on a diverse set of instances. Our experiments indicate that the query performance often depends strongly not only on the type of graph, but also on the result, i.e., *reachable* or *unreachable*. Furthermore, we show that previous algorithms are significantly sped up when combined with our new approach in almost all scenarios. Surprisingly, due to cache effects, a higher investment in space doesn't necessarily pay off: *Reachability queries* can often be answered even faster than single memory accesses in a precomputed full reachability matrix.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Mathematics of computing → Graph algorithms

Keywords and phrases Reachability, Static Graphs, Graph Algorithms, Reachability Index, Algorithm Engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2021.13

Related Version *Full Version*: `https://arxiv.org/abs/2008.10932` [12]

Supplementary Material *Software (Code Repository)*: `https://github.com/o-reach/O-Reach` archived at `swh:1:dir:55d23a5b940f1ead285729c8dbd82c71e28d504a`

Funding The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

1 Introduction

Graphs are used to model problem settings of various different disciplines. A natural question that arises frequently is whether one vertex of the graph can *reach* another vertex via a path of directed edges. *Reachability* finds application in a wide variety of fields, such as program and dataflow analysis [24, 25], user-input dependence analysis [27], XML query processing [34], and more [40]. Another prominent example is the Semantic Web which is composed of RDF/OWL data. These are often very huge graphs with rich content. Here, reachability queries are often necessary to deduce relationships among the objects.

There are two straightforward solutions to the reachability problem: The first is to answer each query individually with a graph traversal algorithm, such as breadth-first search (BFS) or depth-first search (DFS), in worst-case $\mathcal{O}(m + n)$ time and $\mathcal{O}(n)$ space. Secondly, we can precompute a full all-pairs reachability matrix in an initialization step and answer all ensuing queries in worst-case constant time. In return, this approach suffers from a space complexity of $\mathcal{O}(n^2)$ and an initialization time of $\mathcal{O}(n \cdot m)$ using the Floyd-Warshall algorithm [7, 35, 6] or starting a graph traversal at each vertex in turn. Alternatively, the initialization step can be performed in $\mathcal{O}(n^\omega)$ via fast matrix multiplication, where $\mathcal{O}(n^\omega)$ is the time required to multiply two $n \times n$ matrices ($2 \leq \omega < 2.38$ [20]). With increasing graph size, however, both the initialization time and space complexity of this approach become impractical. We therefore strive for alternative algorithms which decrease these complexities whilst still providing fast query lookups.

Contribution. In this paper, we study a variety of approaches that are able to support fast *reachability queries*. All of these algorithms perform some kind of preprocessing on the graph and then use the collected data to answer reachability queries in a timely manner. Based on simple observations, we provide a new algorithm, `O'Reach`, that can improve the query time for a wide range of cases over state-of-the-art reachability algorithms at the expense of some additional precomputation time and space or be run standalone. Furthermore, we show that previous algorithms are significantly sped up when combined with our new approach in almost all scenarios. In addition, we show that the expected query performance of various algorithms does not only depend on the type of graph, but also on the ratio of successful queries, i.e., with result *reachable*. Surprisingly, through cache effects and a significantly smaller memory footprint, especially unsuccessful *reachability queries* can be answered faster than single memory accesses in a precomputed reachability matrix.

2 Preliminaries

Terms and Definitions. Let $G = (V, E)$ be a simple directed graph with vertex set V and edge set $E \subseteq V \times V$. As usual, $n = |V|$ and $m = |E|$. An edge (u, v) is said to be *outgoing* at u and *incoming* at v, and u and v are called *adjacent*. The *out-degree* $\deg^+(u)$ (*in-degree* $\deg^-(u)$) of a vertex u is its number of outgoing (incoming) edges. A vertex without incoming (outgoing) edges is called a *source* (*sink*). The *out-neighborhood* $\mathsf{N}^+(v)$ (*in-neighborhood* $\mathsf{N}^-(v)$) of a vertex u is the set of all vertices v such that $(u, v) \in E$ ($(v, u) \in E$). The *reverse* of an edge (u, v) is an edge $(v, u) = (u, v)^{\mathrm{R}}$. The *reverse* G^{R} of a graph G is obtained by keeping the vertices of G, but substituting each edge $(u, v) \in E$ by its reverse, i.e., $G^{\mathrm{R}} = (V, E^{\mathrm{R}})$.

A sequence of vertices $s = v_0 \to \cdots \to v_k = t$, $k \geq 0$, such that for each pair of consecutive vertices $v_i \to v_{i+1}$, $(v_i, v_{i+1}) \in E$, is called an *s-t path*. If such a path exists, s is said to *reach* t and we write $s \to^* t$ for short, and $s \not\to^* t$ otherwise. The *out-reachability* $\mathsf{R}^+(u) = \{v \mid u \to^* v\}$ (*in-reachability* $\mathsf{R}^-(u) = \{v \mid v \to^* u\}$) of a vertex $u \in V$ is the set of all vertices that u can reach (that can reach u).

A *weakly connected component (WCC)* of G is a maximal set of vertices $C \subseteq V$ such that $\forall u, v \in C : u \to^* v$ in $G = (V, E \cup E^{\mathrm{R}})$, i.e., also using the reverse of edges. Note that if two vertices u, v reside in different WCCs, then $u \not\to^* v$ and $v \not\to^* u$. A *strongly connected component (SCC)* of G denotes a maximal set of vertices $S \subseteq V$ such that $\forall u, v \in S : u \to^* v \wedge v \to^* u$ in G. Contracting each SCC S of G to a single vertex v_S, called its *representative*, while preserving edges between different SCCs as edges between their corresponding representatives, yields the *condensation* G^{C} of G. We denote the SCC a vertex $v \in V$ belongs to by $\mathcal{S}(v)$. A directed graph G is *strongly connected* if it only has a single SCC and *acyclic* if each SCC is a singleton, i.e., if G has n SCCs. Observe that G and

G^{R} have exactly the same WCCs and SCCs and that G^{C} is a directed acyclic graph (DAG). Weakly connected components of a graph can be computed in $\mathcal{O}(n+m)$ time, e.g., via a breadth-first search that ignores edge directions. The strongly connected components of a graph can be computed in linear time [29] as well.

A *topological ordering* $\tau : V \to \mathbb{N}_0$ of a DAG G is a total ordering of its vertices such that $\forall(u,v) \in E : \tau(u) < \tau(v)$. Note that the topological ordering of G isn't necessarily unique, i.e., there can be multiple different topological orderings. For a vertex $u \in V$, the *forward topological level* $\mathcal{F}(u) = \min_\tau \tau(u)$, i.e., the minimum value of $\tau(u)$ among all topological orderings τ of G. Consequently, $\mathcal{F}(u) = 0$ if and only if u is a source. The *backward topological level* $\mathcal{B}(u)$ of $u \in V$ is the topological level of u with respect to G^{R} and $\mathcal{B}(u) = 0$ if and only if u is a sink. A topological ordering as well as the forward and backward topological levels can be computed in linear time [19, 30, 6], see also Sect. 4.

A *reachability query* Query(s,t) for a pair of vertices $s,t \in V$ is called *positive* and answered with `true` if $s \to^* t$, and otherwise *negative* and answered with `false`. Trivially, Query(v,v) is always `true`, which is why we only consider *non-trivial* queries between distinct vertices $s \neq t \in V$ from here on. Let $\mathcal{P}$ ($\mathcal{N}$) denote the set of all positive (negative) non-trivial queries of G, i.e., the set of all $(s,t) \in V \times V$, $s \neq t$, such that Query(s,t) is positive (negative). The *reachability* ρ in G is the ratio of positive queries among all non-trivial queries, i.e., $\rho = \frac{|\mathcal{P}|}{n(n-1)}$. Note, that due to the restriction to non-trivial queries[1], $0 \leq \rho \leq 1$. The *Reachability problem*, studied in this paper, consists in answering a sequence of reachability queries for arbitrary pairs of vertices on a given input graph G.

Basic Observations. With respect to processing a reachability Query(s,t) in a graph G for an arbitrary pair of vertices $s \neq t \in V$, the following basic observations are immediate and have partially also been noted elsewhere [22]:

(B1) If s is a sink or t is a source, then $s \not\to^* t$.
(B2) If s and t belong to different WCCs of G, then $s \not\to^* t$.
(B3) If s and t belong to the same SCC of G, then $s \to^* t$.
(B4) If $\tau(\mathcal{S}(t)) < \tau(\mathcal{S}(s))$ for any topological ordering τ of G^{C}, then $s \not\to^* t$.

As mentioned above, the precomputations necessary for Observations (B2) and (B3) can be performed in $\mathcal{O}(n+m)$ time. Note, however, that Observations (B3) and (B4) together are *equivalent* to asking whether $s \to^* t$: If $s \to^* t$ and $\mathcal{S}(s) \neq \mathcal{S}(t)$, then for every topological ordering τ, $\tau(\mathcal{S}(s)) < \tau(\mathcal{S}(t))$. Otherwise, if $s \not\to^* t$, a topological ordering τ with $\tau(\mathcal{S}(t)) < \tau(\mathcal{S}(s))$ can be computed by topologically sorting $G^{\mathrm{C}} \cup \{(\mathcal{S}(t), \mathcal{S}(s))\}$. Hence, the precomputations necessary for Observation (B4) would require solving the *Reachability* problem for all pairs of vertices already. Furthermore, a DAG can have exponentially many different topological orderings. In consequence, weaker forms are employed, such as the following [38, 39, 22] (see also Sect. 4):

(B5) If $\mathcal{F}(\mathcal{S}(t)) < \mathcal{F}(\mathcal{S}(s))$ w.r.t. G^{C}, then $s \not\to^* t$.
(B6) If $\mathcal{B}(\mathcal{S}(s) < \mathcal{B}(\mathcal{S}(t))$ w.r.t. G^{C}, then $s \not\to^* t$.

Assumptions. Following the convention introduced in preceding work [38, 39, 3, 22] (cf. Sect. 3), we only consider *Reachability* on DAGs from here on and implicitly assume that the condensation, if necessary, has already been computed and Observation (B3) has been applied. For better readability, we also drop the use of $\mathcal{S}(\cdot)$.

[1] Otherwise, $\frac{1}{n} \leq \rho$.

Table 1 Time and space complexity of reachability algorithms. Parameters: $k_{\texttt{IP}}$: #permutations, $h_{\texttt{IP}}$: #vertices with precomputed $\mathsf{R}^+(\cdot)$, $s_{\texttt{BFL}}$: size of Bloom filter (bits), ρ: reachability in G, d: #topological orderings, k: #supportive vertices, p: #candidates per supportive vertex.

Algorithm	Initialization Time	Index Size (Byte)	Queries: Time	Space
BFS/DFS	$\mathcal{O}(1)$	0	$\mathcal{O}(n+m)$	$\mathcal{O}(n)$
Full matrix	$\mathcal{O}(n \cdot (n+m))$	$n^2/8$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`PPL` [37]	$\mathcal{O}(n \log n + m)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
`PReaCH` [22]	$\mathcal{O}(m + n \log n)$	$56n$	$\mathcal{O}(1)$ / $\mathcal{O}(n+m)$	$\mathcal{O}(n)$
`IP`($k_{\texttt{IP}}$, $h_{\texttt{IP}}$) [36]	$\mathcal{O}((k_{\texttt{IP}} + h_{\texttt{IP}})(n+m))$	$\mathcal{O}((k_{\texttt{IP}} + h_{\texttt{IP}})n)$	$\mathcal{O}(k_{\texttt{IP}})$ / $\mathcal{O}(k_{\texttt{IP}} \cdot n \cdot \rho^2)$	$\mathcal{O}(n)$
`BFL`($s_{\texttt{BFL}}$) [28]	$\mathcal{O}(s_{\texttt{BFL}} \cdot (n+m))$	$2\lceil \frac{s_{\texttt{BFL}}}{8} \rceil n$	$\mathcal{O}(s_{\texttt{BFL}})$ / $\mathcal{O}(s_{\texttt{BFL}} \cdot n + m)$	$\mathcal{O}(n)$
`O'Reach`(d, k, p) (Sect. 4)	$\mathcal{O}((d + kp)(n+m))$	$(12 + 12d + 2\lceil \frac{k}{8} \rceil)n$	$\mathcal{O}(k + d + 1)$ / $\mathcal{O}(n+m)$	$\mathcal{O}(n)$

3 Related Work

A large amount of research on reachability indices has been conducted. Existing approaches can roughly be put into three categories: compression of transitive closure [14, 13, 2, 34, 15, 32], hop-labeling-based algorithms [5, 4, 26, 37, 16], as well as pruned search [18, 31, 38, 39, 22, 33, 36, 28]. As Merz and Sanders [22] noted, the first category gives very good query times for small networks, but doesn't scale very well to large networks (which is the focus of this work). Therefore, we do not consider approaches based on this technique more closely. Hop labeling algorithms typically build paths from labels that are stored for each vertex. For example in 2-hop labeling, each vertex stores two sets containing vertices it can reach in the given graph as well as in the reverse graph. A query can then be reduced to the set intersection problem. Pruned-search-based approaches precompute information to speed up queries by pruning the search.

Due to its volume, it is impossible to compare against all previous work. We mostly follow the methodology of Merz and Sanders [22] and focus on five recent techniques. The two most recent hop-labeling-based approaches are `TF` [3] and `PPL` [37]. In the pruned search category, the three most recent approaches are `PReaCH` [22], `IP` [36], and `BFL` [28]. We now go into more detail:

`TF`. The work by Cheng et al. [3] uses a data structure called topological folding. On the condensation DAG, the authors define a topological structure that is obtained by recursively folding the structure in half each time. Using this topological structure, the authors create labels that help to quickly answer reachability queries.

`PPL`. Yano et al. [37] use pruned landmark labeling and pruned path labeling as labels for their reachability queries. In general, the method follows the 2-hop labeling technique mentioned above, which stores sets of vertices for each vertex v and reduces queries to the set intersection problem. Their techniques are able to reduce the size of the stored labels and hence to improve query time and space consumption.

`PReaCH`. Merz and Sanders [22] apply the approach of *contraction hierarchies* (CHs) [9, 10] known from shortest-path queries to the reachability problem. The method first tries to answer queries by using pruning and precomputed information such as topological levels (Observation (B5) and (B6)). It adopts and improves techniques from `GRAIL` [38, 39] for that task, which is distinctly outperformed by `PReaCH` in the subsequent experiments. Should these techniques not answer the query, `PReaCH` instead performs a bidirectional breadth-first search (BFS) using the computed hierarchy, i.e., for a QUERY(s, t) the BFS only considers neighboring vertices with larger topological level and along the CH. The overall approach is simple and guarantees linear space and near linear preprocessing time.

`IP`. Wei et al. [36] use a randomized labeling approach by applying independent permutations on the labels. Contrary to other labeling approaches, `IP` checks for set-containment instead of set-intersection. Therefore, `IP` tries to answer negative queries by checking for at least one vertex that it is contained in only one of the two sets, where each set can consist of at most $k_{\texttt{IP}}$ vertices. If this test fails, `IP` checks another label, which contains precomputed reachability information from the $h_{\texttt{IP}}$ vertices with largest out-degree, and otherwise falls back to depth-first-search (DFS).

`BFL`. Su et al. [28] propose a labeling method which is based on `IP`, but additionally uses Bloom filters for storing and comparing labels, which are then used to answer negative queries. As parameters, `BFL` accepts $s_{\texttt{BFL}}$ and $d_{\texttt{BFL}}$, where $s_{\texttt{BFL}}$ denotes the length of the Bloom filters stored for each vertex and $d_{\texttt{BFL}}$ controls the false positive rate. By default, $d_{\texttt{BFL}} = 10 \cdot s_{\texttt{BFL}}$.

Table 1 subsumes the time and space complexities of the new algorithm `O'Reach` that we introduce in Sect. 4 as well as all algorithms mentioned in this paper except for `TF`, where the expressions describing the theoretical complexities are bulky and quite complex themselves.

4 O'Reach: Faster Reachability via Observations

In this section we propose our new algorithm `O'Reach`, which is based on a set of simple, yet powerful observations that enable us to answer a large proportion of reachability queries in constant time and brings together techniques from both hop labeling and pruned search. Unlike regular hop-labeling-approaches, however, its initialization time is linear. As a further plus, our algorithm is configurable via multiple parameters and extremely space-efficient with an index of only 38n Byte in the most space-saving configuration that could handle all instances used in Sect. 5 and uses all features.

Overview. The hop labeling technique used in our algorithm is inspired by a recent result for experimentally faster reachability queries in a dynamic graph by Hanauer et al. [11]. The idea here is to speed up reachability queries based on a selected set of so-called *supportive vertices*, for which complete out- and in-reachability is maintained explicitly. This information is used in three simple observations, which allow to answer matching queries in constant time. In our algorithm, we transfer this idea to the static setting. We further increase the ratio of queries answerable in constant time by a new perspective on topological orderings and their conflation with depth-first search, which provides additional reachability information and further increases the ratio of queries answerable in constant time. In case that we cannot answer a query via an observation, we fall back to either a pruning bidirectional breadth-first search or one of the existing algorithms.

In the following, we switch the order and first discuss topological orderings in depth, followed by our adaptation of supportive vertices. For both parts, consider a reachability $\textsc{Query}(s,t)$ for two vertices $s, t \in V$ with $s \neq t$.

4.1 Extended Topological Orderings

Taking up on the observation that topological orderings can be used to answer a reachability query decisively negative, we first investigate how Observation (B4) can be used most effectively in practice. Before we dive deeper into this subject, let us briefly review some facts concerning topological orderings and reachability in general.

▶ **Theorem 1.** *Let $\mathcal{N}(\tau) \subseteq \mathcal{N}$ denote the set of negative queries a topological ordering τ can answer, i.e., the set of all $(s,t) \in \mathcal{N}$ such that $\tau(t) < \tau(s)$, and let $\rho^-(\tau) = \mathcal{N}(\tau)/\mathcal{N}$ be the answerable negative query ratio.*

(i) *The reachability in any DAG is at most 50%. In this case, the topological ordering is unique.*

(ii) *Any topological ordering τ witnesses the non-reachability between exactly 50% of all pairs of distinct vertices. Therefore, $\rho^-(\tau) \geq 50\%$.*

(iii) *Every topological ordering of the same DAG can answer the same ratio of all negative queries via Observation (B4), i.e., for two topological orderings τ, τ': $\rho^-(\tau) = \rho^-(\tau')$.*

(iv) *For two different topological orderings $\tau \neq \tau'$ of a DAG, $\mathcal{N}(\tau) \neq \mathcal{N}(\tau')$.*

Proof. Let G be a directed acyclic graph (DAG).

(i) As G is acyclic, there is at least one topological ordering τ of G. Then, for every edge (u,v) of G, $\tau(u) < \tau(v)$, which implies that each vertex u can reach at most all those vertices $w \neq u$ with $\tau(u) < \tau(w)$. Consequently, a vertex u with $\tau(u) = i$ can reach at most $n - i - 1$ *other* vertices (note that $i \geq 0$). Thus, the reachability in G is at most $\frac{1}{n(n-1)} \sum_{i=0}^{n-1} (n - i - 1) = \frac{1}{n(n-1)} \sum_{j=0}^{n-1} j = \frac{n(n-1)}{n(n-1)\cdot 2} = \frac{1}{2}$. Conversely, assume that the reachability in G is $\frac{1}{2}$. Then, each vertex u with $\tau(u) = i$ reaches exactly all $n-i-1$ other vertices ordered after it, which implies that there exists no other topological ordering τ' with $\tau'(u) > \tau(u)$. By induction on i, the topological ordering of G is unique.

(ii) Let τ be an arbitrary topological ordering of G. Then, each vertex u with $\tau(u) = i$ can certainly *reach* those vertices v with $\tau(v) < \tau(u)$. Hence, τ witnesses the non-reachability of exactly $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ pairs of distinct vertices.

(iii) As Observation (B4) corresponds exactly to the non-reachability between those pairs of vertices witnessed by the topological ordering, the claim follows directly from (ii).

(iv) As $\tau \neq \tau'$, there is at least one $i \in \mathbb{N}_0$ such that $\tau(u) = i = \tau'(v)$ and $u \neq v$. Let $j = \tau(v)$. If $j > i$, the number of non-reachabilities from v to another vertex witnessed by τ exceeds the number of those witnessed by τ', and falls behind it otherwise. In both cases, the difference in numbers immediately implies a difference in the set of vertex pairs, which proves the claim. ◀

In consequence, it is pointless to look for one particularly good topological ordering. Instead, to get the most out of Observation (B4), we need topological orderings whose sets of answerable negative queries differ greatly, such that their union covers a large fraction of $\mathcal{N}$. Note that both forward and backward topological levels each represent the set of topological orderings that can be obtained by ordering the vertices in blocks grouped by their level and arbitrarily permuting the vertices in each block. Different algorithms [19, 29, 6] for computing a topological ordering in linear time have been proposed over the years, with Kahn's algorithm [19] in combination with a queue being one that always yields a topological ordering represented by forward topological levels. We therefore complement the forward and backward topological levels by stack-based approaches, as in Kahn's algorithm [19] in combination with a stack or Tarjan's DFS-based algorithm [29] for computing the SCCs of a graph, which as a by-product also yields a topological ordering of the condensation. To diversify the set of answerable negative queries further, we additionally randomize the order in which vertices are processed in case of ties and also compute topological orderings on the reverse graph, in analogy to backward topological levels.

We next show how, with a small extension, the stack-based topological orderings mentioned above can be used to additionally answer positive queries. To keep the description concise, we concentrate on Tarjan's algorithm [29] in the following and reduce it to the part relevant for obtaining a topological ordering of a DAG. In short, the algorithm starts a depth-first

(a)

```
procedure ExtendedTopSort($G = (V, E), S$)
    for all $v \in V$ do $v$.visited ← false
    $i \leftarrow n - 1$ ;   initialize $\tau$, $\tau_H$, $\tau_X$ empty
    for all $s \in S$ in random order do Visit($s$)
    procedure Visit($v$)
        if $v$.visited then return
        $v$.visited ← true;   $\tau_H(v) \leftarrow i$;   $\tau_X(v) \leftarrow i$
        for $u \in \mathsf{N}^+(v)$ in random order do
            Visit($u$);   $\tau_X(v) \leftarrow \max(\tau_X(v), \tau_X(u))$
        $\tau(v) \leftarrow i$ ;   $i \leftarrow i - 1$
    return $\tau$, $\tau_H$, $\tau_X$
```

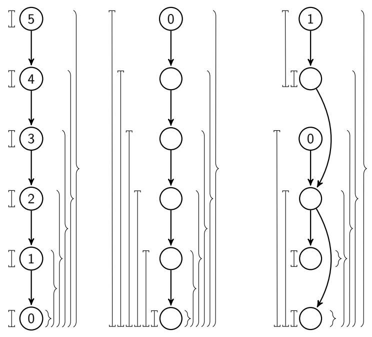
(b) 

Figure 1 (a): Extended Topological Sorting. (b): Three extended topological orderings of two graphs: The labels correspond to the order in the start set S. If the label is empty, the vertex need not be in S or can have any larger number. The brackets to the left show the range $[\tau(v), \tau_H(v)]$, the braces to the right the range $[\tau(v), \tau_X(v)]$.

search at an arbitrary vertex $s \in S$, where $S \subseteq V$ is a given set of vertices to start from. Whenever it visits a vertex v, it marks v as visited and recursively visits all unvisited vertices in its out-neighborhood. On return, it *prepends* v to the topological ordering. A loop over $S = V$ ensures that all vertices are visited. Note that although the vertices are visited in DFS order, the topological ordering is different from a DFS numbering as it is constructed "from back to front" and corresponds to a reverse sorting according to what is also called *finishing time* of each vertex.

To answer positive queries, we exploit the invariant that when visiting a vertex v, all yet unvisited vertices reachable from v will be prepended to the topological ordering prior to v being prepended. Consequently, v can *certainly* reach all vertices in the topological ordering between v and, exclusively, the vertex w that was at the front of the topological ordering when v was visited. Let x denote the vertex preceding w in the final topological ordering, i.e., the vertex with the largest index that was reached recursively from v. For a topological ordering τ constructed in this way, we call $\tau(x)$ the *high index* of v and denote it with $\tau_H(v)$. Furthermore, v *may* be able to also reach w and vertices beyond, which occurs if $v \rightarrow^* y$ for some vertex y, but y had already been visited earlier. We therefore additionally track the *max index*, the largest index of any vertex that v can reach, and denote it with $\tau_X(v)$. Figure 1a shows how to compute an extended topological ordering with both high and max indices in pseudo code and highlights our extensions. Compared to Tarjan's original version [29], the running time remains unaffected by our modifications and is still in $\mathcal{O}(n + m)$.

Note that neither max nor high indices yield an ordering of V: Every vertex that is visited recursively starting from v and before vertex x with $\tau(x) = \tau_H(v)$, inclusively, has the same high index as v, and the high index of each vertex in a graph consisting of a single path, e.g., would be $n - 1$. In particular, neither max nor high index form a DFS numbering and also differ in definition and use from the DFS finishing times $\hat{\phi}$ used in PReaCH, where a vertex v can *certainly* reach vertices with DFS number up to $\hat{\phi}$ and *certainly none* beyond. Conversely, v may be able to also reach vertices with smaller DFS number than its own, which cannot occur in a topological ordering.

If ExtendedTopSort is run on the reverse graph, it yields a topological ordering τ' and high and max indices τ'_H and τ'_X, such that reversing τ' yields again a topological ordering τ of the original graph. Furthermore, $\tau_L(v) := n - 1 - \tau'_H(v)$ is a *low index* for each vertex v, which denotes the smallest index of a vertex in τ that can certainly reach v, i.e., the out-reachability of v is replaced by in-reachability. Analogously, $\tau_N(v) := n - 1 - \tau'_X(v)$ is a *min index* in τ and no vertex u with $\tau(u) < \tau_N(v)$ can reach v.

The following observations show how such an extended topological ordering τ can be used to answer both positive and negative reachability queries:

(T1) If $\tau(s) \leq \tau(t) \leq \tau_H(s)$, then $s \rightarrow^* t$. **(T4)** If $\tau_L(t) \leq \tau(s) \leq \tau(t)$, then $s \rightarrow^* t$.
(T2) If $\tau(t) > \tau_X(s)$, then $s \not\rightarrow^* t$. **(T5)** If $\tau(s) < \tau_N(t)$, then $s \not\rightarrow^* t$.
(T3) If $\tau(t) = \tau_X(s)$, then $s \rightarrow^* t$. **(T6)** If $\tau(s) = \tau_N(t)$, then $s \rightarrow^* t$.

Recall that by definition, $\tau(s) \leq \tau_H(s) \leq \tau_X(s)$ and $\tau_N(t) \leq \tau_L(t) \leq \tau(t)$. Figure 1b depicts three examples for extended topological orderings. In contrast to negative queries, not every extended topological ordering is equally effective in answering positive queries, and it can be arbitrarily bad, as shown in the extremes on the left (worst) and at the center (best) of Figure 1b:

▶ **Theorem 2.** *Let $\mathcal{P}(\tau) \subseteq \mathcal{P}$ be the set of positive queries an extended topological ordering τ can answer and let $\rho^+(\tau) = \mathcal{P}(\tau)/\mathcal{P}$ be the answerable positive query ratio. Then,* $0 \leq \rho^+(\tau) \leq 1$.

Instead, the effectiveness of an extended topological ordering depends positively on the size of the ranges $[\tau(v), \tau_H(v)]$ and $[\tau_L(v), \tau(v)]$, and negatively on $[\tau_H(v), \tau_X(v)]$ and $[\tau_N(v), \tau_L(v)]$ which in turn depend on the recursion depths during construction and the order of recursive calls. The former two can be maximized if the first, non-recursive call to Visit in line 4 in ExtendedTopSort always has a source as its argument, i.e., if the algorithm's parameter S corresponds to the set of all sources. Clearly, this still guarantees that every vertex is visited.

In addition to the forward and backward topological levels, `O'Reach` thus computes a set of d extended topological orderings starting from sources, where d is a tuning parameter, and $d/2$ of them are obtained via the reverse graph. It then applies Observation (B4) as well as Observations (T1)–(T6) to all extended topological orderings.

4.2 Supportive Vertices

We now show how to apply and improve the idea of supportive vertices in the static setting. A vertex v is *supportive* if the set of vertices that v can reach and that can reach v, $R^+(v)$ and $R^-(v)$, respectively, have been precomputed and membership queries can be performed in sublinear time. We can then answer reachability queries using the following simple observations [11]:

(S1) If $s \in R^-(v)$ and $t \in R^+(v)$ for *any* $v \in V$, then $s \rightarrow^* t$.
(S2) If $s \in R^+(v)$ and $t \notin R^+(v)$ for *any* $v \in V$, then $s \not\rightarrow^* t$.
(S3) If $s \notin R^-(v)$ and $t \in R^-(v)$ for *any* $v \in V$, then $s \not\rightarrow^* t$.

To apply these observations, our algorithm selects a set of k supportive vertices during the initialization phase. In contrast to the original use scenario in the dynamic setting, where the graph changes over time and it is difficult to choose "good" supportive vertices that can help to answer many queries, the static setting leaves room for further optimizations here: With respect to Observation (S1), we consider a supportive vertex v "good" if $|R^+(v)| \cdot |R^-(v)|$ is large as it maximizes the possibility that $s \in R^-(v) \wedge t \in R^+(v)$. With respect to Observation (S2) and (S3), we expect a "good" supportive vertex to have out- or in-reachability sets, respectively, of size close to $\frac{n}{2}$, i.e., when $|R^+(v)| \cdot |V \setminus R^+(v)|$ or $|R^-(v)| \cdot |V \setminus R^-(v)|$, respectively, are maximal. Furthermore, to increase total coverage and avoid redundancy, the set of queries Query(s,t) covered by two different supportive vertices should ideally overlap as little as possible.

`O'Reach` takes a parameter k specifying the number of supportive vertices to pick. Intuitively speaking, we expect vertices in the topological "mid-levels" to be better candidates than those at the ends, as their out- and in-reachabilities (or non-reachabilities) are likely to be more balanced. Furthermore, if *all* vertices on one forward (backward) level i were supportive, then *every* $\textsc{Query}(s,t)$ with $\mathcal{F}(s) < i < \mathcal{F}(t)$ $(\mathcal{B}(t) < i < \mathcal{B}(s))$ could be answered using only Observation (S1). As finding a "perfect" set of supportive vertices is computationally expensive and we strive for linear preprocessing time, we experimentally evaluated different strategies for the selection process. Due to page limits, we only describe the most successful one: A forward (backward) level i is called *central*, if $\frac{1}{5}L_{\max} \leq i \leq \frac{4}{5}L_{\max}$, where $L_{\max}$ is the maximum topological level. A level i is called *slim* if there at most h vertices having this level, where h is a parameter to `O'Reach`. We first compute a set of candidates of size at most $k \cdot p$ that contains all vertices on slim forward or backward levels, arbitrarily discarding vertices as soon as the threshold $k \cdot p$ is reached. p is another parameter to `O'Reach` and together with k controls the size of the candidate set. If the threshold is not reached, we fill up the set of candidates by picking the missing number of vertices uniformly at random from all other vertices whose forward level is central. In the next step, the out- and in-reachabilities of all candidates are obtained and the k vertices v with largest $|R^{+}(v)| \cdot |R^{-}(v)|$ are chosen as supportive vertices. This strategy primarily optimizes for Observation (S1), but worked better in experiments than strategies that additionally tried to optimize for Observation (S2) and (S3). The time complexity of this process is in $\mathcal{O}(kp(n+m) + kp\log(kp))$.

We remark that this is a general-purpose approach that has shown to work well across different types of instance, albeit possibly at the expense of an increased initialization time. It seems natural that more specialized routines for different graph classes can improve both running time and coverage.

4.3 The Complete Algorithm

Given a graph G and a sequence of queries Q, we summarize in the following how `O'Reach` proceeds. During initialization, it performs the following steps:

Step 1: Compute the WCCs
Step 2: Compute forward/backward topological levels
Step 3: Obtain d random extended topological orderings
Step 4: Pick k supportive vertices, compute $\mathsf{R}^{+}(\cdot)$ and $\mathsf{R}^{-}(\cdot)$

Steps 1 and 2 run in linear time. As shown in Sect. 4.1 and Sect. 4.2, the same applies to Steps 3 and 4, assuming that all parameters are constants. The required space is linear for all steps. The reachability index consists of the following information for each vertex v: one integer for the WCC, one integer each for $\mathcal{F}(v)$ and $\mathcal{B}(v)$, three integers for each of the d extended topological orderings τ $(\tau(v), \tau_H(v)/\tau_L(v), \tau_X(v)/\tau_N(v))$, two bits for each of the k supportive vertices, indicating its reachability to/from v. For graphs with $n \leq 2^{32}$, 4 Byte per integer suffice. Furthermore, we group the bits encoding the reachabilities to and from the supportive vertices, respectively, and represent them each by one suitably sized integer, e.g., using `uint8_t` (8 bit), for $k \leq 8$ supportive vertices. As the smallest integer has at least 8 bit on most architectures, we store $12 + 12\mathrm{d} + 2 \cdot \lceil \frac{\mathrm{k}}{8} \rceil$ Byte per vertex.

For each query $\textsc{Query}(s,t)$, `O'Reach` tries to answer it using one of the observations in the order given below, which on the one hand has been optimized by some preliminary experiments on a small subset of benchmark instances (see Sect. 5 for details) and on the other hand strives for a fair alternation between "positive" and "negative" observations to avoid overfitting. Note that all observation-based tests run in constant time. As soon as one of them can answer the query affirmatively, the result is returned immediately. A test leading to a positive or negative answer is marked as ○ or ●, respectively.

Test 1: ○ $s = t$?

Test 2: ● ● topological levels (B5), (B6)

Test 3: ○ k supportive vertices, positive (S1)

Test 4: ● ○ ● ○ first topological ordering (B4), (T1), (T2), (T3)

Test 5: ● ● k supportive vertices, negative (S2), (S3)

Test 6: ● ○ ● ○ remaining $d - 1$ topological orderings (B4), (T1)/(T4), (T2)/(T5), (T3)/(T6)

Test 7: ● different WCCs (B2)

Observe that the tests for Observation (S1), (S2), and (S3) can each be implemented easily using boolean logic, which allows for a concurrent test of all supports whose reachability information is encoded in one accordingly-sized integer: For Observation (S1), it suffices to test whether $r^-(s) \wedge r^+(t) > 0$, and $r^+(s) \wedge \neg r^+(t) > 0$ and $\neg r^-(s) \wedge r^-(t) > 0$ for Observations (S2) and (S3), where r^+ and r^- hold the respective forward and backward reachability information in the same order for all supports. Each test hence requires at most one comparison of two integers plus at most two elementary bit operations. Also note that Observation (B1) is implicitly tested by Observations (B5) and (B6). Using the data structure described above, our algorithm requires at most one memory transfer for s and one for t for each QUERY(s, t) that is answerable by one of the observations. Note that there are more observations that allow to identify a negative query than a positive query, which is why we expect a more pronounced speedup for the former. However, as stated in Theorem 1, the reachability in DAGs is always less than 50 %, which justifies a bias towards an optimization for negative queries.

If the query can not be answered using any of these tests, we instead fall back to either another algorithm or a bidirectional BFS with pruning, which uses these tests for each newly encountered vertex v in a subquery QUERY(v, t) (forward step) or QUERY(s, v) (backward step). If a subquery can be answered decisively positive by a test, the bidirectional BFS can immediately answer QUERY(s, t) positively. Otherwise, if a subquery is answered decisively negative by a test, the encountered vertex v is no longer considered (pruning step). If the subquery could not be answered by a test, the vertex v is added to the queue as in a regular (bidirectional) BFS.

5 Experimental Evaluation

We evaluated our new algorithm `O'Reach` as a preprocessor to various recent state-of-the-art algorithms listed below against running these algorithms on their own. Furthermore, we use as an additional fallback solution the pruned bidirectional BFS (`pBiBFS`). Our experimental study follows the methodology in [22] and comprises the algorithms `PPL` [37], `TF` [3], `PReaCH` [22], `IP` [36], and `BFL` [28]. Moreover, our evaluation is the first that directly relates `IP` and `BFL` to `PReaCH` and studies the performance of `IP` and `BFL` separately for successful (*positive*) and unsuccessful (*negative*) reachability queries. For reasons of comparison, we also assess the query performance of a full reachability matrix by computing the transitive closure of the input graph entirely during initialization, storing it in a matrix using 1 bit per pair of vertices, and answering each query by a single memory lookup. We refer to this algorithm simply as `Matrix`. As the reachability in DAGs is small and cache locality can influence lookup times, we also experimented with various hash set implementations. However, none was faster or more memory-efficient than `Matrix`.

Setup and Methodology. We implemented `O'Reach` in C++14[2] with `pBiBFS` as built-in fallback strategy. For `PPL`[3], `TF`[3], `PReaCH`[4], `IP`[5], and `BFL`[6] we used the original C++ implementation in each case. All source code was compiled with GCC 7.5.0 and full optimization (`-O3`). The experiments were run on a Linux machine under Ubuntu 18.04 with kernel 4.15 on four AMD Opteron 6174 CPUs clocked at 2.2 GHz with 512 kB and 6 MB L2 and L3 cache, respectively and 12 cores per CPU. Overall, the machine has 48 cores and a total of 256 GB of RAM. Unless indicated otherwise, each experiment was run sequentially and exclusively on one processor and its local memory. As non-local memory accesses incur a much higher cost, an exception to this rule was only made for `Matrix`, where we would otherwise have been able to only run twelve instead of 29 instances. We also parallelized the initialization phase for `Matrix`, where the transitive closure is computed, using 48 threads. However, all queries were processed sequentially.

To counteract artifacts of measurement and accuracy, we ran each algorithm five times on each instance and in general use the median for the evaluation. As `O'Reach` uses randomization during initialization, we instead report the average running time over five different seeds. For `IP` and `BFL`, which are randomized in the same way, but don't accept a seed, we just give the average over five repetitions. We note that also taking the median instead or increasing the number of repetitions or seeds does not change the overall picture.

Instances. To facilitate comparability, we adopt the instances used in the papers introducing `PReaCH` [22] and `TF` [3], which overlap with those used to evaluate `IP` [36] and `BFL` [28], and which are available either from the `GRAIL` code repository[7] or the Stanford Network Analysis Platform SNAP [21]. Furthermore, we extended the set of benchmark graphs by further instance sizes and Delaunay graphs. Table 2 provides a short overview on the left side, more details are available in the full version [12]. As we only consider DAGs, all instances are condensations of their respective originals, if they were not acyclic already. We also adopt the grouping of the instances as in [39, 22] and provide only a short description of the different sets in the following.

Kronecker: These instances were generated by the RMAT generator for the Graph500 benchmark [23] and oriented acyclically from smaller to larger node ID. The name encodes the number of vertices 2^i as *kron_logni*. *Random:* Graphs generated according to the Erdős-Renyí model $G(n,m)$ and oriented acyclically from smaller to larger node ID. The name encodes $n = 2^i$ and $m = 2^j$ as *randni-j*. *Delaunay:* Delaunay graphs from the 10th DIMACS Challenge [1, 8]. *delaunay_ni* is a Delaunay triangulation of 2^i random points in the unit square. *Large real:* Introduced in [39], these instances represent citation networks (*citeseer.scc*, *citeseerx*, *cit-Patents*), a taxonomy graph (*go-uniprot*), as well as excerpts from the RDF graph of a protein database (*uniprotm22*, *uniprotm100*, *uniprotm150*). *Small real dense:* Among these instances, introduced in [17], are again citation networks (*arXiv*, *pubmed_sub*, *citeseer_sub*), a taxonomy graph (*go_sub*), as well as one obtained from a semantic knowledge database (*yago_sub*). *Small real sparse:* These instances were introduced in [18] and represent XML documents (*xmark*, *nasa*), metabolic networks (*amaze*, *kegg*) or originate from pathway and genome databases (all others). *SNAP:* The e-mail network graph

[2] Source code and instances are available from `https://oreach.taa.univie.ac.at`.
[3] Provided directly by the authors.
[4] `https://github.com/fiji-flo/preach2014/tree/master/original_code`
[5] `https://github.com/datourat/IP-label-for-graph-reachability`
[6] `https://github.com/BoleynSu/bfl`
[7] `https://code.google.com/archive/p/grail/`

Table 2 Left: Instance sizes (read $/10^3$: in thousands), density, and reachability. Right: Median initialization time in ms over five repetitions. Highlighted results are the overall best.

Instance	$n/10^3$	$m/10^3$	$\frac{m}{n}$	$\rho\%$	`O'Reach`	`PReaCH`	`PPL`	`TF`	`IP(s)`	`IP(d)`	`BFL(s)`	`BFL(d)`
kron_logn12	4.1	117.0	28.55	27.4760	451.0	13.5	56.5	46 555.2	22.6	53.0	**2.0**	4.0
kron_logn16	65.5	2 456.1	37.48	21.2187	13 045.7	602.4	1 869.5		685.5	1 283.0	**88.8**	118.3
kron_logn17	131.1	5 114.0	39.02	19.4544	31 835.0	1 425.8	4 268.9		1 611.7	2 897.9	**228.1**	288.1
kron_logn20	1 048.6	44 619.4	42.55	5.8195	380 698.0	20 791.9	62 836.0		22 788.2	37 103.7	**3 301.1**	3 999.0
kron_logn21	2 097.2	91 040.9	43.41	1.2150	812 416.0	46 559.0	151 870.0		49 988.1	79 226.0	**7 513.1**	9 014.9
randn20-21	1 048.6	2 097.2	2.00	0.0012	4 272.7	2 878.3	11 579.3	11 615.8	2 434.7	2 635.1	**626.1**	677.2
randn20-22	1 048.6	4 194.3	4.00	0.0352	5 706.9	4 459.6	43 761.5	47 679.2	3 364.6	3 704.3	**892.0**	976.9
randn20-23	1 048.6	8 388.6	8.00	1.9067	13 724.7	7 128.3	9 348 510.0		4 830.2	5 311.5	**1 287.7**	1 449.3
randn23-24	8 388.6	16 777.2	2.00	0.0001	46 043.5	28 959.1	132 570.0	122 270.0	24 566.7	25 906.9	**6 094.8**	6 580.6
randn23-25	8 388.6	33 554.4	4.00	0.0044	61 206.2	45 573.7	413 684.0	465 300.0	34 145.7	36 815.0	**8 964.7**	9 715.1
delaunay_n15	32.8	98.3	3.00	0.4380	104.4	38.9	174.2	602.1	42.5	55.3	**7.0**	9.0
delaunay_n20	1 048.6	3 145.7	3.00	0.0093	2 816.5	1 788.4	9 350.5	24 563.9	2 339.1	2 785.1	**299.8**	351.5
delaunay_n22	4 194.3	12 582.9	3.00	0.0020	11 402.7	7 363.9	38 674.1	108 297.0	10 106.6	11 911.6	**1 203.1**	1 394.5
citeseer.scc	693.9	312.3	0.45	0.0002	865.9	503.4	1 185.3	1 579.7	602.5	613.4	**107.0**	122.5
citeseerx	6 540.4	15 011.3	2.30	0.1367	90 695.8	12 545.7	73 061.0	145 773.0	11 208.0	11 807.4	**2 349.2**	2 700.0
cit-Patents	3 774.8	16 518.9	4.38	0.0409	22 358.6	15 989.7	393 412.0	342 680.0	13 098.4	14 384.0	**2 905.4**	3 210.1
go_uniprot	6 968.0	34 769.3	4.99	0.0004	28 270.0	11 858.8	34 660.6	90 942.4	11 935.8	13 381.6	**3 137.0**	3 701.2
uniprotenc_22m	1 595.4	1 595.4	1.00	0.0001	2 802.5	714.8	2 762.0	3 446.0	1 322.6	1 313.7	**147.8**	189.3
uniprotenc_100m	16 087.3	16 087.3	1.00	0.0000	39 539.9	10 420.6	30 967.4	59 660.2	16 089.1	16 194.7	**2 169.6**	2 639.2
uniprotenc_150m	25 037.6	25 037.6	1.00	0.0000	65 983.9	17 612.9	50 254.7	86 052.0	26 453.4	26 730.9	**3 830.4**	4 548.6
go_sub	6.8	13.4	1.97	0.2258	10.4	4.0	16.6	37.6	5.0	6.2	**1.0**	**1.0**
pubmed_sub	9.0	40.0	4.45	0.6458	19.4	9.1	31.3	101.5	8.9	10.8	**2.0**	3.0
yago_sub	6.6	42.4	6.38	0.1506	12.5	6.0	18.9	61.5	7.5	10.4	**1.1**	2.0
citeseer_sub	10.7	44.3	4.13	0.3672	25.3	11.3	48.4	131.9	11.8	15.3	**2.3**	3.0
arXiv	6.0	66.7	11.12	15.4643	223.2	9.7	60.8	10 008.7	14.9	26.3	**2.0**	3.0
amaze	3.7	3.6	0.97	17.2337	12.0	1.2	5.3	25.9	2.2	2.4	**0.0**	0.4
kegg	3.6	4.4	1.22	20.1636	16.3	1.4	6.8	18.3	2.7	2.8	**0.3**	0.5
nasa	5.6	6.5	1.17	0.5284	7.0	2.4	11.6	27.3	3.3	3.8	**1.0**	**1.0**
xmark	6.1	7.1	1.16	1.4513	10.7	2.3	12.9	24.2	3.9	4.3	**1.0**	**1.0**
vchocyc	9.5	10.3	1.09	0.1517	12.0	2.9	13.4	53.7	5.4	5.9	**1.0**	**1.0**
mtbrv	9.6	10.4	1.09	0.1511	11.1	3.0	13.7	24.0	5.4	6.0	**1.0**	**1.0**
anthra	12.5	13.1	1.05	0.0951	15.4	3.8	18.3	62.5	7.1	7.8	**1.0**	**1.0**
ecoo	12.6	13.4	1.06	0.1088	15.9	3.9	18.8	41.4	7.4	8.0	**1.0**	**1.0**
agrocyc	12.7	13.4	1.06	0.1060	16.1	3.9	19.1	48.1	7.4	8.1	**1.0**	**1.0**
human	38.8	39.6	1.02	0.0231	49.1	13.5	56.5	104.1	23.7	25.8	**3.0**	4.0
p2p-Gnutella31	48.4	55.3	1.14	0.7725	120.6	28.4	89.2	52.3	43.8	44.5	**5.0**	7.0
email-EuAll	230.8	223.0	0.97	5.0732	945.2	115.3	340.5	241.3	170.1	171.4	**24.8**	32.0
web-Google	371.8	517.8	1.39	14.8090	5 783.6	369.3	928.1	918.4	452.6	472.0	**73.8**	88.0
soc-LiveJournal1	970.3	1 024.1	1.06	5.3781	3 663.5	739.6	2 086.3	1 827.9	1 160.5	1 181.4	**142.3**	173.0
wiki-Talk	2 281.9	2 311.6	1.01	0.8117	6 347.0	1 492.1	4 317.8	2 715.4	2 597.7	2 620.7	**269.9**	343.5

(*email-EuAll*), peer-to-peer network (*p2p-Gnutella31*), social network (*soc-LiveJournal1*), web graph (*web-Google*), as well as the communication network (*wiki-Talk*) are part of SNAP and were first used in [3].

Queries. Following the methodology of [22], we generated three sets of 100 000 queries each: *positive*, *negative*, and *random*. Each set consists of random queries, which were generated by picking two vertices uniformly at random and filtering out negative or positive queries for the *positive* and *negative* query sets, respectively. The fourth query set, *mixed*, is a randomly shuffled union of all queries from *positive* and *negative* and hence contains 200 000 pairs of vertices. As the order of the queries within each set had an observable effect on the running time due to caching effects and memory layout, we randomly shuffled every query set five times and used a different permutation for each repetition of an experiment to ensure equal conditions for all algorithms.

5.1 Experimental Results

We ran `O'Reach` with $k = 16$ supportive vertices, picked from 1 200 candidates ($p = 75$, $h = 8$) and $d = 4$ extended topological orderings. We ran `IP` with the two configurations used also by the authors [36] and refer to the resulting algorithms as `IP(s)` (*sparse*, $h_{\texttt{IP}} = k_{\texttt{IP}} = 2$)

Table 3 Average query time per algorithm and query set.

Query set	O'R+ pBiBFS	PReaCH	O'R+ PReaCH	PPL	O'R+ PPL	IP(s)	O'R+ IP(s)	IP(d)	O'R+ IP(d)	BFL(s)	O'R+ BFL(s)	BFL(d)	O'R+ BFL(d)
random	3.523	1.596	1.483	0.271	**0.149**	12.865	11.193	9.778	8.516	6.645	5.073	5.063	3.361
mixed	19.964	6.351	6.102	0.352	**0.258**	80.572	73.625	60.352	56.433	32.456	28.496	22.002	17.541
positive	37.554	11.508	11.069	0.399	**0.345**	156.016	145.532	118.835	109.014	62.338	54.329	42.632	33.699
negative	2.382	1.188	1.154	0.260	**0.149**	5.342	5.059	3.727	3.793	2.496	2.506	1.345	1.358

and `IP(d)` (*dense*, $h_{\texttt{IP}} = k_{\texttt{IP}} = 5$). Similarly, we evaluated `BFL` [28] with configuration *sparse* as `BFL(s)` ($s_{\texttt{BFL}} = 64$) and *dense* as `BFL(d)` ($s_{\texttt{BFL}} = 160$), following the presets given by the authors.

Average query times. Table A.6 lists the average time per query for the query sets *negative* and *positive*. All missing values are due to a memory requirement of more than 32 GB (`TF`) and `Matrix` (256 GB). For each instance and query set, the running time of the fastest algorithm is printed in bold. If `Matrix` was fastest, also the running time of the second-best algorithm is highlighted. Besides `Matrix`, the table shows the running times of `PReaCH`, `PPL`, `IP(d)`, and `BFL(d)` alone as well as multiple versions for `O'Reach`: one with a pruned bidirectional BFS (`O'R+pBiBFS`) as fallback as well as one per competitor (`O'R+...`), where `O'Reach` was run without fallback and the queries left unanswered were fed to the competitor. Analogously, the running times for `IP(s)`, `BFL(s)`, and `TF` alone and as fallback for `O'Reach` are given in Table A.9.

Our results by and large *confirm* the performance comparison of `PReaCH`, `PPL`, and `TF` conducted by Merz and Sanders [22]. `PReaCH` was the fastest on three out of five Kronecker graphs for the negative query set, once beaten by `O'R+PReaCH` and `O'R+PPL` each, whereas `PPL` and `O'R+PPL` dominated all others on the positive query set in this class as well as on three of the five random graphs, while `O'R+TF` was slightly faster on the other two. `PReaCH` was also the dominating approach on the small real sparse and SNAP instances in the aforementioned study [22]. By contrast, it was *outperformed* on these classes here by `O'Reach` with almost any fallback on all instances for the positive query set, and by either `IP(d)` or `BFL(s)` on almost all instances for the negative query set. On the Delaunay and large real instances, `BFL(s)` often was the fastest algorithm on the set of negative queries. The results also reveal that `BFL` and in particular `IP` have a weak spot in answering positive queries. *On average over all instances*, `O'R+PPL` had the *fastest average query time* both for *negative* and *positive* queries.

Notably, `Matrix` was *outperformed* quite often, especially for queries in the set *negative*, which correlates with the fact that a large portion of these queries could be answered by constant-time observations (see also the detailed analysis of observation effectiveness below) and is due to its larger memory footprint. Across all instances and seeds, more than 95 % of all queries in this set could be answered by `O'Reach` directly. On the set *positive*, the average query time for `Matrix` was in almost all cases less than on the *negative* query set, which is explained by the small reachability of the instances and a resulting higher spatial locality and better cacheability of the few and naturally clustered one-entries in the matrix. Consequently, this effect was distinctly reduced for the *mixed* query set, as shown in Table A.7.

There are some instances where `O'Reach` had a fallback rate of over 90 % for the *positive* query set, e.g., on *cit-Patents*, which is clearly reflected in the running time. Except for `PPL`, all algorithms had difficulties with positive queries on this instance. Conversely, the fallback rate on all *uniprotenc_** instances and *citeseer.scc*, e.g., was 0 %. On average across all instances and seeds, `O'Reach` could answer over 70 % of all *positive* queries by constant-time observations.

Table 4 Mean speedups with `O'Reach` plus fallback over pure fallback algorithm. Values greater 1.00 are highlighted.

	negative				positive				random				mixed			
Instance	PReaCH	PPL	IP(d)	BFL(d)	PReaCH	PPL	IP(d)	BFL(d)	PReaCH	PPL	IP(d)	BFL(d)	PReaCH	PPL	IP(d)	BFL(d)
Geometric Mean	**1.10**	**2.22**	0.92	**1.06**	**1.33**	**1.90**	**3.98**	**3.14**	**1.29**	**2.53**	**1.26**	**2.40**	**1.29**	**2.04**	**2.77**	**2.31**
Ratio Runtime Avgs	**1.03**	**1.75**	0.98	0.99	**1.04**	**1.16**	**1.09**	**1.27**	**1.08**	**1.82**	**1.15**	**1.51**	**1.04**	**1.36**	**1.07**	**1.25**
Average	**1.13**	**2.32**	0.98	**1.35**	**1.41**	**2.25**	**5.87**	**6.25**	**1.33**	**2.69**	**1.41**	**8.22**	**1.33**	**2.23**	**3.37**	**3.63**

The results on the query sets *random* and *mixed* are similar and listed in Table A.7 and Table A.10. Once again, `O'R+PPL` showed the *fastest query time on average across all instances* for both query sets. As the reachability in a DAG is low in general (see also Theorem 1) and particularly in the benchmark instances, the average query times for *random* resemble those for *negative*. On the other hand, the results for the *mixed* query set are more similar to those for the *positive* query set, as the relative differences in performance among the algorithms are more pronounced there. Table 3 compactly shows the average query time over all instances for each query set. Only `PPL` and `O'R+PPL` achieved an average query time of less than 1 µs (and even less than 0.35 µs).

Speedups by `O'Reach`. We next investigate the relative speedup of `O'Reach` with different fallback solutions over running only the fallback algorithms. Table A.8 lists the ratios of the average query time of each competitor algorithm run standalone divided by the average query time of `O'Reach` plus that algorithm as fallback, for all four query sets. A compact version is also given in Table 4. In the large majority of cases, using `O'Reach` as a preprocessor resulted in a speedup, except in case of *negative* or *random* queries for `BFL` and partially `IP` on the large real instances as well as for `PReaCH` and partially again `IP` on the small real sparse and SNAP instances. The largest speedup of around 105 could be achieved for `BFL` on *kegg* for random queries. The mean speedup (geometric) is at least 1.29 for all fallback algorithms on the query sets *positive*, *random*, and *mixed*, where the maximum was reached for `IP(s)` on *positive* queries with a factor of 4.21. Only for purely *negative* queries, `IP(d)` and `BFL(s)` were a bit faster alone in the mean values. *In summary*, given that the algorithms are often already faster than single memory lookups, the speedups achieved by `O'Reach` are quite high.

Initialization Times (Table 2, right). On all graphs, `BFL(s)` had the fastest initialization time, followed by `BFL(d)` and `PReaCH`. For `O'Reach`, the overhead of computing the comparatively large out- and in-reachabilities of all 1 200 candidates for $k = 16$ supportive vertices is clearly reflected in the running time on denser instances and can be reduced greatly if lower parameters are chosen, albeit at the expense of a slightly reduced query performance, e.g., for $k = 8$. `PPL` often consumed a lot of time in this step, especially on denser instances, with a maximum of 2.6 h on *randn20-23*.

Based on the average query time per instance, the *minimum number of random queries necessary to amortize* the additional investment in initialization time if `O'Reach` is run as preprocessor is between 9.6 thousand (`O'R+BFL(d)`) and 499 thousand (`O'R+PReaCH`). Counting cases where `O'Reach` could not achieve a speedup in the average query time as infinity, the *median number of random queries* required for amortization is between 2.5 million (`O'R+BFL(d)`) and 101 million (`O'R+IP(d)`). For the on average fastest algorithm, `O'R+PPL`, the initialization cost is recovered after 210 thousand (*nasa*) to 6.15 billion (*kron_logn21*) *random* queries, which equals about 0.77 % (*nasa*) and 0.14 % (*kron_logn21*) of all vertex pairs, respectively.

Effectiveness of Observations. We collected a vast amount of statistical data to perform an analysis of the effectiveness of the different observations used in `O'Reach`. To make the analysis more robust, we increased the number of seeds to 25 here.

First, we look only at *fast queries*, i.e., those queries that could be answered without a fallback. Across all query sets, the *most effective* observation was the negative basic observation on topological orderings, (B4), which answered around 30 % of all fast queries. As the average reachability in the *random* query set is very low, negative queries predominate in the overall picture. It thus does not come as a surprise that the most effective observation is a negative one. On the *negative* query set, (B4) could answer 45 % of all fast queries. After lowering the number of topological orderings to $d = 2$, (B4) was still the most effective and could answer 23 % of all fast queries and 33 % of those in the *negative* query set. The negative observations second to (B4) in effectiveness were those looking at the forward and backward topological levels, Observation (B5) and (B6), which could answer around 15 % each on the *negative* query set and around 10 % of all fast queries. Note that we increased the counter for *all* observations that could answer a query for this analysis, not just the first in order, which is why there may be overlaps. The observations using the max and min indices of extended topological orderings, (T2) and (T5), could answer 9 % and 6 % of the fast queries in the *negative* query set, and the observations based on supportive vertices, (S2) and (S3), around 3 % each. Reducing the number of topological orderings to $d = 2$ decreased the effectiveness of (T2) and (T5) to around 5 %.

The *most effective positive observation* and the second-best among all query sets, was the supportive-vertices-based Observation (S1), which could answer almost 16 % of all fast queries and almost 55 % in the *positive* query set. Follow-up observations were the ones using high and low indices, (T1) and (T4), with 18 % and 16 % effectiveness for the *positive* query set. The remaining two, (T2) and (T5), could answer 6 % and 4 % in this set. Reducing the number of topological orderings to $d = 2$ led to a slight deterioration in case of (T1) and (T4) to 14 %, and to 5 % and 3 % in case of (T2) and (T5), each with respect to the *positive* query set.

Among all fast queries that could be answered by *only one* observation, the most effective observation was the positive supportive-vertices-based Observation (S1) with over 40 % for all query sets and 68 % for the *positive* query set, followed by the negative basic observation using topological orderings, (B4), with a bit over 20 % for all query sets and 52 % for the *negative* query set.

Looking now at the entire query sets, our statistics show that 95 % of all *queries could be answered via an observation* on the *negative* set. In 70 % of all cases, (B5) in the second test, which uses topological forward levels, could already answer the query. In further 16 % of all cases, the observation based on topological backward levels, (B6), was successful. On the *positive* query set, the fallback rate was 28 % and hence higher than on the *negative* query set. 52 % of all queries in this set could be answered by the supportive-vertices-based observation (S1), and the high and low indices of extended topological orderings (T1) and (T4) were responsible for another 7 % each. Observe that here, the first observation in the order that can answer a query "wins the point", i.e., there are no overlaps in the reported effectiveness.

Memory Consumption. Table 5 lists the memory each algorithm used for their *reachability index*. As `O'Reach` was configured with $k = 16$ and $d = 4$, its index size is 64n Byte. Consequently, the reachability indices of `O'Reach`, `PReaCH`, `PPL`, `IP`, `BFL`, and, with one exception for `TF`, fit in the L3 cache of 6 MB for all small real instances. For `Matrix`, this

Table 5 Real index size in memory (in MB).

Instance	`O'Reach`	`PReaCH`	`PPL`	`TF`	`IP(s)`	`IP(d)`	`BFL(s)`	`BFL(d)`	`Matrix`
kron_logn12	0.3	0.2	**0.1**	19.2	**0.1**	0.2	**0.1**	0.2	2.0
kron_logn16	4.0	3.5	1.5	0.0	1.5	3.1	**1.3**	2.3	512.0
kron_logn17	8.0	7.0	3.0	0.0	2.9	6.1	**2.5**	4.6	2 047.9
kron_logn20	64.0	56.0	25.1	0.0	22.1	44.8	**18.9**	34.1	131 070
kron_logn21	128.0	112.0	50.4	0.0	43.5	87.3	**37.1**	66.4	0.0
randn20-21	64.0	56.0	24.2	64.8	**18.0**	31.5	20.6	38.7	131 070
randn20-22	64.0	56.0	136.8	482.3	**19.0**	37.6	22.3	43.3	131 070
randn20-23	64.0	56.0	4 380.3	0.0	**19.5**	40.8	23.1	45.6	131 070
randn23-24	512.0	448.0	193.7	518.2	**144.3**	252.3	164.5	309.4	0.0
randn23-25	512.0	448.0	1 073.3	3 844.1	**152.0**	300.7	178.0	346.0	0.0
delaunay_n15	2.0	1.7	0.8	4.7	**0.6**	1.2	0.7	1.4	128.0
delaunay_n20	64.0	56.0	33.0	126.7	**19.1**	38.1	22.5	43.9	131 070
delaunay_n22	256.0	224.0	135.0	497.9	**76.6**	152.5	90.0	175.8	0.0
citeseer.scc	42.4	37.1	7.1	28.3	9.4	11.3	**9.2**	13.7	57 406.5
citeseerx	399.2	349.3	120.9	1 773.0	111.8	151.0	**107.6**	185.1	0.0
cit-Patents	230.4	201.6	659.2	780.0	72.9	138.0	**71.7**	132.9	0.0
go_uniprot	425.3	372.1	261.0	680.2	**106.4**	184.7	113.1	193.1	0.0
uniprotenc_22m	97.4	85.2	18.5	67.2	**24.5**	24.7	26.1	44.8	0.0
uniprotenc_100m	981.9	859.2	197.2	690.4	**251.2**	269.1	270.8	471.9	0.0
uniprotenc_150m	1 528.2	1 337.1	318.5	1 087.0	**395.0**	439.6	428.5	753.8	0.0
go_sub	0.4	0.4	0.2	0.4	**0.1**	0.2	**0.1**	0.3	5.5
pubmed_sub	0.5	0.5	0.3	1.1	**0.1**	0.2	0.2	0.3	9.7
yago_sub	0.4	0.4	0.2	0.5	**0.1**	0.2	**0.1**	0.2	5.3
citeseer_sub	0.7	0.6	0.3	1.2	**0.2**	0.3	**0.2**	0.4	13.7
arXiv	0.4	0.7	0.3	14.9	**0.1**	0.3	**0.1**	0.2	4.3
amaze	0.2	0.2	0.0	0.2	**0.1**	**0.1**	**0.1**	**0.1**	1.6
kegg	0.2	0.2	**0.1**	0.2	**0.1**	**0.1**	**0.1**	**0.1**	1.6
nasa	0.3	0.3	**0.1**	0.3	**0.1**	0.2	**0.1**	0.2	3.7
xmark	0.4	0.3	0.2	0.4	**0.1**	0.2	**0.1**	0.2	4.4
vchocyc	0.6	0.5	**0.2**	0.7	**0.2**	0.3	**0.2**	0.3	10.7
mtbrv	0.6	0.5	**0.2**	0.4	**0.2**	0.3	**0.2**	0.3	11.0
anthra	0.8	0.7	**0.2**	0.8	**0.2**	0.4	**0.2**	0.4	18.6
ecoo	0.8	0.7	**0.2**	0.9	**0.2**	0.4	**0.2**	0.4	19.0
agrocyc	0.8	0.7	**0.2**	0.9	**0.2**	0.4	**0.2**	0.4	19.2
human	2.4	2.1	**0.6**	2.1	0.7	1.2	**0.6**	1.1	179.6
p2p-Gnutella31	3.0	2.6	0.7	2.1	0.9	1.5	**0.8**	1.4	279.7
email-EuAll	14.1	12.3	2.6	9.7	**3.7**	5.8	**3.7**	6.4	6 349.8
web-Google	22.7	19.9	5.4	16.7	7.0	11.2	**6.5**	11.5	16 475.5
soc-LiveJournal1	59.2	51.8	13.0	41.0	19.1	31.8	**15.9**	27.2	112 225
wiki-Talk	139.3	121.9	26.2	95.9	52.0	103.5	**37.1**	63.3	0.0

was only the case for the four smallest instances from the small real sparse set, three of the small real dense ones, and the smallest Kronecker graph, which is clearly reflected in its average query time for the *negative*, *random*, and, to a slightly lesser extent, *mixed* query sets. Whereas for `O'Reach`, `PReaCH`, and `Matrix`, the index size depends solely on the number of vertices, `IP`, `BFL`, `PPL` and `TF` consumed more memory the larger the density $\frac{m}{n}$. `IP(s)` usually was the most space-efficient and never used more than 395 MB, followed by `BFL(s)` (429 MB), `IP(d)` (440 MB), `BFL(d)` (754 MB), `PReaCH` (1.3 GB), `O'Reach` (1.5 GB), and `PPL` (4.4 GB). All these algorithms are hence suitable to handle graphs with several millions of vertices even on hardware with relatively little memory (with respect to current standards). `TF` used up to 3.8 GB (*randn23-25*), but required even more than 64 GB at least during initialization on all instances where the data is missing in the table.

6 Conclusion

In this paper, we revisited existing techniques for the static reachability problem and combined them with new approaches to support a large portion of *reachability queries* in constant time using a linear-sized *reachability index*. Our extensive experimental evaluation shows that in almost all scenarios, combining any of the existing algorithms with our new techniques implemented in `O'Reach` can speed up the query time by several factors. In particular *supportive vertices* have proven to be effective to answer positive queries quickly. As a further plus, `O'Reach` is flexible: memory usage, initialization time, and expected query time can be influenced directly by three parameters, which allow to trade space for time or initialization time for query time. Moreover, our study demonstrates that, due to cache effects, a high investment in space does not necessarily pay off: *Reachability queries* can often be answered even significantly faster than single memory accesses in a precomputed full reachability matrix.

The on average fastest algorithm across all instances and types of queries was a combination of `O'Reach` and `PPL` with an average query time of less than 0.35 µs. As the initialization time of `PPL` is relatively high, we also recommend `O'Reach` combined with `PReaCH` as a less expensive alternative solution with respect to initialization time and partially also memory, which still achieved an average query time of at most 11.1 µs on all query sets.

References

1 D. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*. Springer, 2014.

2 Yangjun Chen and Yibin Chen. An efficient algorithm for answering graph reachability queries. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 893–902. IEEE Computer Society, 2008. doi:10.1109/ICDE.2008.4497498.

3 James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. TF-label: a topological-folding labeling scheme for reachability querying in a large graph. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 193–204. ACM, 2013. doi:10.1145/2463676.2465286.

4 Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. Fast computation of reachability labeling for large graphs. In Yannis E. Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Michael Hatzopoulos, Klemens Böhm, Alfons Kemper, Torsten Grust, and Christian Böhm, editors, *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, volume 3896 of *Lecture Notes in Computer Science*, pages 961–979. Springer, 2006. doi:10.1007/11687238_56.

5 Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003. doi:10.1137/S0097539702403098.

6 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms, 2009.

7 R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, June 1962. doi:10.1145/367766.368168.

8 Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018*, 2018.

9 Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

10 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

11 Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Faster Fully Dynamic Transitive Closure in Practice. In Simone Faro and Domenico Cantone, editors, *18th International Symposium on Experimental Algorithms (SEA 2020)*, volume 160 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SEA.2020.14.

12 Kathrin Hanauer, Christian Schulz, and Jonathan Trummer. O'Reach: Even faster reachability in static graphs. *CoRR*, abs/2008.10932, 2021. arXiv:2008.10932.

13 H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990. doi:10.1145/99935.99944.

14 Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Xu Yu. SCARAB: scaling reachability computation on large graphs. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 169–180. ACM, 2012. doi:10.1145/2213836.2213856.

15 Ruoming Jin, Ning Ruan, Yang Xiang, and Haixun Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Trans. Database Syst.*, 36(1):7:1–7:44, 2011. doi:10.1145/1929934.1929941.

16 Ruoming Jin and Guan Wang. Simple, fast, and scalable reachability oracle. *Proc. VLDB Endow.*, 6(14):1978–1989, 2013. doi:10.14778/2556549.2556578.

17 Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-hop: A high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 813–826, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1559845.1559930.

18 Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 595–608. ACM, 2008. doi:10.1145/1376616.1376677.

19 A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962. doi:10.1145/368996.369025.

20 F. Le Gall. Powers of tensors and fast matrix multiplication. In K. Nabeshima, K. Nagasaka, F. Winkler, and Á. Szántó, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303. ACM, 2014. doi:10.1145/2608628.2608664.

21 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

22 F. Merz and P. Sanders. Preach: A fast lightweight reachability index using pruning and contraction hierarchies. In A. S. Schulz and D. Wagner, editors, *European Symposium on Algorithms*, pages 701–712, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

23 Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

24 Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998.

25 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

26 Ralf Schenkel, Anja Theobald, and Gerhard Weikum. HOPI: an efficient connection index for complex XML document collections. In Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors, *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, volume 2992 of *Lecture Notes in Computer Science*, pages 237–255. Springer, 2004. doi:10.1007/978-3-540-24741-8_15.

27 B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 25–34, 2008.

28 Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. Reachability querying: Can it be even faster? *IEEE Trans. Knowl. Data Eng.*, 29(3):683–697, 2017. doi:10.1109/TKDE.2016.2631160.

29 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi:10.1137/0201010.

30 Robert Endre Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

31 Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 845–856. ACM, 2007. doi:10.1145/1247480.1247573.

32 Sebastiaan J. van Schaik and Oege de Moor. A memory efficient reachability data structure through bit vector compression. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 913–924. ACM, 2011. doi:10.1145/1989323.1989419.

33 Renê Rodrigues Veloso, Loïc Cerf, Wagner Meira, and Mohammed J. Zaki. Reachability queries in very large graphs: A fast refined online search approach. In *EDBT*, pages 511–522, 2014.

34 Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 75. IEEE Computer Society, 2006. doi:10.1109/ICDE.2006.53.

35 S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962. doi:10.1145/321105.321107.

36 Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. Reachability querying: an independent permutation labeling approach. *VLDB J.*, 27(1):1–26, 2018. doi:10.1007/s00778-017-0468-3.

37 Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi, editors, *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pages 1601–1606. ACM, 2013. doi:10.1145/2505515.2505724.

38 Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. Grail: Scalable reachability index for large graphs. *Proc. VLDB Endow.*, 3(1–2):276–284, 2010. doi:10.14778/1920841.1920879.

39 Hilmi Yıldırım, Vineet Chaoji, and Mohammed J Zaki. GRAIL: a scalable index for reachability queries in very large graphs. *The VLDB Journal*, 21(4):509–534, 2012.

40 Jeffrey Xu Yu and Jiefeng Cheng. Graph reachability queries: A survey. In Charu C. Aggarwal and Haixun Wang, editors, *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, pages 181–215. Springer, 2010. doi:10.1007/978-1-4419-6045-0_6.

A Appendix

Table A.6 Average query times in µs for 100 000 negative (left) and positive queries (right). Highlighted results are the overall best/second-best after `Matrix` per query set over *all* tested algorithms.

										← negative	positive→									
	O'R+		O'R+		O'R+		O'R+		O'R+		O'R+		O'R+		O'R+		O'R+		O'R+	
Instance	pBiBFS	PReaCH	PReaCH	PPL	PPL	IP(d)	IP(d)	BFL(d)	BFL(d)	Matrix	pBiBFS	PReaCH	PReaCH	PPL	PPL	IP(d)	IP(d)	BFL(d)	BFL(d)	Matrix
kron_logn12	0.031	0.020	**0.017**	0.030	0.018	0.028	0.028	0.097	0.046	**0.017**	0.347	0.361	0.251	0.035	**0.032**	2.213	0.824	3.278	0.984	**0.014**
kron_logn16	0.094	**0.057**	0.069	0.109	0.075	0.078	0.113	0.161	0.153	0.533	3.246	3.467	2.637	0.115	**0.106**	25.690	10.530	25.007	9.725	0.262
kron_logn17	0.122	**0.072**	0.078	0.135	0.086	0.095	0.131	0.117	0.117	1.111	2.365	2.537	0.692	0.152	**0.100**	20.548	4.597	9.929	1.734	0.881
kron_logn20	0.184	**0.117**	0.128	0.221	0.119	0.167	0.201	0.325	0.353	2.413	46.186	25.092	23.331	0.274	**0.265**	342.887	163.902	373.848	162.405	1.778
kron_logn21	0.224	0.146	0.162	0.250	**0.139**	0.207	0.257	0.237	0.286		67.184	15.416	5.998	**0.313**	0.355	306.828	193.212	203.695	113.502	
randn20-21	0.255	0.342	0.234	0.278	0.154	0.218	0.191	0.056	0.122	1.692	1.298	1.044	0.858	0.482	0.404	2.513	1.792	0.856	0.693	0.844
randn20-22	3.016	2.473	2.298	0.424	0.335	2.569	2.519	0.357	0.429	1.056	35.747	21.873	21.832	**1.153**	1.278	53.591	53.431	9.369	9.345	**0.361**
randn20-23	82.960	36.193	36.139	3.092	**2.449**	139.544	142.731	51.125	51.046	4.993	870.787	247.700	246.218	**4.625**	4.775	3 524.734	3 462.790	971.807	944.526	**1.822**
randn23-24	0.356	0.494	0.426	0.402	0.216	0.310	0.264	0.055	0.168		1.747	1.366	1.169	0.680	0.557	3.272	2.345	1.074	0.892	
randn23-25	4.130	3.244	3.011	0.546	0.426	3.237	3.143	0.409	0.529		51.713	29.098	29.108	**1.212**	1.379	70.327	70.425	11.781	11.825	
delaunay_n15	0.125	0.172	0.076	0.118	0.055	0.189	0.088	0.053	0.045	0.063	1.227	0.150	0.156	**0.102**	0.108	5.207	2.693	0.762	0.569	**0.050**
delaunay_n20	0.237	0.350	0.187	0.288	0.142	0.355	0.183	0.055	0.124	0.924	2.984	**0.334**	0.397	0.417	0.419	9.180	6.407	2.173	1.817	0.505
delaunay_n22	0.280	0.418	0.274	0.363	0.177	0.415	0.221	0.054	0.154		3.354	**0.425**	0.590	0.560	0.561	9.249	7.219	2.945	2.532	
citeseer.scc	0.056	0.075	0.056	0.229	0.056	0.050	0.056	0.043	0.056	1.312	0.113	**0.089**	0.112	0.303	0.112	0.384	0.112	0.179	0.112	0.653
citeseerx	0.211	0.212	0.214	0.358	0.141	0.160	0.172	**0.058**	0.141		0.544	0.248	0.222	0.296	**0.085**	2.511	0.426	1.686	0.291	
cit-Patents	3.965	2.732	2.562	0.526	**0.329**	3.915	3.797	0.658	0.737		238.913	118.427	117.240	**1.961**	2.123	473.083	482.668	118.097	116.791	
go_uniprot	0.098	0.121	0.099	0.385	0.098	0.068	0.098	0.042	0.098		208.494	1.026	0.902	0.534	**0.435**	1.054	0.712	0.688	0.519	
uniprotenc_22m	0.067	0.068	0.066	0.254	0.066	0.045	0.066	0.043	0.066		**0.072**	0.076	**0.072**	0.274	**0.072**	0.334	**0.072**	0.196	**0.072**	
uniprotenc_100m	0.130	0.163	0.131	0.410	0.131	0.098	0.131	0.043	0.131		0.118	**0.108**	0.118	0.452	0.118	0.504	0.118	0.233	0.118	
uniprotenc_150m	0.152	0.206	0.153	0.444	0.153	0.116	0.153	0.044	0.153		0.139	**0.121**	0.139	0.509	0.139	0.565	0.139	0.239	0.139	
go_sub	0.033	0.046	0.028	0.058	0.026	0.050	0.031	0.057	**0.024**	0.025	0.198	0.139	0.088	0.092	**0.055**	2.447	0.448	0.355	0.154	**0.012**
pubmed_sub	0.078	0.076	0.066	0.068	0.044	0.058	0.057	0.061	**0.039**	**0.029**	0.546	0.491	0.340	0.090	**0.073**	1.441	0.577	0.922	0.399	**0.019**
yago_sub	0.025	0.030	0.023	0.058	0.023	0.022	0.023	0.048	0.022	0.026	0.146	0.097	0.074	0.086	**0.057**	0.342	0.137	0.225	0.102	**0.020**
citeseer_sub	0.083	0.089	0.059	0.071	0.038	0.072	0.054	0.055	**0.029**	0.032	0.580	0.285	0.223	0.095	**0.087**	1.187	0.642	0.574	0.318	**0.026**
arXiv	0.247	0.258	0.223	0.076	**0.047**	0.299	0.242	0.130	0.091	**0.024**	1.209	1.216	0.637	**0.046**	0.047	6.445	2.790	3.464	1.657	**0.018**
amaze	0.012	0.014	0.013	0.030	0.013	**0.011**	0.013	0.048	0.013	0.016	0.010	0.015	**0.009**	0.031	**0.009**	0.089	**0.009**	0.102	**0.009**	**0.009**
kegg	0.014	0.015	0.015	0.033	0.015	0.014	0.015	0.053	0.015	0.032	0.010	0.016	**0.009**	0.031	**0.009**	0.094	**0.009**	0.102	**0.009**	0.010
nasa	**0.026**	0.031	0.029	0.048	0.027	0.032	0.031	0.054	**0.026**	**0.022**	0.061	0.058	0.044	0.044	**0.022**	1.627	0.148	0.351	0.044	**0.008**
xmark	0.031	0.032	0.027	0.052	0.026	0.042	0.031	0.055	**0.023**	0.024	0.036	0.049	0.026	0.032	**0.014**	0.432	0.045	2.163	0.021	**0.008**
vchocyc	0.016	0.016	0.017	0.050	0.017	**0.013**	0.017	0.047	0.017	0.029	0.015	0.024	**0.014**	0.039	**0.014**	0.241	0.015	0.096	0.015	**0.007**
mtbrv	0.017	0.016	0.018	0.050	0.018	**0.013**	0.018	0.047	0.018	0.029	0.017	0.025	0.017	0.039	**0.016**	0.233	0.019	0.105	0.017	**0.006**
anthra	0.017	0.018	0.019	0.054	0.019	**0.013**	0.019	0.047	0.020	0.033	**0.014**	0.025	**0.014**	0.043	**0.014**	0.283	0.015	0.087	**0.014**	**0.005**
ecoo	0.017	0.017	0.019	0.053	0.019	**0.013**	0.019	0.047	0.019	0.055	0.015	0.027	**0.014**	0.040	**0.014**	0.266	0.015	0.111	**0.014**	**0.006**
agrocyc	0.018	0.017	0.021	0.054	0.021	**0.013**	0.021	0.046	0.021	0.033	**0.014**	0.027	**0.014**	0.042	**0.014**	0.249	0.015	0.139	0.015	**0.006**
human	0.025	0.025	0.033	0.097	0.033	**0.015**	0.033	0.045	0.033	0.072	**0.022**	0.036	**0.022**	0.083	**0.022**	0.281	**0.022**	0.118	**0.022**	**0.006**
p2p-Gnutella31	0.031	0.030	0.037	0.111	0.037	**0.017**	0.037	0.046	0.036	0.100	**0.026**	0.037	**0.026**	0.070	**0.026**	0.191	**0.026**	0.274	**0.026**	**0.023**
email-EuAll	0.054	0.062	0.061	0.161	0.062	0.055	0.062	0.045	0.061	5.267	**0.042**	0.058	**0.042**	0.204	**0.042**	0.342	**0.042**	0.197	**0.042**	2.858
web-Google	0.077	0.079	0.076	0.175	0.075	0.081	0.076	0.052	0.070	1.400	0.049	0.068	**0.048**	0.190	**0.048**	0.458	**0.048**	0.237	**0.048**	1.405
soc-LiveJournal1	0.065	0.062	0.070	0.192	0.072	0.057	0.072	0.046	0.069	3.785	**0.058**	0.077	**0.058**	0.240	**0.058**	0.446	**0.058**	0.201	**0.058**	1.806
wiki-Talk	0.075	0.073	0.083	0.269	0.083	0.049	0.083	0.044	0.083		0.058	0.077	**0.057**	0.330	**0.057**	0.356	**0.057**	0.172	**0.057**	
Min	0.012	0.014	0.013	0.030	0.013	**0.011**	0.013	0.042	0.013		0.010	0.015	**0.009**	0.031	**0.009**	0.089	**0.009**	0.087	**0.009**	
Average	2.382	1.188	1.154	0.260	**0.149**	3.727	3.793	1.345	1.358		37.554	11.508	11.069	0.399	**0.345**	118.835	109.014	42.632	33.699	
Max	82.960	36.193	36.139	3.092	**2.449**	139.544	142.731	51.125	51.046		870.787	247.700	246.218	**4.625**	4.775	3 524.734	3 462.790	971.807	944.526	

Table A.7 Average query times in µs for 100 000 random (left) and 200 000 mixed queries (right). Highlighted results are the overall best/second-best after `Matrix` per query set over *all* tested algorithms.

	← random										mixed→									
	O'R+		O'R+		O'R+		O'R+		O'R+		O'R+		O'R+		O'R+		O'R+		O'R+	
Instance	pBiBFS	PReaCH	PReaCH	PPL	PPL	IP(d)	IP(d)	BFL(d)	BFL(d)	Matrix	pBiBFS	PReaCH	PReaCH	PPL	PPL	IP(d)	IP(d)	BFL(d)	BFL(d)	Matrix
kron_logn12	0.120	0.120	0.084	0.045	**0.028**	0.635	0.251	3.014	0.300	**0.019**	0.194	0.197	0.139	0.039	**0.029**	1.121	0.433	1.694	0.516	**0.016**
kron_logn16	0.776	0.821	0.628	0.155	**0.081**	5.463	2.297	7.194	2.215	1.006	1.682	1.788	1.351	0.143	**0.088**	12.943	5.317	12.614	4.932	0.110
kron_logn17	0.606	0.607	0.237	0.180	**0.092**	4.040	1.001	3.617	0.482	1.326	1.258	1.333	0.389	0.176	**0.097**	10.361	2.362	5.061	0.932	0.173
kron_logn20	7.327	4.273	3.998	0.276	**0.150**	53.853	26.239	59.319	26.638	2.430	23.183	12.651	11.753	0.285	**0.207**	171.584	82.016	187.036	81.384	0.393
kron_logn21	10.364	2.458	1.064	0.311	**0.179**	44.566	27.592	33.318	17.089		33.677	7.806	3.085	0.323	**0.266**	153.798	97.018	102.013	56.982	
randn20-21	0.260	0.419	0.306	0.284	0.152	0.224	0.189	0.056	0.116	1.771	0.795	0.710	0.560	0.391	0.291	1.387	1.005	0.466	0.420	0.372
randn20-22	3.007	2.552	2.363	0.431	0.339	2.494	2.445	0.355	0.420	1.060	19.356	12.227	12.090	0.806	0.824	28.163	28.046	4.882	4.904	**0.338**
randn20-23	110.867	42.880	42.520	2.963	**2.405**	278.295	279.266	89.953	87.178	5.242	477.009	141.865	140.705	3.977	**3.715**	1 794.925	1 805.922	511.440	497.870	**0.355**
randn23-24	0.355	1.033	0.784	0.413	0.224	0.310	0.266	0.056	0.171		1.073	0.928	0.739	0.553	0.401	1.819	1.314	0.580	0.547	
randn23-25	4.160	3.916	3.671	0.555	0.433	3.323	3.230	0.419	0.540		27.927	16.212	16.092	**0.901**	0.920	36.916	36.914	6.130	6.175	
delaunay_n15	0.133	0.175	0.077	0.133	0.054	0.229	0.102	0.091	**0.045**	0.064	0.685	0.166	0.120	0.129	**0.080**	2.846	1.406	0.420	0.307	**0.056**
delaunay_n20	0.236	0.408	0.226	0.295	0.139	0.363	0.176	0.058	0.118	0.992	1.639	0.361	0.301	0.368	**0.292**	4.800	3.318	1.128	0.982	0.384
delaunay_n22	0.280	0.650	0.315	0.373	0.181	0.417	0.220	0.056	0.154		1.856	0.430	**0.371**	0.478	0.377	4.900	3.731	1.502	1.354	
citeseer.scc	0.057	0.077	0.057	0.235	0.057	0.050	0.057	0.043	0.057	1.323	0.105	**0.095**	0.106	0.294	0.106	0.243	0.106	0.124	0.106	0.385
citeseerx	0.208	0.253	0.238	0.369	0.138	0.164	0.163	0.079	0.134		0.397	0.204	0.179	0.386	**0.121**	1.407	0.301	0.890	0.227	
cit-Patents	4.107	3.029	2.857	0.532	**0.329**	3.998	3.889	0.680	0.743		121.444	60.488	59.975	1.301	**1.255**	236.166	240.466	59.523	58.810	
go_uniprot	0.108	0.123	0.103	0.394	0.101	0.069	0.101	0.042	0.101		103.633	0.411	0.348	0.485	**0.288**	0.610	0.435	0.378	0.337	
uniprotenc_22m	0.067	0.068	0.068	0.260	0.068	0.045	0.068	0.043	0.068		0.093	0.099	**0.092**	0.277	**0.092**	0.213	**0.092**	0.130	**0.092**	
uniprotenc_100m	0.132	0.165	0.134	0.419	0.134	0.098	0.134	0.043	0.134		0.149	0.152	0.148	0.450	0.148	0.342	0.148	0.149	0.148	
uniprotenc_150m	0.154	0.210	0.156	0.454	0.156	0.116	0.156	0.044	0.156		0.170	0.172	0.170	0.499	0.170	0.383	0.170	0.152	0.170	
go_sub	0.034	0.046	0.025	0.064	0.023	0.054	0.030	0.076	**0.022**	0.027	0.122	0.099	0.060	0.077	**0.041**	1.259	0.247	0.212	0.091	**0.021**
pubmed_sub	0.083	0.083	0.061	0.077	0.038	0.070	0.054	0.118	**0.033**	**0.030**	0.318	0.296	0.205	0.089	**0.059**	0.740	0.316	0.497	0.220	**0.026**
yago_sub	0.025	0.031	0.018	0.064	0.018	0.022	0.018	0.061	**0.016**	0.027	0.092	0.070	0.051	0.076	**0.041**	0.188	0.081	0.147	0.064	**0.023**
citeseer_sub	0.085	0.092	0.060	0.083	0.041	0.076	0.056	0.088	**0.030**	0.033	0.338	0.196	0.146	0.095	**0.066**	0.634	0.347	0.320	0.179	**0.029**
arXiv	0.379	0.377	0.253	0.085	**0.049**	1.215	0.622	1.774	0.322	**0.026**	0.743	0.752	0.433	0.068	**0.049**	3.424	1.606	1.795	0.877	**0.023**
amaze	0.015	0.017	**0.014**	0.041	**0.014**	0.030	**0.014**	1.262	**0.014**	0.019	**0.015**	0.020	**0.015**	0.039	**0.015**	0.057	**0.015**	0.086	**0.015**	**0.015**
kegg	**0.015**	0.017	**0.015**	0.043	**0.015**	0.035	**0.015**	1.542	**0.015**	0.018	0.016	0.020	**0.015**	0.039	**0.015**	0.060	**0.015**	0.086	**0.015**	**0.015**
nasa	0.026	0.030	0.023	0.054	0.021	0.041	0.024	0.094	**0.019**	0.024	0.048	0.051	0.037	0.056	**0.025**	0.838	0.088	0.209	0.036	**0.019**
xmark	0.029	0.030	0.023	0.059	0.023	0.049	0.029	0.188	**0.020**	0.025	0.037	0.046	0.029	0.053	**0.022**	0.239	0.040	1.118	0.024	**0.020**
vchocyc	0.017	0.016	0.017	0.058	0.016	**0.014**	0.016	0.059	0.016	0.031	0.020	0.025	**0.019**	0.056	**0.019**	0.136	**0.019**	0.080	0.020	0.023
mtbrv	0.016	0.016	0.017	0.058	0.016	**0.013**	0.016	0.060	0.016	0.031	0.021	0.026	**0.020**	0.056	**0.020**	0.132	0.021	0.085	**0.020**	0.024
anthra	0.017	0.019	0.017	0.064	0.017	**0.013**	0.017	0.054	0.017	0.035	0.020	0.027	**0.019**	0.061	0.020	0.161	0.020	0.075	0.020	0.025
ecoo	0.017	0.018	0.017	0.064	0.017	**0.013**	0.017	0.056	0.017	0.035	**0.020**	0.027	**0.020**	0.060	**0.020**	0.148	**0.020**	0.088	**0.020**	0.026
agrocyc	0.017	0.018	0.018	0.064	0.018	**0.014**	0.017	0.055	0.017	0.036	**0.020**	0.027	**0.020**	0.061	**0.020**	0.143	**0.020**	0.100	**0.020**	0.026
human	0.024	0.026	0.026	0.109	0.026	**0.015**	0.026	0.048	0.026	0.074	0.028	0.033	**0.027**	0.108	**0.027**	0.159	**0.027**	0.091	**0.027**	0.046
p2p-Gnutella31	0.030	0.034	0.031	0.123	0.030	**0.019**	0.030	0.104	0.030	0.102	0.033	0.037	**0.032**	0.126	**0.032**	0.116	0.033	0.173	**0.032**	0.072
email-EuAll	0.058	0.069	**0.056**	0.175	0.057	0.075	0.057	0.451	**0.056**	5.497	0.062	0.072	**0.060**	0.200	**0.060**	0.218	0.061	0.134	**0.060**	0.318
web-Google	0.081	0.094	0.079	0.204	0.079	0.147	0.079	1.187	**0.074**	1.461	0.075	0.084	0.071	0.224	0.072	0.292	0.073	0.157	**0.070**	0.262
soc-LiveJournal1	0.075	0.077	0.074	0.239	0.076	0.152	0.075	1.668	**0.073**	3.074	0.080	0.089	**0.077**	0.260	0.078	0.274	0.078	0.135	**0.077**	0.390
wiki-Talk	0.076	0.075	0.076	0.278	0.076	**0.054**	0.076	0.105	0.076		**0.087**	0.095	0.088	0.319	0.088	0.228	0.088	0.123	0.088	
Min	0.015	0.016	0.014	0.041	0.014	**0.013**	0.014	0.042	0.014		**0.015**	0.020	**0.015**	0.039	**0.015**	0.057	**0.015**	0.075	**0.015**	
Average	3.523	1.596	1.483	0.271	**0.149**	9.778	8.516	5.063	3.361		19.964	6.351	6.102	0.352	**0.258**	60.352	56.433	22.002	17.541	
Max	110.867	42.880	42.520	2.963	**2.405**	278.295	279.266	89.953	87.178		477.009	141.865	140.705	3.977	**3.715**	1 794.925	1 805.922	511.440	497.870	

Table A.8 Speedups with `O'Reach` plus fallback over pure fallback algorithm. Values greater 1.00 are highlighted.

	negative							positive							random							mixed						
Instance	PReaCH	PPL	TF	IP(s)	IP(d)	BFL(s)	BFL(d)	PReaCH	PPL	TF	IP(s)	IP(d)	BFL(s)	BFL(d)	PReaCH	PPL	TF	IP(s)	IP(d)	BFL(s)	BFL(d)	PReaCH	PPL	TF	IP(s)	IP(d)	BFL(s)	BFL(d)
kron_logn12	**1.16**	**1.65**	**2.99**	0.97	1.00	**1.87**	**2.11**	**1.43**	**1.11**	**2.30**	**2.45**	**2.69**	**3.13**	**3.33**	**1.43**	**1.58**	**2.58**	**2.34**	**2.53**	**9.86**	**10.06**	**1.42**	**1.32**	**2.39**	**2.41**	**2.59**	**3.06**	**3.28**
kron_logn16	0.83	**1.45**		0.68	0.69	0.99	**1.05**	**1.31**	**1.08**		**2.29**	**2.44**	**2.42**	**2.57**	**1.31**	**1.90**		**2.27**	**2.38**	**3.16**	**3.25**	**1.32**	**1.61**		**2.28**	**2.43**	**2.41**	**2.56**
kron_logn17	0.92	**1.57**		0.73	0.73	0.93	0.99	**3.67**	**1.52**		**4.34**	**4.47**	**5.14**	**5.72**	**2.57**	**1.94**		**3.98**	**4.04**	**6.92**	**7.50**	**3.42**	**1.82**		**4.27**	**4.39**	**4.88**	**5.43**
kron_logn20	0.92	**1.85**		0.84	0.83	0.90	0.92	**1.08**	**1.03**		**2.07**	**2.09**	**2.21**	**2.30**	**1.07**	**1.84**		**2.02**	**2.05**	**2.16**	**2.23**	**1.08**	**1.38**		**2.07**	**2.09**	**2.21**	**2.30**
kron_logn21	0.90	**1.80**		0.82	0.80	0.80	0.83	**2.57**	0.88		**1.65**	**1.59**	**1.75**	**1.79**	**2.31**	**1.74**		**1.67**	**1.62**	**1.90**	**1.95**	**2.53**	**1.22**		**1.65**	**1.59**	**1.75**	**1.79**
randn20-21	**1.46**	**1.81**	**1.92**	**1.43**	**1.14**	0.36	0.46	**1.22**	**1.19**	**1.38**	**1.56**	**1.40**	**1.22**	**1.23**	**1.37**	**1.87**	**2.00**	**1.44**	**1.19**	0.39	0.49	**1.27**	**1.34**	**1.53**	**1.37**	**1.38**	**1.08**	**1.11**
randn20-22	**1.08**	**1.27**	**1.50**	**1.03**	**1.02**	0.94	0.83	1.00	0.90	**1.15**	**1.01**	1.00	**1.01**	1.00	**1.08**	**1.27**	**1.52**	**1.05**	**1.02**	0.94	0.85	**1.01**	0.98	**1.23**	**1.02**	1.00	1.00	1.00
randn20-23	1.00	**1.26**		**1.05**	0.98	**1.01**	1.00	**1.01**	0.97		**1.01**	**1.02**	**1.02**	**1.03**	**1.01**	**1.23**		**1.03**	1.00	**1.02**	**1.03**	**1.01**	**1.07**		**1.04**	0.99	**1.01**	**1.03**
randn23-24	**1.16**	**1.87**	**2.08**	**1.38**	**1.17**	0.27	0.33	**1.17**	**1.22**	**1.43**	**1.44**	**1.40**	**1.19**	**1.20**	**1.32**	**1.84**	**2.06**	**1.47**	**1.17**	0.26	0.33	**1.26**	**1.38**	**1.62**	**1.41**	**1.38**	**1.03**	**1.06**
randn23-25	**1.08**	**1.28**	**1.53**	**1.04**	**1.03**	0.89	0.77	1.00	0.88	**1.13**	**1.01**	1.00	1.00	1.00	**1.07**	**1.28**	**1.53**	**1.28**	**1.03**	0.90	0.78	**1.01**	0.98	**1.23**	1.00	1.00	0.99	0.99
delaunay_n15	**2.24**	**2.16**	**2.71**	**2.80**	**2.14**	0.88	**1.19**	0.96	0.94	**1.34**	**2.14**	**1.93**	**1.34**	**1.34**	**2.26**	**2.46**	**3.02**	**2.75**	**2.24**	**1.73**	**2.02**	**1.39**	**1.62**	**1.83**	**2.12**	**2.02**	**1.33**	**1.37**
delaunay_n20	**1.87**	**2.03**	**2.59**	**2.63**	**1.94**	0.31	0.44	0.84	1.00	**1.34**	**1.46**	**1.43**	**1.20**	**1.20**	**1.81**	**2.13**	**2.70**	**2.78**	**2.06**	0.35	0.49	**1.20**	**1.26**	**1.62**	**1.52**	**1.45**	**1.14**	**1.15**
delaunay_n22	**1.52**	**2.05**	**2.68**	**2.51**	**1.88**	0.25	0.35	0.72	1.00	**1.29**	**1.29**	**1.28**	**1.17**	**1.16**	**2.06**	**2.06**	**2.66**	**2.53**	**1.89**	0.26	0.36	**1.16**	**1.27**	**1.61**	**1.45**	**1.31**	**1.11**	**1.11**
citeseer.scc	**1.35**	**4.11**	0.42	0.93	0.89	0.62	0.78	0.79	**2.70**	**2.68**	**2.85**	**3.43**	**1.37**	**1.60**	**1.35**	**4.11**	0.51	0.90	0.87	0.60	0.75	0.89	**2.77**	**2.02**	**2.00**	**2.29**	0.98	**1.16**
citeseerx	0.99	**2.55**	**2.97**	1.00	0.93	0.41	0.41	**1.12**	**3.47**	**16.99**	**4.12**	**5.89**	**4.17**	**5.79**	**1.07**	**2.67**	**3.01**	**1.06**	**1.01**	0.54	0.59	**1.14**	**3.20**	**9.66**	**3.51**	**4.67**	**3.18**	**3.93**
cit-Patents	**1.07**	**1.60**	**2.02**	**1.03**	**1.03**	0.97	0.89	**1.01**	0.92	**1.16**	0.99	0.98	1.00	**1.01**	**1.06**	**1.62**	**2.03**	**1.06**	**1.03**	0.98	0.92	**1.01**	**1.04**	**1.26**	**1.03**	0.98	1.00	**1.01**
go_uniprot	**1.22**	**3.92**	**1.08**	0.70	0.70	0.34	0.43	**1.14**	**1.23**	**1.38**	**1.45**	**1.48**	**1.26**	**1.33**	**1.20**	**3.89**	**1.07**	0.68	0.68	0.33	0.42	**1.18**	**1.68**	**1.39**	**1.36**	**1.40**	**1.06**	**1.12**
uniprotenc_22m	**1.02**	**3.82**	**1.20**	0.68	0.68	0.50	0.65	**1.05**	**3.80**	**2.49**	**4.60**	**4.63**	**2.41**	**2.72**	1.00	**3.83**	**1.20**	0.67	0.67	0.49	0.63	**1.08**	**3.02**	**1.78**	**2.33**	**2.33**	**1.14**	**1.42**
uniprotenc_100m	**1.25**	**3.14**	**1.43**	0.76	0.75	0.25	0.33	0.92	**3.83**	**2.95**	**4.21**	**4.27**	**1.70**	**1.97**	**1.24**	**3.14**	**1.43**	0.74	0.73	0.25	0.32	**1.03**	**3.03**	**2.10**	**2.27**	**2.31**	0.84	**1.01**
uniprotenc_150m	**1.35**	**2.91**	**1.50**	0.77	0.76	0.22	0.29	0.87	**3.67**	**2.97**	**3.97**	**4.07**	**1.50**	**1.72**	**1.35**	**2.92**	**1.51**	0.76	0.75	0.22	0.28	**1.01**	**2.93**	**2.14**	**2.25**	**2.25**	0.74	0.89
go_sub	**1.65**	**2.25**	**1.63**	**2.28**	**1.60**	**1.77**	**2.33**	**1.58**	**1.67**	**4.46**	**6.28**	**5.47**	**2.44**	**2.31**	**1.82**	**2.76**	**2.04**	**2.34**	**1.83**	**2.84**	**3.44**	**1.66**	**1.88**	**3.79**	**6.17**	**5.10**	**2.34**	**2.33**
pubmed_sub	**1.14**	**1.53**	**1.46**	**1.06**	**1.02**	**1.24**	**1.58**	**1.44**	**1.22**	**1.43**	**2.08**	**2.50**	**2.36**	**2.31**	**1.35**	**2.05**	**1.92**	**1.34**	**1.29**	**2.91**	**3.61**	**1.45**	**1.51**	**1.53**	**1.98**	**2.34**	**2.30**	**2.26**
yago_sub	**1.28**	**2.54**	**1.03**	**1.05**	0.93	**1.74**	**2.22**	**1.31**	**1.51**	**1.40**	**2.22**	**2.50**	**1.96**	**2.19**	**1.71**	**3.65**	**1.64**	**1.43**	**1.25**	**3.10**	**3.79**	**1.37**	**1.86**	**1.44**	**2.07**	**2.31**	**1.95**	**2.29**
citeseer_sub	**1.50**	**1.89**	**1.71**	**1.51**	**1.33**	**1.51**	**1.87**	**1.28**	**1.10**	**1.28**	**1.87**	**1.85**	**1.89**	**1.81**	**1.53**	**2.03**	**1.91**	**1.55**	**1.36**	**2.49**	**2.90**	**1.34**	**1.44**	**1.44**	**1.90**	**1.82**	**1.85**	**1.79**
arXiv	**1.16**	**1.61**	**2.67**	**1.25**	**1.24**	**1.28**	**1.42**	**1.91**	0.98	**1.61**	**2.12**	**2.31**	**2.12**	**2.09**	**1.49**	**1.75**	**2.42**	**1.92**	**1.95**	**4.72**	**5.52**	**1.74**	**1.38**	**2.04**	**1.95**	**2.13**	**2.07**	**2.05**
amaze	**1.05**	**2.31**	0.84	0.85	0.83	**2.98**	**3.72**	**1.68**	**3.40**	**2.38**	**9.11**	**9.80**	**7.76**	**11.21**	**1.17**	**2.91**	**1.35**	**1.94**	**2.10**	**87.92**	**90.01**	**1.34**	**2.62**	**1.62**	**3.59**	**3.86**	**4.15**	**5.85**
kegg	0.97	**2.14**	0.85	**1.01**	0.94	**2.72**	**3.54**	**1.66**	**3.34**	**2.28**	**9.17**	**10.03**	**7.29**	**10.88**	**1.17**	**2.87**	**1.36**	**2.24**	**2.33**	**105.29**	**105.07**	**1.31**	**2.62**	**1.61**	**3.73**	**3.96**	**4.01**	**5.79**
nasa	**1.05**	**1.77**	**1.50**	**1.34**	**1.04**	**1.63**	**2.09**	**1.33**	**1.99**	**5.17**	**13.37**	**11.02**	**6.38**	**7.89**	**1.35**	**2.61**	**2.23**	**2.06**	**1.71**	**4.36**	**4.91**	**1.37**	**2.26**	**3.55**	**11.49**	**9.55**	**4.87**	**5.88**
xmark	**1.18**	**2.01**	**1.57**	**1.41**	**1.36**	**1.85**	**2.36**	**1.87**	**2.19**	**3.99**	**9.48**	**9.68**	**97.90**	**103.17**	**1.31**	**2.53**	**1.97**	**1.81**	**1.71**	**8.77**	**9.40**	**1.60**	**2.37**	**2.69**	**6.13**	**5.99**	**44.76**	**45.83**
vchocyc	0.93	**2.95**	**1.84**	0.87	0.75	**2.21**	**2.81**	**1.72**	**2.86**	**5.57**	**38.80**	**16.40**	**5.34**	**6.43**	0.97	**3.59**	**2.31**	0.97	0.85	**3.03**	**3.68**	**1.32**	**2.94**	**3.31**	**15.10**	**6.99**	**3.25**	**4.10**
mtbrv	0.87	**2.75**	**1.46**	0.83	0.69	**2.05**	**2.62**	**1.49**	**2.36**	**4.31**	**30.50**	**12.26**	**4.61**	**6.25**	0.98	**3.59**	**1.95**	0.99	0.82	**3.12**	**3.76**	**1.27**	**2.81**	**2.88**	**14.41**	**6.18**	**3.09**	**4.20**
anthra	0.96	**2.85**	**1.78**	0.72	0.71	**1.91**	**2.42**	**1.84**	**3.19**	**22.65**	**25.90**	**18.64**	**4.87**	**6.37**	**1.10**	**3.72**	**2.42**	0.79	0.78	**2.63**	**3.18**	**1.36**	**3.12**	**9.41**	**10.81**	**7.93**	**2.81**	**3.74**
ecoo	0.90	**2.88**	**1.85**	0.80	0.69	**2.05**	**2.52**	**1.95**	**2.88**	**7.09**	**20.26**	**17.74**	**5.82**	**7.77**	**1.03**	**3.73**	**2.48**	0.87	0.78	**2.65**	**3.30**	**1.39**	**3.08**	**4.03**	**8.11**	**7.30**	**3.28**	**4.44**
agrocyc	0.82	**2.58**	**1.67**	0.71	0.62	**1.77**	**2.21**	**1.91**	**3.04**	**29.05**	**37.58**	**16.48**	**8.19**	**9.28**	0.99	**3.65**	**2.45**	1.00	0.79	**2.63**	**3.15**	**1.38**	**3.13**	**11.87**	**14.22**	**7.07**	**4.12**	**5.07**
human	0.78	**2.97**	**1.23**	0.46	0.45	**1.07**	**1.40**	**1.65**	**3.86**	**23.00**	**14.94**	**12.74**	**4.42**	**5.47**	**1.01**	**4.26**	**1.99**	0.59	0.57	**1.46**	**1.86**	**1.20**	**3.99**	**10.70**	**6.75**	**5.81**	**2.59**	**3.34**
p2p-Gnutella31	0.81	**3.01**	**1.28**	0.46	0.46	0.97	**1.25**	**1.43**	**2.70**	**4.47**	**6.66**	**7.36**	**8.31**	**10.57**	**1.10**	**4.09**	**1.94**	0.63	0.64	**3.13**	**3.51**	**1.16**	**3.92**	**3.15**	**3.25**	**3.56**	**4.27**	**5.38**
email-EuAll	**1.02**	**2.62**	0.59	0.91	0.89	0.57	0.73	**1.40**	**4.92**	**4.05**	**8.04**	**8.24**	**3.86**	**4.75**	**1.23**	**3.07**	**1.04**	**1.30**	**1.31**	**8.06**	**8.04**	**1.20**	**3.32**	**2.48**	**3.50**	**3.58**	**1.72**	**2.23**
web-Google	**1.04**	**2.32**	**1.81**	**1.11**	**1.07**	0.56	0.74	**1.40**	**3.92**	**5.08**	**9.12**	**9.44**	**4.18**	**4.90**	**1.19**	**2.58**	**2.24**	**1.84**	**1.86**	**16.62**	**16.03**	**1.18**	**3.13**	**3.20**	**3.88**	**4.00**	**1.82**	**2.24**
soc-LiveJournal1	0.89	**2.69**	**1.40**	0.80	0.80	0.49	0.66	**1.32**	**4.11**	**5.09**	**7.39**	**7.63**	**2.90**	**3.44**	**1.04**	**3.15**	**2.29**	**1.96**	**2.01**	**23.84**	**22.75**	**1.15**	**3.35**	**3.17**	**3.40**	**3.50**	**1.36**	**1.75**
wiki-Talk	0.89	**3.24**	**1.15**	0.60	0.59	0.40	0.53	**1.35**	**5.76**	**5.17**	**6.01**	**6.21**	**2.22**	**3.00**	0.98	**3.65**	**1.34**	0.70	0.71	**1.22**	**1.38**	**1.09**	**3.63**	**2.88**	**2.52**	**2.60**	**1.05**	**1.40**
GEOMETRIC MEAN	**1.10**	**2.22**		1.00	0.92	0.88	**1.06**	**1.33**	**1.90**		**4.21**	**3.98**	**2.79**	**3.14**	**1.29**	**2.53**		**1.35**	**1.26**	**2.12**	**2.40**	**1.29**	**2.04**		**2.94**	**2.77**	**2.02**	**2.31**
RATIO RUNTIME AVGS	**1.03**	**1.75**		**1.06**	0.98	1.00	0.99	**1.04**	**1.16**		**1.07**	**1.09**	**1.15**	**1.27**	**1.08**	**1.82**		**1.15**	**1.15**	**1.31**	**1.51**	**1.04**	**1.36**		**1.09**	**1.07**	**1.14**	**1.25**
AVERAGE	**1.13**	**2.32**		**1.09**	0.98	**1.11**	**1.35**	**1.41**	**2.25**		**7.59**	**5.87**	**5.48**	**6.25**	**1.33**	**2.69**		**1.52**	**1.41**	**7.96**	**8.22**	**1.33**	**2.23**		**4.05**	**3.37**	**3.21**	**3.63**

Table A.9 Average query times in µs for 100 000 negative (left) and positive queries (right). Highlighted results are the overall best/second-best after `Matrix` per query set over *all* tested algorithms.

						← negative	positive→					
		`O'R+`		`O'R+`		`O'R+`		`O'R+`		`O'R+`		`O'R+`
Instance	`TF`	`TF`	`IP(s)`	`IP(s)`	`BFL(s)`	`BFL(s)`	`TF`	`TF`	`IP(s)`	`IP(s)`	`BFL(s)`	`BFL(s)`
kron_logn12	0.448	0.150	0.025	0.025	0.074	0.039	2.222	0.966	2.214	0.903	3.100	0.992
kron_logn16			0.072	0.106	0.177	0.179			29.244	12.765	23.413	9.661
kron_logn17			0.091	0.124	0.111	0.119			27.734	6.396	9.437	1.835
kron_logn20			0.164	0.195	0.351	0.388			345.677	167.109	341.645	154.522
kron_logn21			0.204	0.249	0.225	0.281			316.522	191.688	184.889	105.423
randn20-21	0.287	0.150	0.319	0.223	**0.044**	0.123	0.501	**0.364**	2.832	1.815	0.837	0.687
randn20-22	0.449	**0.299**	4.248	4.126	0.840	0.898	1.337	1.160	84.935	83.959	18.779	18.685
randn20-23			198.518	188.459	96.362	95.814			4 720.272	4 656.298	1 683.989	1 656.785
randn23-24	0.438	0.211	0.453	0.328	**0.046**	0.171	0.732	**0.513**	3.785	2.635	1.045	0.880
randn23-25	0.607	**0.396**	5.394	5.178	0.950	1.064	1.589	1.404	113.423	112.633	23.804	23.875
delaunay_n15	0.150	0.055	0.336	0.120	**0.040**	0.045	0.243	0.181	5.105	2.385	0.655	0.490
delaunay_n20	0.367	0.141	0.588	0.223	**0.038**	0.124	0.664	0.495	8.549	5.864	2.085	1.739
delaunay_n22	0.475	0.177	0.667	0.266	**0.039**	0.154	0.818	0.635	8.575	6.658	2.818	2.403
citeseer.scc	**0.023**	0.056	0.052	0.056	0.034	0.056	0.301	0.112	0.320	0.112	0.154	0.112
citeseerx	0.450	0.152	0.183	0.183	0.063	0.154	2.615	0.154	2.792	0.678	2.007	0.482
cit-Patents	1.078	0.533	6.259	6.049	1.845	1.904	10.640	9.168	701.034	708.037	245.211	244.524
go_uniprot	0.115	0.107	0.069	0.098	**0.033**	0.098	44.738	32.490	0.924	0.637	0.613	0.488
uniprotenc_22m	0.080	0.066	0.045	0.066	**0.033**	0.066	0.180	**0.072**	0.332	**0.072**	0.174	**0.072**
uniprotenc_100m	0.187	0.131	0.099	0.131	**0.033**	0.131	0.348	0.118	0.497	0.118	0.201	0.118
uniprotenc_150m	0.229	0.153	0.117	0.153	**0.034**	0.153	0.411	0.139	0.551	0.139	0.208	0.139
go_sub	0.042	0.026	0.089	0.039	0.044	0.025	0.338	0.076	4.302	0.685	0.385	0.158
pubmed_sub	0.069	0.047	0.070	0.066	0.055	0.044	0.228	0.160	1.482	0.714	1.260	0.535
yago_sub	0.024	0.023	0.026	0.024	0.037	**0.021**	0.085	0.060	0.250	0.113	0.178	0.091
citeseer_sub	0.066	0.038	0.100	0.066	0.046	0.030	0.155	0.121	1.247	0.666	0.600	0.317
arXiv	0.681	0.255	0.354	0.283	0.173	0.136	1.470	0.915	6.698	3.161	4.315	2.034
amaze	**0.011**	0.013	**0.011**	0.013	0.039	0.013	0.022	**0.009**	0.083	**0.009**	0.071	**0.009**
kegg	**0.013**	0.015	0.015	0.015	0.041	0.015	0.021	**0.009**	0.086	**0.009**	0.068	**0.009**
nasa	0.039	**0.026**	0.046	0.034	0.042	**0.026**	0.130	0.025	2.216	0.166	0.307	0.048
xmark	0.040	0.025	0.047	0.033	0.043	**0.023**	0.081	0.020	0.461	0.049	2.160	0.022
vchocyc	0.031	0.017	0.015	0.017	0.037	0.017	0.076	**0.014**	0.571	0.015	0.080	0.015
mtbrv	0.026	0.018	0.015	0.018	0.037	0.018	0.071	**0.016**	0.569	0.019	0.078	0.017
anthra	0.033	0.019	0.014	0.019	0.037	0.019	0.307	**0.014**	0.385	0.015	0.067	**0.014**
ecoo	0.034	0.019	0.015	0.019	0.038	0.019	0.100	**0.014**	0.308	0.015	0.084	**0.014**
agrocyc	0.035	0.021	0.015	0.021	0.037	0.021	0.402	**0.014**	0.559	0.015	0.118	**0.014**
human	0.040	0.033	**0.015**	0.033	0.035	0.033	0.496	**0.022**	0.328	**0.022**	0.096	**0.022**
p2p-Gnutella31	0.047	0.037	**0.017**	0.037	0.035	0.036	0.115	**0.026**	0.173	**0.026**	0.215	**0.026**
email-EuAll	0.036	0.061	0.056	0.062	**0.035**	0.061	0.168	**0.042**	0.334	**0.042**	0.160	**0.042**
web-Google	0.135	0.074	0.086	0.077	**0.039**	0.070	0.246	**0.048**	0.442	**0.048**	0.202	**0.048**
soc-LiveJournal1	0.099	0.071	0.057	0.072	**0.034**	0.069	0.298	**0.058**	0.432	**0.058**	0.170	**0.058**
wiki-Talk	0.095	0.083	0.050	0.083	**0.033**	0.083	0.297	**0.057**	0.344	**0.057**	0.127	**0.057**
MIN			**0.011**	0.013	0.033	0.013			0.083	**0.009**	0.067	**0.009**
AVERAGE			5.342	5.059	2.496	2.506			156.016	145.532	62.338	54.329
MAX			198.518	188.459	96.362	95.814			4 720.272	4 656.298	1 683.989	1 656.785

Table A.10 Average query times in µs for 100 000 random (left) and 200 000 mixed queries (right). Highlighted results are the overall best/second-best after `Matrix` per query set over *all* tested algorithms.

						← random	mixed→					
		`O’R+`		`O’R+`		`O’R+`		`O’R+`		`O’R+`		`O’R+`
Instance	`TF`	`TF`	`IP(s)`	`IP(s)`	`BFL(s)`	`BFL(s)`	`TF`	`TF`	`IP(s)`	`IP(s)`	`BFL(s)`	`BFL(s)`
kron_logn12	0.995	0.385	0.631	0.269	2.933	0.297	1.349	0.564	1.128	0.469	1.594	0.520
kron_logn16			6.212	2.731	6.794	2.148			14.705	6.440	11.845	4.923
kron_logn17			5.507	1.385	3.515	0.508			13.973	3.269	4.795	0.983
kron_logn20			54.122	26.731	54.180	25.126			173.231	83.873	170.936	77.473
kron_logn21			45.584	27.339	30.225	15.939			158.059	95.937	92.489	52.906
randn20-21	0.293	0.147	0.329	0.228	**0.047**	0.118	0.413	**0.269**	1.593	1.160	0.450	0.417
randn20-22	0.452	**0.297**	4.161	3.978	0.840	0.895	0.921	**0.747**	44.638	43.895	9.833	9.813
randn20-23			393.758	382.669	161.427	157.768			2 454.174	2 367.299	891.487	879.033
randn23-24	0.449	0.218	0.450	0.306	**0.044**	0.173	0.610	**0.377**	2.139	1.513	0.556	0.542
randn23-25	0.619	**0.405**	5.551	4.324	0.993	1.106	1.131	0.919	59.395	59.119	12.398	12.506
delaunay_n15	0.168	0.055	0.371	0.135	0.077	**0.045**	0.212	0.116	2.742	1.292	0.359	0.271
delaunay_n20	0.372	0.138	0.604	0.217	**0.041**	0.118	0.533	0.330	4.657	3.064	1.075	0.946
delaunay_n22	0.479	0.180	0.671	0.265	**0.040**	0.154	0.669	0.415	4.744	3.268	1.429	1.290
citeseer.scc	**0.029**	0.057	0.052	0.057	0.035	0.057	0.215	0.106	0.213	0.106	0.104	0.106
citeseerx	0.448	0.149	0.184	0.174	**0.078**	0.143	1.587	0.164	1.543	0.440	1.048	0.329
cit-Patents	1.064	0.525	6.626	6.270	1.869	1.911	5.937	4.717	353.571	343.027	123.587	123.261
go_uniprot	0.109	0.101	0.069	0.101	**0.033**	0.101	22.618	16.328	0.540	0.397	0.342	0.322
uniprotenc_22m	0.081	0.068	0.046	0.068	**0.033**	0.068	0.163	**0.092**	0.213	**0.092**	0.104	**0.092**
uniprotenc_100m	0.191	0.134	0.098	0.134	**0.033**	0.134	0.311	0.148	0.337	0.148	**0.124**	0.148
uniprotenc_150m	0.236	0.156	0.118	0.156	**0.034**	0.156	0.365	0.170	0.382	0.170	**0.126**	0.170
go_sub	0.047	0.023	0.091	0.039	0.063	**0.022**	0.196	0.052	2.166	0.351	0.220	0.094
pubmed_sub	0.080	0.042	0.084	0.062	0.115	0.039	0.158	0.103	0.787	0.398	0.667	0.290
yago_sub	0.030	0.018	0.027	0.019	0.050	**0.016**	0.062	0.043	0.145	0.070	0.115	0.059
citeseer_sub	0.077	0.040	0.106	0.068	0.078	0.031	0.119	0.083	0.693	0.364	0.330	0.179
arXiv	0.751	0.311	1.291	0.674	1.929	0.408	1.112	0.545	3.571	1.832	2.253	1.088
amaze	0.019	**0.014**	0.028	**0.014**	1.232	**0.014**	0.024	**0.015**	0.053	**0.015**	0.061	**0.015**
kegg	0.020	**0.015**	0.034	**0.015**	1.545	**0.015**	0.024	**0.015**	0.056	**0.015**	0.060	**0.015**
nasa	0.044	0.020	0.055	0.027	0.083	**0.019**	0.092	0.026	1.150	0.100	0.181	0.037
xmark	0.044	0.022	0.053	0.029	0.174	**0.020**	0.067	0.025	0.261	0.043	1.106	0.025
vchocyc	0.037	0.016	0.016	0.016	0.049	0.016	0.063	**0.019**	0.294	**0.019**	0.064	0.020
mtbrv	0.031	0.016	0.016	0.016	0.050	0.016	0.058	**0.020**	0.307	0.021	0.063	**0.020**
anthra	0.041	0.017	0.014	0.017	0.045	0.017	0.183	**0.019**	0.219	0.020	0.056	0.020
ecoo	0.043	0.017	0.015	0.017	0.045	0.017	0.079	**0.020**	0.165	**0.020**	0.065	**0.020**
agrocyc	0.043	0.017	0.018	0.018	0.046	0.017	0.232	**0.020**	0.287	**0.020**	0.082	**0.020**
human	0.051	0.026	**0.015**	0.026	0.037	0.026	0.290	**0.027**	0.184	**0.027**	0.070	**0.027**
p2p-Gnutella31	0.058	0.030	**0.019**	0.030	0.093	0.030	0.102	**0.032**	0.106	0.033	0.138	**0.032**
email-EuAll	0.059	0.057	0.074	0.057	0.452	**0.056**	0.150	**0.060**	0.213	0.061	0.103	**0.060**
web-Google	0.175	0.078	0.147	0.080	1.231	**0.074**	0.229	0.072	0.285	0.073	0.127	**0.070**
soc-LiveJournal1	0.172	0.075	0.148	0.075	1.748	**0.073**	0.246	0.078	0.266	0.078	0.105	**0.077**
wiki-Talk	0.102	0.076	**0.054**	0.076	0.093	0.076	0.253	0.088	0.221	0.088	0.093	0.088
Min			0.014	0.014	0.033	0.014			0.053	**0.015**	0.056	**0.015**
Average			12.865	11.193	6.645	5.073			80.572	73.625	32.456	28.496
Max			393.758	382.669	161.427	157.768			2 454.174	2 367.299	891.487	879.033