The VLDB Journal (2012) 21:509–534
DOI 10.1007/s00778-011-0256-4

REGULAR PAPER

GRAIL: a scalable index for reachability queries in very large graphs

Hilmi Yıldırım · Vineet Chaoji · Mohammed J. Zaki

Received: 24 February 2011 / Revised: 2 June 2011 / Accepted: 8 September 2011 / Published online: 23 September 2011

Abstract Given a large directed graph, rapidly answering reachability queries between source and target nodes is an important problem. Existing methods for reachability trade-off indexing time and space versus query time performance. However, the biggest limitation of existing methods is that they do not scale to very large real-world graphs. We present a simple yet scalable reachability index, called GRAIL, that is based on the idea of randomized interval labeling and that can effectively handle very large graphs. Based on an extensive set of experiments, we show that while more sophisticated methods work better on small graphs, GRAIL is the only index that can scale to millions of nodes and edges. GRAIL has linear indexing time and space, and the query time ranges from constant time to being linear in the graph order and size. Our reference C++ implementations are open source and available for download at http://www.code.google.com/p/grail/.

Keywords Graph query processing · Scalable graph indexing · Reachability queries

This work was supported in part by NSF Grants EMT-0829835, and CNS-0103708, and NIH Grant 1R01EB0080161-01A1.

H. Yıldırım (✉) · M. J. Zaki
Rensselaer Polytechnic Institute, Troy, NY, USA
e-mail: yildih2@cs.rpi.edu

M. J. Zaki
e-mail: zaki@cs.rpi.edu

V. Chaoji
Yahoo! Labs, Bangalore, India
e-mail: chaojv@yahoo-inc.com

1 Introduction

Let $G = (V, E)$ be a directed graph, where V is the set of vertices, and E is the set of directed edges, with $|V| = n$ and $|E| = m$. A *reachability query* asks whether there exists a path p from a source node u to a target node v in the directed graph G. If such as path exists, we say that u can reach v (or v is reachable from u) and denote it as $u \rightsquigarrow v$. If u cannot reach v, we denote it as $u \not\rightsquigarrow v$. The reachability query itself is denoted as $u \stackrel{?}{\rightsquigarrow} v$.

Answering graph reachability queries quickly has been the focus of research for over 20 years. Traditional applications include reasoning about inheritance in class hierarchies and testing concept subsumption in knowledge representation systems. However, interest in the reachability problem has revived in recent years with the advent of new applications that have very large graph-structured data that are queried for reachability repeatedly. The emerging area of Semantic Web is composed of RDF/OWL data that are indeed graphs with rich content, and there exist RDF data with millions of nodes and billions of edges. Reachability queries are often necessitated on these data to infer the relationships among the objects. In network biology, reachability plays a role in querying protein–protein interaction networks, metabolic pathways and gene regulatory networks. As specific examples of graph reachability consider the following applications.

Semantic query engines: The vision of the Semantic Web is to allow machines to understand the meaning of the information on the World Wide Web [19]. The data stores and interchange formats used include RDF and XML. XML documents are typically trees, but with the use of ID/IDREF links it is possible to represent graphs. On the other hand, RDF documents represent relationships and are naturally graph

```
SELECT ?name
WHERE {
   ?x foaf:name ?name .
   ?y citation:title ?title .
   ?y citation:author ?x .
   ?x citation:cites+/citation:title ?pname .
   FILTER regex(?name, "Joe Example")
   FILTER regex(?pname, "Sample Paper")
}
```

Fig. 1 A SPARQL query that uses property paths

structured. XQuery and SPARQL are the query languages for XML and RDF, respectively. SPARQL lets the user define and search for graph patterns using SQL-like syntax, over RDF stores. With SPARQL 1.1, it is possible to search for patterns that include paths of arbitrary length via *property paths* [29]. For example, the following SPARQL query asks for all researchers named *Joe Example* who might be influenced by the paper titled *Some Paper* in a citation database (Fig. 1).

One strategy to answer this query is to use a reachability index to check if any paper of *Joe Example* can reach *Some Paper* by citations links (`cites+`). With SPARQL 1.1, the necessity for reachability indexes in RDF stores will become more apparent.

Concept subsumption in ontologies: In knowledge representation systems, ontologies define a set of concepts as well as their properties and interrelationships. Ontologies are typically used to describe a domain or to reason about the entities within a domain. In Semantic Web, inference engines generate all the possible conclusions and store the resulting triples in the data store as well. For instance, if a class A is a subclass of B which is a subclass of C, it can be inferred that A is a subclass of C, and this fact is stored in the database. For instance, the Gene Ontology [16], which consists of a directed acyclic graph of terms describing genes, also includes the inferred relationships to speedup querying [17]. This is equivalent to keeping the transitive closure of the graph in the database, which is not scalable for very large graphs. An alternate strategy is to avoid inference computation at least for transitive rules such as *rdfs:subPropertyOf* and *rdfs:subClassOf* [33]. In this case, the inference engine should infer these relations on demand using a reachability labeling mechanism. One of the test cases in our experimental evaluation in Sect. 6 is composed of the terms of the Gene Ontology and the gene products from the Uniprot [31] annotations database. The overall data have more than 7 million nodes and 56 million connections.

Biological networks: With the advance of high-throughput data acquisition technologies, biologists have amassed a large amount of heterogeneous graph data such as metabolic pathway databases, gene regulatory networks, signal transduction networks, etc. In these data sets, nodes represent entities such as proteins, genes, compounds and edges represent how they interact. A query whether a gene A regulates gene B directly or indirectly in a gene regulatory network maps to a reachability query. Similarly, one can ask the question of whether a specific amino acid A is broken down to a specific chemical compound B in a metabolic pathway graph, which is also a reachability query.

In general, given the ubiquity of large graphs in the form of ontologies, social networks, biological networks and so on, there is a crucial need for highly scalable indexing schemes for reachability queries.

1.1 Reachability index

It is worth noting at the outset that the problem of reachability on directed graphs can be reduced to reachability on directed acyclic graphs (DAGs). Given a directed graph G, we can obtain an equivalent DAG, G' (called the *condensation graph* of G), in which each node corresponds to a strongly connected component of the original graph, and an edge exists between two components if any node in one component can reach a node in the other component. To answer whether node u can reach v in G, we simply lookup their corresponding strongly connected components, S_u and S_v, respectively, which are the nodes in G'. If $S_u = S_v$, then by definition u and reach v (and vice-versa). If $S_u \neq S_v$, then we pose the question whether S_u can reach S_v in G'. Thus, all reachability queries on the original graph can be answered on the DAG. Figure 2 shows an example directed graph and the corresponding DAG obtained by coalescing the strongly connected components that are marked in dashed circles. In that graph $N \overset{?}{\rightsquigarrow} P$ is true since they are in the same component.

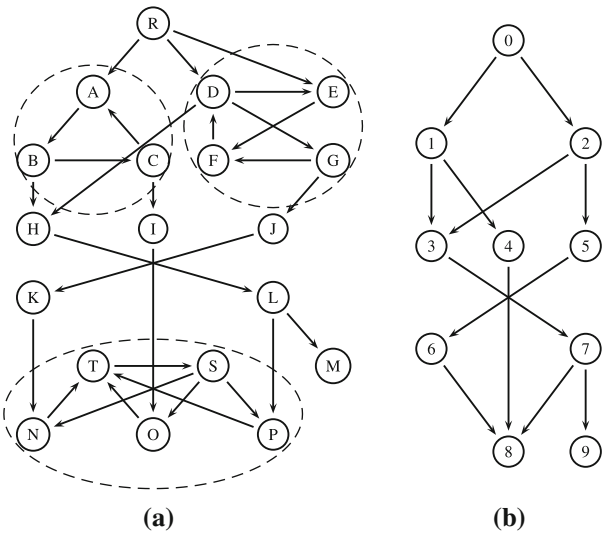

Fig. 2 Coalescing strongly connected components. **a** A directed graph. **b** Corresponding DAG

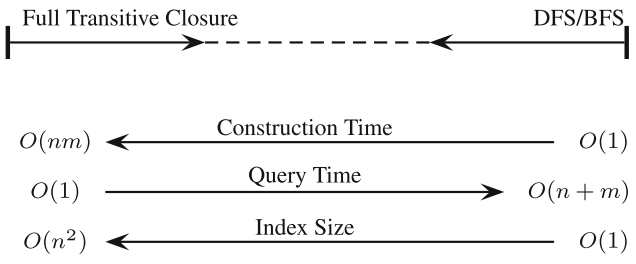

Fig. 3 Trade-off between query time and index size

The query of $G \overset{?}{\rightsquigarrow} P$ in the original graph is equivalent to the query of $2 \overset{?}{\rightsquigarrow} 8$ in the corresponding DAG. Henceforth, we assume that all input graphs have been transformed into their corresponding DAGs, unless otherwise stated.

There are two basic approaches to answer the reachability queries which lie at the two extremes of the index design space, as illustrated in Fig. 3. Given a DAG G, with n vertices and m edges, one extreme (shown on left) is to precompute and store the full transitive closure; this allows one to answer reachability queries in constant time by a single lookup, but it unfortunately requires a quadratic space index, making it practically unfeasible for large graphs. On the other extreme (shown on right), one can use a depth-first (DFS) or breadth-first (BFS) traversal of the graph starting from node u, until either the target, v, is reached or it is determined that no such path exists. This approach requires no index, but requires $O(n+m)$ time for each query, which is unacceptable for large graphs. Existing approaches to graph reachability indexing lie in-between these two extremes.

While there is no single best indexing scheme for DAGs, the reachability problem on trees can be solved effectively by *interval labeling* [15], which takes linear time and space for constructing the index and provides constant time querying. Given a tree/forest $T = (V, E)$ that has $|V| = n$ edges and $|E| = m$ edges, interval labeling assigns each node u a label $L_u = [s_u, e_u]$ that represents an interval starting at s_u and ending at e_u. A desired labeling has to satisfy the condition that L_u subsumes L_v if and only if u reaches v. A reachability query $u \overset{?}{\rightsquigarrow} v$ can then be answered by just comparing the corresponding intervals, i.e., $u \rightsquigarrow v$ if and only if $L_v \subseteq L_u$ ($s_u \leq s_v$ and $e_u \geq e_v$).

Consider the *min-post-labeling* method for trees, which assign $L_u = [s_u, e_u]$ to each node u where $e_u = post(u)$ is the post-order value of the node u, defined as the rank of the node u in a post-order traversal of the tree. The starting value of the interval is defined as $s_u = min\{s_x | x \in children(u)\}$ if u is not a leaf, and $s_u = e_u$ if u is a leaf. It is easy to see that s_u is equivalent to $min\{e_x | x \in descendants(u)\}$, that is, s_u is the lowest post-order rank for any node x in the subtree rooted at u (i.e., including u). Figure 4a shows the min-post-labeling for an example tree. It is easy to see that reachability queries in trees can be answered by interval containment. For example, $1 \rightsquigarrow 9$, since $L_9 = [2, 2] \subset [1, 6] = L_1$, but $2 \not\rightsquigarrow 7$, since $L_7 = [1, 3] \not\subseteq [7, 9] = L_2$.

Interval labeling can be generalized to DAGs. Existing methods [1,5,15,22,30] use either min-post-labeling or pre–post-labeling (see Sect. 2) on a subtree of the DAG, as shown in Fig. 4b (the dashed edges are the non-tree edges). The labeling obtained is the same as in Fig. 4a, because the same tree is used in both approaches, and dashed edges are never followed. However, we can easily see that for DAGs, the labeling is not sufficient to answer reachability queries, since the labeling fails to cover some of the reachable pairs. For example, $2 \rightsquigarrow 7$ but $L_7 = [1, 3] \not\subseteq [7, 9] = L_2$. Existing methods thus have to supplement the index by auxiliary indexes or some other approaches to correctly answer the queries.

1.2 Our contributions

In this paper, we present a novel and scalable graph indexing approach for very large graphs called GRAIL,[1] that stands for **G**raph **R**eachability Indexing via **RA**ndomized **I**nterval **L**abeling. GRAIL applies min-post-labeling directly on the DAG, as opposed to a subtree of the DAG, as in other interval labeling variants. Figure 4c shows the GRAIL labeling on the example DAG. This labeling captures all reachable pairs at the cost of falsely categorizing some pairs as reachable. For instance, compared to Fig. 4b, the label of node 6 is enlarged to [1, 7] since node 8 is a descendant of node 6 and its label is [1, 1]. With this new labeling, $2 \rightsquigarrow 7$ can be answered using label containment (i.e., $L_7 = [1, 3] \subseteq [1, 9] = L_2$), whereas it cannot be answered using the labeling only on the DAG subtree as in Fig. 4b. However, the new labeling introduces false positives, also called as *exceptions*. For example, $L_4 = [1, 4] \subseteq [1, 8] = L_5$ but $5 \not\rightsquigarrow 4$. GRAIL uses different strategies to handle such exceptions, as detailed in Sect. 3.

In this paper, we make the following original contributions:

- To our knowledge, GRAIL is the first direct DAG interval labeling approach. The key idea is to do very fast elimination for those pairs of query nodes for which non-reachability can be determined via the intervals. In other words, if $L_v \not\subseteq L_u$, we immediately return $u \not\rightsquigarrow v$.
- Instead of using a single interval per node, the basic approach GRAIL uses multiple intervals obtained via randomized DAG traversals, a technique we call *randomized multiple interval labeling*. This approach drastically reduces the number of exceptions. We also empirically compare alternative multiple interval labeling methods designed to minimize the number of exceptions.

[1] A preliminary version of GRAIL appears in [34].

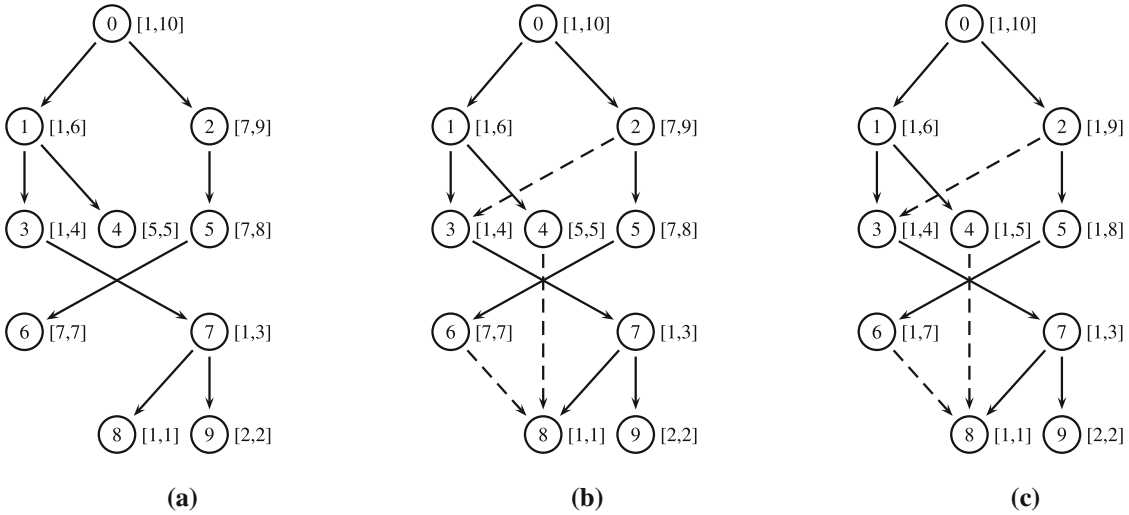

Fig. 4 **a** Min-post-labeling. **b** Min-post-labeling on DAG subtree. **c** Min-post-labeling on DAG

- To guarantee correctness, GRAIL uses "smart" graph search methods with recursive label-based pruning. We study the impact of DFS/BFS search and also explore the use of bidirectional BFS. Independently, these search methods also form optimized baseline methods for comparison against GRAIL. In addition, we propose an alternative method to directly maintain *exception lists* per node, which can eliminate false positives completely. This can offer benefit for indexing smaller graphs, with pruning-based search as the scalable alternative for massive graphs.
- We propose several enhancements to the basic GRAIL approach, which can dramatically improve the querying time. The *topological level filter* is a simple yet effective strategy to prune non-reachable pairs of nodes. The *positive cut filter* uses the subtree labelings induced by each one of the GRAIL's multiple DAG labelings to guarantee reachability for a subset of all the reachable pairs. For these guaranteed reachable pairs u and v, if $L_v \subseteq L_v$, then GRAIL immediately returns $u \rightsquigarrow v$.
- Via an extensive set of experiments, we show that GRAIL outperforms existing methods on very large real and synthetic graphs, sometimes by over an order of magnitude. Whereas previous methods studied graphs mainly under 50 K nodes and 70 K edges when transformed into DAGs, we use large real and synthetic graphs with millions of nodes and edges (see Tables 5, 6). In many cases, GRAIL is the only method that can even run on such large graphs.

GRAIL is a highly effective and light-weight index to answer reachability queries in massive graphs with millions of nodes and edges. It maintains d interval labels per node, and thus, its index size is $O(dn)$. The index construction time for GRAIL is $O(d(n+m))$, since d random graph traversals are made to obtain those labels. For reachability queries, it takes only $O(d)$ time if the query node pairs are not reachable. Likewise, if the node pairs are part of the guaranteed reachable pairs, then again it takes only $O(d)$ time. In other cases, GRAIL resorts to pruning-based recursive graph search, which can take $O(n+m)$ in the worst case. GRAIL thus retains the best properties on both ends of indexing extremes shown in Fig. 3. Since d is typically a small constant (usually capped at $d = 5$ even for the largest graphs), GRAIL requires index construction time and space linear in the graph size and order. It is worth noting that since most large real-world graphs are very sparse, most (random) query node pairs are non-reachable, and these queries can be answered by GRAIL in constant time. Further, due to the positive cut filter, a significant portion of reachable pairs can also be answered in constant time. When either of these fail, GRAIL can take time linear in the graph to answer the query. However, here too the time increases gracefully, since interval-based pruning is applied recursively during the search.

2 Related work

Existing approaches for graph reachability combine aspects of indexing and pure search, trading off index space for querying time. Major approaches can be classified into two main groups, namely interval labeling [1,7,22,30,32] and 2HOP labeling [8,11,18,27,28,35]. These methods are discussed later and summarized in Table 1.

The interval labeling approaches use either the min-post-labeling [1] (illustrated in Fig. 4b), or pre–post-labeling [15,30,5,22] (illustrated in Fig. 5), on a spanning subtree of the DAG. *Pre–post-labeling* assigns $L_u = [s_u, e_u]$ to each node u where s_u and e_u are the pre-order and post-order

Table 1 Comparison of approaches

	Construction time	Query time	Index size
Opt. Tree Cover [1]	$O(nm)$	$O(n)$	$O(n^2)$
GRIPP [30]	$O(m+n)$	$O(m-n)$	$O(m+n)$
Dual labeling [32]	$O(n+m+t^3)$	$O(1)$	$O(n+t^2)$
PathTree [22]	$O(mk)$ or $O(mn)$	$O(\log^2 k)$	$O(nk)$
2HOP [11]	$O(n^4)$	$O(\sqrt{m})$	$O(n\sqrt{m})$
HOPI [28]	$O(n^3)$	$O(\sqrt{m})$	$O(n\sqrt{m})$
GRAIL (this paper)	$O(d(n+m))$	$O(d)/O(n+m)$	$O(dn)$

n number of vertices, m number of edges, $t = O(m-n)$ number of non-tree edges, k number of paths/chains, d number of intervals

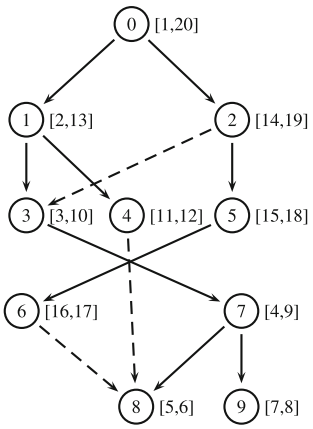

Fig. 5 Pre–post-labeling on a DAG

ranks of node u in a DFS traversal of the DAG, starting from the root(s), with the rank being incremented each time we enter a node or back-track from a node, as shown in Fig. 5. In contrast, as noted previously, in a min-post-labeling, e_u is the post-order rank of u, and s_u is the minimum rank of any node under u, as illustrated in Fig. 4. The interval-based indexing is not self-contained since the labeling can fail to cover some of the reachable pairs. For example, in Fig. 5, $2 \rightsquigarrow 7$ but $L_7 = [4,9] \not\subseteq [14,19] = L_2$. Thus, for completeness, the existing methods supplement the label index with auxiliary indexes. Different interval-based graph indexing methods differ from each other in terms of how the subtree of the DAG is selected, and what kind of auxiliary index is used.

Optimal Tree Cover [1] is the first known variant of interval labeling for DAGs. The approach first creates interval labels for a spanning tree of the DAG. This is not enough to correctly answer the reachability queries, as mentioned previously. To guarantee correctness, the method processes nodes in reverse topological order for each non-tree edge (i.e., an edge that is not part of the spanning tree) between u and v, with u inheriting all the intervals associated with node v. Furthermore, node u inherits the intervals of its other children whose intervals are updated. Thus, u is guaranteed to contain all of its children's intervals. Testing reachability is equivalent to deciding whether the interval list of the source node subsumes the first interval of the target node. The construction complexity of this method is the same as a full transitive closure because selecting the tree that provides optimal labeling requires the pre-computation of the transitive closure.

GRIPP [30] is an another variant of interval labeling. Instead of inflating the index size for the non-tree edges as in [1], reachability testing is done via multiple containment queries. Given nodes u and v, if L_v is not contained in L_u, the non-tree edges (x, y), such that x is a descendant of u, are fetched, and recursively a new query (y, v) is issued for every y, until either v is reachable from a y node or if all non-tree edges are exhausted. If one of the y nodes can reach v, then u can reach v. Since there are $m-n$ non-tree edges, the query time complexity is $O(m-n)$.

Dual labeling [32] also uses interval labeling, but it processes non-tree edges in a different way. Their main observation is that if there exists a single non-tree edge (x, y) in the path from u to v, it must be true that L_u contains x and L_y contains v. Based on this, a non-tree edge $e = (x, y)$ is connected to another non-tree edge $e' = (x', y')$ if and only if L_y contains $L_{x'}$. After labeling the selected tree, *dual labeling* computes the transitive closure of non-tree edges so that the entry for the edge pair $(e = (x, y), e' = (x', y'))$ being 1 implies that all nodes u, whose interval contains x, can reach all nodes v whose interval is contained by $L_{y'}$. Therefore, for each query, they scan relevant edge pairs to find out the reachability. With further optimizations, they reduce the query time to $O(1)$; however, their index size is $O(n+t^2)$, and construction time is $O(n+m+t^3)$ where $t = O(m-n)$ denotes the number of non-tree edges. GRIPP and dual labeling thus lie on the opposite sides of the trade-off illustrated in Fig. 3.

A chain decomposition approach was proposed in [20] to compress the transitive closure. The graph is split into node-disjoint chains. A node u can reach to node v if they exist in the same chain, and u precedes v. Each node also keeps the highest node that it can reach in every other chain. Thus, the space requirement is $O(kn)$ where k is the number of chains. Such a chain decomposition is computed in $O(n^3)$ time. This bound was improved in [7], where they proposed a decomposition which can be computed in $O(n^2 + kn\sqrt{k})$ time. Recently, [6] further improved this scheme by using general spanning trees in which each edge corresponds to a path in the original graph. [3] solves a variant of the reachability problem where the input is assumed to be a collection of non-disjoint paths instead of a graph.

PathTree [22] is the generalization of the tree cover approach. It extracts the disjoint paths of a DAG and then creates a tree of paths on which a variant of interval labeling

is applied. That labeling captures most of the transitive information, and the rest of the closure is computed in an efficient way. PathTree has very fast querying and construction times, but its index size might get very large on dense graphs (k in Table 1 denotes the number of paths in the decomposition). In a recent paper by the same authors, they proposed 3HOP [21] which addresses the issue of large index size. Although 3HOP has a reduced index size, the construction and query times degrade significantly.

The other major class of graph reachability indexing methods is based on *2HOP Indexing* [8,11,18,27,28,35], where each node determines a set of intermediate nodes it can reach, and a set of intermediate nodes which can reach it. The query between u and v returns success if the intersection of the successor set of u and predecessor set of v is not empty. 2HOP was first proposed in [11], where they also showed that computing the minimum 2HOP cover is NP-Hard and gave an $O(\log m)$-approximation algorithm based on a greedy algorithm for the set-cover problem. Its $O(n^4)$ construction time was improved in [35] by using a geometric approach that produces slightly larger 2HOP cover than obtained in [11]. A divide-and-conquer strategy to 2HOP indexing was proposed in [27,28]. HOPI [28] partitions the graph into k subgraphs, computes the 2HOP indexing within each subgraph and finally merges their 2HOP covers by processing the cross-edges between subgraphs [27], by the same authors, improved the merge phase by changing the way in which cross-edges between subgraphs are processed. Cheng [8] partition the graph in a top-down hierarchical manner, instead of a flat partitioning into k subgraphs. The graph is partitioned into two subgraphs repeatedly, and then their 2HOP covers are merged more efficiently than in [27]. Their approach outperforms existing 2HOP approaches in large and dense data sets.

The HLSS [18] method is a hybrid of 2HOP and interval labeling. It first labels a spanning tree of the graph with interval labeling, and then extracts a remainder graph whose transitive closure is yet to be computed. The transitive closure of the remainder graph is computed and compressed by a variant of 2HOP labeling. The overall time complexity for constructing HLSS is $O(m^3)$, but it produces more compressed labelings.

Although there has been much interest in static transitive closure, not much attention has been paid to practical algorithms for the dynamic case, though several theoretical studies exist [13,23,26]. Practical works on dynamic transitive closure [14,24] and dynamic 2HOP indexing [4] have only recently been proposed.

A preliminary version of the GRAIL appeared in [34]. GRAIL is also utilizes interval labeling to construct the index. However, it is diametrically opposite to the other interval labeling approaches mentioned previously. Previous methods provide an incomplete coverage of the reachable set, i.e., they use spanning trees, chain covers, path decompositions and so on, to index the reachability for nodes only in a subtree (or a set of chains/paths) of the input DAG. If $u \rightsquigarrow v$, these methods do not guarantee that $L_v \subseteq L_u$; this is true only for the paths over tree edges. They thus have to use auxiliary indexes to ensure completeness. In contrast, GRAIL uses min-post-traversals directly on the DAG, and it fully covers all reachable pairs, as well as some non-reachable pairs—the false positives. GRAIL guarantees that if $u \rightsquigarrow v$, then $L_v \subseteq L_u$. By contraposition, if the interval label of v is not contained in that for u, GRAIL can immediately return $u \not\rightsquigarrow v$. However, due to the false positives, GRAIL uses graph search, and other optimizations to answer reachability. The GRAIL approach is more favorable in large, sparse graphs, since it is more likely that a random pair of nodes is not reachable, and for these GRAIL can immediately prune the search. GRAIL can also handle higher-density large graphs; since in these cases, the subtree/path-based interval labeling of previous approaches covers an even smaller fraction of the reachable pairs. A detailed description of the basic GRAIL approach and other novel optimizations, and an extensive set of supporting experiments appears in the sections below.

3 GRAIL: Scalable reachability index for large graphs

In this section, we present our novel and scalable GRAIL approach for indexing very large graphs. As opposed to other interval labeling variants, GRAIL uses min-post-labeling directly on the directed acyclic graph. Furthermore, instead of using a single interval, GRAIL employs multiple min-post-intervals that are obtained via random graph traversals (or other strategies). We use the symbol d to denote the number of intervals per node, which also corresponds to the number of graph traversals used to obtain the node label. For example, Fig. 6c shows a DAG labeling using $d = 2$ intervals (the first interval assumes a left-to-right ordering of the children, whereas the second interval assumes a right-to-left ordering).

3.1 Basic GRAIL approach

Our approach to reachability indexing is motivated by the observation that existing interval labeling variants identify a subgraph of the DAG (i.e., trees in [1,30,32] and path-tree in [22]) in the first stage, and incorporate the remaining (uncovered) portion of the DAG, in the second phase of indexing or during the query time. However, most of the reachability information is captured in the first stage. The motivating idea in GRAIL is to use interval labeling multiple times to reduce the workload of the second phase of indexing or the querying. The multiple intervals yield a hyper-rectangle instead of

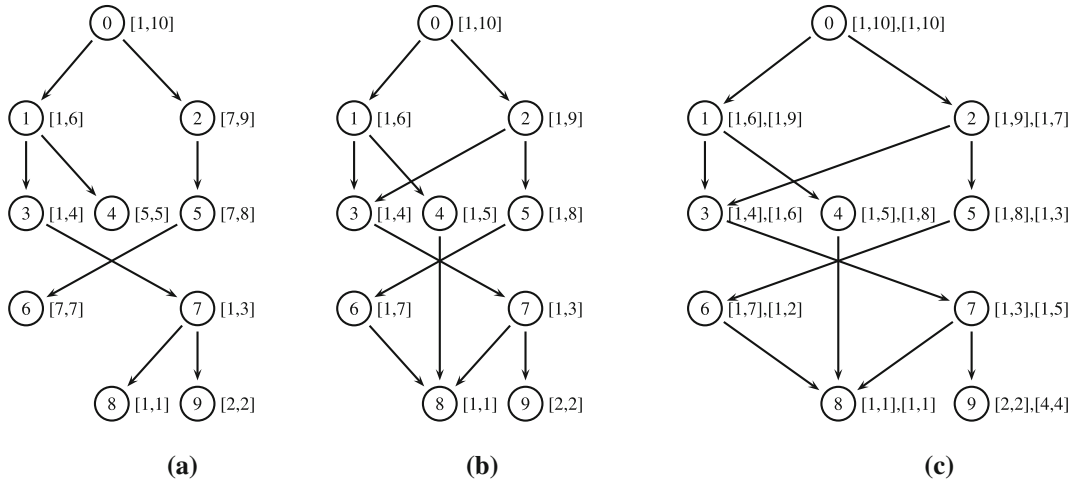

Fig. 6 Interval labeling: Tree (**a**) and DAG: single (**b**) and multiple (**c**)

single interval per node, and the containment check is over these hyper-rectangles.

In GRAIL, for a given node u, the new label is given as $L_u = L_u^1, L_u^2, \ldots, L_u^d$, where L_u^i is the interval label obtained from the i-th (random) traversal of the DAG, and $1 \leq i \leq d$, where d is the *dimension* or number of intervals. We say that L_v is contained in L_u, denoted as $L_v \subseteq L_u$, if and only if $L_v^i \subseteq L_u^i$ for all $i \in [1, d]$. If $L_v \not\subseteq L_u$, then we can conclude that $u \not\rightsquigarrow v$, as per the lemma below.

Lemma 1 *If $L_v \not\subseteq L_u$, then $u \not\rightsquigarrow v$.*

Proof Given that $L_v \not\subseteq L_u$, there must exist a "dimension" i, such that $L_v^i \not\subseteq L_u^i$. Assume that $u \rightsquigarrow v$, and let x and y be the lowest ranked nodes under u and v, respectively, in the post-order traversal. In this case, $L_v^i = [r_y, r_v]$ and $L_u^i = [r_x, r_u]$, where r_n denotes the rank of node n. But, $u \rightsquigarrow v$ implies that $r_u > r_v$ in post-order, and further that $r_x \leq r_y$, which in turn implies that $L_v^i = [r_y, r_v] \subseteq [r_x, r_u] = L_u^i$. But, this is a contradiction to our assumption that $u \rightsquigarrow v$. We conclude that $u \not\rightsquigarrow v$. □

On the other hand, if $L_v \subseteq L_u$, it is possible that this is a false positive, i.e., it can still happen that $u \not\rightsquigarrow v$. We call such a false positive containment an *exception*. For example, in Fig. 6b, there are 15 exceptions in total, as listed in Table 2. For instance, for node 2, node 1 is an exception, since $L_1 = [1, 6] \subseteq [1, 9] = L_2$, but in fact $2 \not\rightsquigarrow 1$. The basic intuition in GRAIL is that using multiple random labels makes it more likely that such false containments, i.e., exceptions, are minimized. For example, when one considers the 2-dimensional intervals given in Fig. 6c, for the very same graph, 12 out of the 15 exceptions get eliminated! For instance, we see that 1 is no longer an exception for 2, since $L_1 = [1, 6], [1, 9] \not\subseteq [1, 9], [1, 7] = L_2$, since for the second interval we have $[1, 9] \not\subseteq [1, 7]$. We can thus conclude that $2 \not\rightsquigarrow 1$. However, note that 3 is still an exception for 4 since $L_3 = [1, 4], [1, 6] \subseteq [1, 5][1, 8] = L_4$. For 4, nodes 7 and 9 also remain as exceptions. In general, using multiple intervals drastically cuts down on the exception list, but is not guaranteed to completely eliminate exceptions.

Table 2 Exceptions for DAG in Fig. 6b

Node	Exceptions (E)	Direct (E^d)	Indirect (E^i)
2	{1, 4}	∅	{1, 4}
4	{3, 7, 9}	{3, 7, 9}	∅
5	{1, 3, 4, 7, 9}	∅	{1, 3, 4, 7, 9}
6	{1, 3, 4, 7, 9}	{1, 3, 4, 7, 9}	∅

There are two main issues in GRAIL: (i) how to compute the d random interval labels while indexing, and (ii) how to deal with exceptions, while querying. We will discuss these in detail later.

3.2 Index construction

The index construction step in GRAIL is very straightforward; we generate the desired number of post-order interval labels by simply changing the visitation order of the children randomly during each depth-first traversal. Algorithm 1 shows an implementation of this strategy; an interval L_u^i is denoted as

$$L_u^i = [s_u^i,\ e_u^i]$$

The number of possible labelings is exponential, but most graphs can be indexed very compactly with a small number of dimensions depending on the edge density of the graph. Furthermore, since it is not guaranteed that all exceptions will be eliminated, the best strategy is to cease labeling after a small number of dimensions (such as 5), with reduced exceptions, rather than trying to totally eliminate

all exceptions, which might require a very large number of dimensions.

Algorithm 1: GRAIL indexing: randomized intervals

```
RandomizedLabeling(G, d):
foreach i ← 1 to d do
    r ← 1  //global variable: rank of node
    Roots ← {n : n ∈ roots(G)}
    foreach x ∈ Roots, in random order do
        Call RandomizedVisit(x, i, G)

RandomizedVisit(x, i, G)  :
if x visited before then return
foreach y ∈ Children(x), in random order do
    Call RandomizedVisit(y, i, G)
r*_c ← min{s^i_c : c ∈ Children(x)}
L^i_x ← [min(r, r*_c), r]
r ← r + 1
```

In terms of the traversal strategies, we aim to generate labelings that are as different from each other as possible. We experimented with the following traversal strategies.

Randomized: This is the strategy shown in Algorithm 1, with a random traversal order for each dimension.

Randomized pairs: In this approach, we first randomize the order of the roots and children, and fix it. We then generate pairs of labeling, using left-to-right (L–R) and right-to-left (R–L) traversals over each node's children. The intuition is to make the intervals as different as possible; a node that is visited first in L–R order is visited last in R–L order. An example of such a pair is shown in Fig. 6c.

Heuristic (guided) traversals: It is also possible to use deterministic approaches with the hope of eliminating exceptions as much as possible. Intuitively, during the ith traversal, the idea would be to choose the node with the most number of exceptions in the first $i-1$ dimensions. However, computing the number of exceptions after each traversal is expensive; instead, we experiment with the following heuristic approaches. For each of the approaches, during the ith traversal, from any node in the graph, we choose the child that maximizes the given objective:

- **Maximum volume:** The volume of a node is defined as the volume of its hyper-rectangle in the first $i-1$ traversals, given as

$$vol(u) = \prod_{j=1}^{i-1} (e_u^j - s_u^j)$$

  Choosing u's child in decreasing order of volume is based on the intuition that a larger volume may contain more exceptions than a smaller volume.

- **Maximum of minimum interval:** The volume of node u can be large even if in one of the first $i-1$ traversals it has a tight interval. Therefore, another approach is to sort the children of u in decreasing order of their minimum interval ranges in the first $i-1$ traversals. The objective is given as:

$$mi(u) = \min_{j=1}^{i-1} \left\{ (e_u^j - s_u^j) \right\}$$

- **Maximum adjusted volume:** The nodes which have bigger reachable sets (the number of nodes they can reach) are expected to have larger intervals and thus larger volumes. Therefore, a larger volume does not directly imply larger number of exceptions. In the ideal case, each node should have intervals whose lengths are equal to the size of its reachable set. Thus, we adjust our computations to eliminate the effect of reachable set sizes by subtracting them from the actual values. The adjusted volume of a node is given as

$$adj\_vol(u) = \prod_{j=1}^{i-1} (e_u^j - s_u^j - tcs(u))$$

  where $tcs(u)$ is the size of the reachable set of u. Since computing the exact tcs values is not practical, we use a linear-time estimation approach [9,10] to approximate the tcs values.

- **Maximum of adjusted minimum interval:** This is similar to the minimum interval, but using the maximum of the adjusted intervals. That is, we sort the children of each node u in decreasing order of

$$adj\_mi(u) = \min_{j=1}^{i-1} \left\{ (e_u^j - s_u^j - tcs(u)) \right\}$$

With randomized or randomized pair traversal, the index construction time for GRAIL is $O(d(n+m))$, corresponding to the d traversals for the graph G. For the other traversal strategies, since we have to sort the children of each node, the construction complexity is $O(d(n+m) + dn(o \log o))$, where o is the maximum out-degree in the graph. Computing adjusted interval (in terms of tcs) does not increase the complexity as we used linear-time reachability set estimation [9]. The space complexity of the GRAIL index is exactly $2dn = O(dn)$, since d intervals are kept per node.

3.3 Reachability queries

To answer reachability queries between two nodes, u and v, GRAIL adopts a two-pronged approach. GRAIL first checks whether $L_v \not\subseteq L_u$. If so, we can immediately conclude that

$u \not\rightsquigarrow v$, by Lemma 1. On the other hand, if $L_v \subseteq L_u$, nothing can be concluded immediately since we know that the index can have false positives, i.e., exceptions.

There are basically two ways of tackling exceptions. One approach is to explicitly maintain an *exception list* per node. Given node x, we denote by E_x, the list of exceptions involving node x, given as:

$$E_x = \{y : (x, y) \text{ is an exception, i.e.,} L_y \subseteq L_x \text{ and } x \not\rightsquigarrow y\}$$

For example, for the DAG in Fig. 6b, we noted that there were 15 exceptions in total, as shown in Table 2. From the table, we can see that $E_2 = \{1, 4\}$, $E_4 = \{3, 7, 9\}$ and so on. If every node has an explicit exception list, then once we know that $L_v \subseteq L_u$, all we have to do is check if $v \in E_u$. If yes, then the pair (u, v) is an exception, and we return $u \not\rightsquigarrow v$. If no, then the containment is not an exception, and we answer $u \rightsquigarrow v$. We describe how to construct exception lists in Sect. 4.

Unfortunately, keeping explicit exception lists per node adds significant overhead in terms of time and space and further does not scale to very large graphs. Thus, the default approach in GRAIL is to not maintain exceptions at all. Rather, GRAIL uses a "smart" DFS with recursive containment check-based pruning to answer queries. This strategy does not require the computation of exception lists so its construction time and index size are linear in the graph.

Algorithm 2: GRAIL query: reachability testing

```
   Reachable(u, v, G):
1  if L_v ⊈ L_u then
2      return False      //u ↛ v
3  else if use exception lists then
4      if v ∈ E_u then return False      //u ↛ v
5      else return True      //u ⇝ v
6  else
       //DFS with pruning
7      foreach c ∈ Children(u) such that L_v ⊆ L_c do
8          if Reachable(c, v, G) then
9              return True      //u ⇝ v
10     return False      //u ↛ v
```

Algorithm 2 shows the pseudo-code for reachability testing in GRAIL. Line 1 tests whether $L_v \not\subseteq L_u$, and if so, returns false. Line 3 is applied only if exception lists are explicitly maintained, either complete or memoized (see Sect. 4): if $v \in E_u$, then GRAIL returns false, otherwise it returns true. Lines 7–10 implement the default strategy of recursive DFS with pruning. If there exists a child c of u, which satisfies the condition that $L_v \subseteq L_c$, and we check and find that $c \rightsquigarrow v$, we can conclude that $u \rightsquigarrow v$, and GRAIL returns true (Line 9). Otherwise, if none of the children can reach v, then we conclude that $u \not\rightsquigarrow v$, and we return false in Line 10. As an example, let us consider the single interval index in Fig. 6b. Let $u = 2$, and let $v = 4$, and assume that we are not using exception lists. Since $L_4 = [1, 5] \subseteq [1, 9] = L_2$, we have to do a DFS to determine reachability. Both 3 and 5 are children of 2, but only 5 satisfies the condition that $L_4 = [1, 5] \subseteq [1, 8] = L_5$, we, therefore, check if 5 can reach 4. Applying the DFS recursion, we will check 6 and then finally conclude that 5 cannot reach 4. Thus, the condition in Line 8 fails, and we return false as the answer (Line 10), i.e., $2 \not\rightsquigarrow 4$.

Computational complexity: It is easy to see that querying takes $O(d)$ time if $L_v \not\subseteq L_u$. If exception lists are to be used, and they are maintained in a hash table, then the check in Line 3 takes $O(1)$ time; otherwise, if the exceptions list is kept sorted, then the times is $O(log(|E_u|))$. The default option is to perform DFS, but note that it is possible we may terminate early due to the containment-based pruning. Thus, the worst-case complexity is $O(n + m)$ for the DFS, but in practice, it can be much faster, depending on the topological level of u and depending on the effectiveness of pruning. Thus, the query time ranges from $O(d)$ to $O(n + m)$.

4 Exception lists

A naive approach for computing the exception lists is to precompute the transitive closure of each node and then to check whether each pair of nodes is an exception or not. Its overall complexity is $O(nm + n^2d)$. $O(nm)$ is for computing transitive closure, and $O(n^2d)$ is the time spent for checking each pair. It also requires quadratic space. This is clearly not acceptable for large graphs. Instead, we suggest another approach (see Algorithm 3) that has a better computational complexity and is faster in practice.

We categorize exceptions into two classes: (i) If L_u contains v,[2] but none of the children of u contain v, then we call the exception between u and v a *direct exception*. (ii) If at least one child of u contains v as an exception, then we call the exception between u and v an *indirect exception*. For example, in Fig. 6b, 3 is a direct exception for 4, but 1 is an indirect exception for 2, since 5 is a child of 2, and 1 is also an exception for 5. Table 2 shows the list of direct (denoted E^d) and indirect (denoted E^i) exceptions for the DAG in Fig. 6b.

The algorithm first computes the direct exceptions considering only the first dimension (or traversal). Then, indirect exceptions from the first traversal are inferred via the method *ExtractIndirectExceptions*. After each node, u has two lists, E^d_u and E^i_u, namely the direct and indirect exception lists,

[2] In this section, the phrases "u contains v", "L_u contains v", and "u contains L_v" are used interchangeably. All are equivalent to saying that L_u contains L_v.

and these lists are updated after each additional traversal, via the method *ShrinkExceptionLists*.

Algorithm 3: GRAIL: exception list extraction

```
   ExceptionListExtraction(G, L, d):
1  T ← Reverse Topological Order(G)
2  Call FindDirectExceptions(G, L, E^d)
3  Call ExtractIndirectExceptions(G, L^1, E^d, E^i, C)
4  foreach i ← 2 to d do
5      ShrinkExceptionLists(G, L, E^d, E^i, C, dim)

   FindDirectExceptions(G, L, E^d) :
6  Initiate an empty IntervalTree IT
7  T ← Sort by Increasing Interval Size
8  foreach node u ∈ T do
9      c ← Sort children of u - with keys (s^1_{c_j}(↑), e^1_{c_j}(↓))
10     qr ← Find query regions by scanning c
11     foreach region r_i ∈ q_r do
12         Query IT for the range r_i
13         Add resulting nodes to E^d_u
14     Add the interval [s^1_{T_i}, e^1_{T_i}] to IT

   ExtractIndirectExceptions(G, L, E^d, E^i):
15 foreach node u in bottom up order do
16     foreach x ∈ (⋃_{j=1}^{k} E_{c_j}) where c_j is a child of u do
17         if x is not a proper descendant of any child c then
18             Add x to E^i_u
19             C^x_u ← Number of children u that contain x

   ShrinkExceptionLists(G, L, E^d, E^i, C, dim):
20 foreach node u do
21     foreach exception x ∈ E^d_u do
22         if L^{dim}_u ⊉ L^{dim}_x then
23             Remove x from E^d_u
24             foreach parent p of u where x ∈ E^i_p do
25                 C^x_p ← C^x_p − 1
26                 if C^x_p = 0 then
27                     Move x from E^i_p to E^d_p
```

Direct exceptions: Let us assume that $d = 1$, that is, each node has only one interval. GRAIL uses an *interval tree* [12] which keeps a list of intervals. Querying the interval tree for intervals intersecting a given range or interval of interest can be done in $O(\log n + K)$ time, where n is the number of intervals, and K is the number of results returned. Given the interval labeling, GRAIL constructs the exception lists for all nodes in the graph as described later. We illustrate how the algorithm works on the example in Fig. 7, where we want to determine the exceptions for node u. c_i denote the children's intervals, whereas x_i denote the exceptions to be found.

The nodes are processed in increasing interval size order so that when a node u is being processed, all the intervals that are contained in L_u are in the interval tree (line 7). To find the

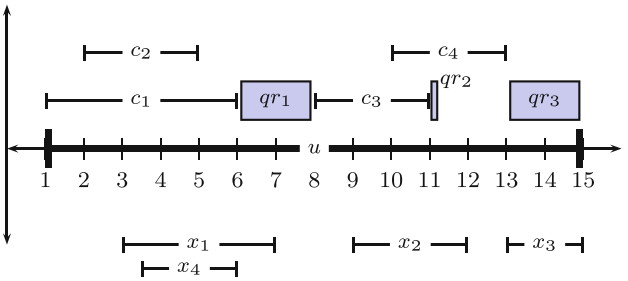

Fig. 7 Direct exceptions: c_i denote children, and x_i denote exceptions, for node u

direct exceptions of u, we look for the intervals that are not covered by any of the children of u using the interval tree. We first sort the children by increasing s_c^1 values, and if there is a tie on these values, they are ordered by decreasing e_c^1 values (line 9). It is clear that if an exception is contained completely within any one of the children intervals, it cannot be a direct exception. There are two cases where a direct exception can occur: (i) The regions not covered by the union of the child intervals. We can see that these regions are $qr_1 = [6, 8]$ and $qr_3 = [13, 15]$ in our example. These regions are queried in the interval tree, and we find the nodes x_1 and x_3 as the direct exceptions of u. (ii) Some exceptions might fall into the region of the union of two consecutive children as in the case of c_3 and c_4 in our example. x_2 is not covered by both of them, but it is an exception for u so it should be a direct exception. To find such nodes, we query a very tiny interval at the end of the first child. In our case, we query the region $[11, 11 + \delta]$, where δ is a constant less than 1. Note that we can skip the end regions of the children that are contained by other children such as in the case of c_1 and c_2. For example, if we had queried the region $[5, 5 + \delta]$, it would return x_4 that is indeed covered by c_1 so it is not a direct exception. Thus, scanning the intervals in that particular order is sufficient to find out such query regions (line 10). Next, we query each region in the interval tree and add the resulting intervals to the direct exception list of the node being processed (lines 11–13). Finally, that node is added to the interval tree so that it can be found if it is an exception for other nodes. The complexity of this method is $O(no(\log n + p))$, where o is the maximum out-degree, and p is maximum number of exceptions returned by a query. This is because at each node, we can query at most o times, each of which runs in $O(\log n + p)$.

Indirect exceptions: Given that we have the list of direct exceptions E_u^d for each node, the construction of the indirect exceptions (E_u^i) proceeds in a bottom-up manner from the leaves to the roots. Let $E_{c_j} = E_{c_j}^d \cup E_{c_j}^i$ denote the list of direct or indirect exceptions for a child node c_j of u. To compute E_u^i, for each exception $x \in E_{c_j}$ we check whether there

exists another child c that can reach x (line 17). This reachability check is easy since by that time we know the exception list of c (i.e., E_c) A child c_k reaches to x if $L_x^1 \subseteq L_c^1$ and $x \notin E_c$. On the other hand, if there is no such children, then x must be an indirect exception for u, and we add it to E_u (line 24). We also keep a counter for each exception x of the node u, C_u^x , which records the number of children of u that have x as exception. These counters are utilized while incorporating the other traversals. For example, consider node 2 in Fig. 6b. Assume we have already computed the exception lists for each of its children, $E_3 = E_3^d \cup E_3^i = \emptyset$, and $E_5 = E_5^d \cup E_5^i = \{1, 3, 4, 7, 8\}$. We find that for each $x \in E_5$, nodes 1, and 4 fail the test with respect to E_3, since $L_1 \not\subseteq L_3$ and $L_4 \not\subseteq L_3$, therefore, $E_2^i = \{1, 4\}$, as illustrated in Table 2. The complexity of this step is $O(neo^2)$, where o is the maximum out-degree, and e is maximum number of exceptions a child node has. This is because every exception of a child node c_i of a node u has to be checked in the exception list of every other child c_j.

Multiple intervals: To find the exceptions when $d > 1$, GRAIL first computes the direct and indirect exceptions from the first traversal, as described previously. We maintain the counters C_u^x to keep track of the number of children of u which has x in their exception list. It is worth noting that an exception can be removed only when C_u^x becomes zero. For computing the remaining exceptions after the ith traversal, GRAIL processes the nodes in a bottom-up order. For every direct exception $x \in E_u^d$, remove x from the direct exception list if x is not an exception for u for the ith dimension (line 24) and further decrement the counter for x in the indirect exceptions list E_p^i for each parent p of u (line 25). Also, if after decrementing, the counter for any indirect exception x becomes zero, then move x to the direct exception list E_p^d of the parent p, providing $L_x \subseteq L_p$ (lines 26–27). In this way, all exceptions can be found out for the i-th dimension or traversal. The complexity of this step is $O(n\phi e)$ as every direct exception is checked in constant time, and the counters for the parent nodes are decremented if needed in $O(\phi)$ time, where ϕ is the maximum in-degree.

Therefore, the overall complexity of exception list maintenance is $O(neo(\log n + e + o))$ with the reasonable assumption of maximum in-degree is in the order of maximum out-degree. It is expected to be better than $O(nm)$ as long as e is not close to n. In practice, the bottleneck of this algorithm is the second step as there are many exceptions just after the first traversals. However, the number of exceptions reduces drastically with the addition of second traversal; therefore, the *shrinking exception lists* step runs very fast although it is called multiple times. It is possible to speedup the algorithm if direct exceptions can be found by considering the first two traversals; we plan to explore this idea in the future.

5 GRAIL optimizations

In this section, we discuss several optimizations over the basic GRAIL approach. The proposed methods can be very effective in speeding up the query times.

5.1 Topological level filter

Once the graph is transformed into a directed acyclic graph, the nodes can be divided into levels so that the leaf nodes are placed into the first level, and the interior nodes are placed in a higher level than all of their children. Therefore, the level of u is

$$l_u = \begin{cases} 1 & \text{if } u \text{ is leaf} \\ 1 + \max\limits_{v \in children(u)} \{l_v\} & \text{otherwise} \end{cases}$$

There is no possibility for a node u at level l_u to reach another node v at level l_v if $l_v \leq l_u$. Thus, the topological level filter can be utilized during GRAIL's recursive DFS to prune unreachable node pairs. Even though this filter is very simple, our experimental results show that it is very effective in improving query times.

5.2 Different search strategies

We perform a depth-first search from the source node to the target node in Algorithm 2 for reachability testing. However, it is not always the best strategy. Breadth-first search (BFS) guarantees to find the shortest path with the cost of using more memory, though this is not a limitation in our case because the memory usage is limited by the graph size. In average, BFS is expected to be faster considering the cases where the target node is close to the source node that has a large reachability set. However, this difference might not be apparent in sparse graphs where DFS is also very fast. In negative queries, they both have to exhaust the whole reachable set to conclude non-reachability, so they are expected to have similar performance.

Besides the standard graph search strategies of DFS and BFS, we also consider bidirectional breadth-first search (BBFS) that has a few advantages over BFS. In bidirectional search, two BFS are started simultaneously utilizing separate queues for the query $u \overset{?}{\rightsquigarrow} v$. The first adds all the children $c_i \in children(u)$ to a BFS queue Q_C of children, whereas the second adds all the parents $p_i inparents(v)$ to another BFS queue Q_P of parents. Then, BBFS alternatingly extracts one node from Q_C pushing its children back to Q_C and extracts one node from Q_P pushing its parents back to Q_P. BBFS stops positively if any newly added node c_i is already in Q_P, or any newly added node p_j is already in Q_C, and BBFS returns $u \not\rightsquigarrow v$ the moment either of the queues becomes empty. Furthermore, BBFS prunes the branch of c_i

by not adding into Q_C if $L_v \not\subseteq L_{c_i}$. Symmetrically, p_j is not added to Q_P if $L_{p_j} \not\subseteq L_u$. The first advantage of bidirectional search is that it can answer positive queries faster when the average degree of the graph is high. In negative queries, the worst-case scenario is that bidirectional search takes twice as much time as a BFS would take; however, there are substantial amount of cases where bidirectional search is much faster than BFS. For instance, consider the case in which the source node has a large reachable set and the target has a small number of ancestors. In this scenario, bidirectional search terminates after visiting the ancestors of the target node that is significantly smaller than the search space of a standard BFS that has to exhaust the large reachable set of the source node.

Our experiments show that BBFS can be a very effective strategy, especially in combination with the level filter. In fact, we use the three different search strategies DFS, BFS and BBFS (with and without the level filter) as the baseline methods to compare against GRAIL as well as other indexing methods.

5.3 Positive cut filter

Recall that GRAIL use direct DAG interval labeling to guarantee that if $u \rightsquigarrow v$ then $L_v \subseteq L_u$. However, it can have false positives, i.e., for a pair of nodes with $L_v \subseteq L_u$, it may be that case that $u \not\rightsquigarrow v$. Recall also that other interval labeling methods try to label only a subtree (or a set of paths), and for paths restricted to the tree edges (which typically constitute only a small fraction of the DAG), reachability can be guaranteed. It is possible for GRAIL to also make use of the subtree edges, via a new pruning method we call *positive cut* filter.

Notice that from each of the d traversals in GRAIL, we can easily keep track of additional interval labels restricted to only a (spanning) subtree of the input DAG. For example, in Fig. 8a, a min-post-labeling that is produced by a traversal of GRAIL is shown. We call these set of labels the *exterior labels*, denoted as EL_u for node u. Figure 8b is the min-post-labeling of the corresponding spanning tree, which is composed of the solid edges (the dashed edges are the non-tree edges). We call this set of labels the *interior labels*, denoted as IL_u for node u. The figure shows only one label, but for d traversals, there are a set of d exterior and interior labels.

Given a query $u \overset{?}{\rightsquigarrow} v$, we can first check whether $IL_v \subseteq IL_u$. If yes, then we are guaranteed that $u \rightsquigarrow v$, and we can immediately return a positive answer. On the other hand, if the interior check fails, then we default to the basic GRAIL strategy of first checking the exterior set and then recursive search. That is, we next check if $EL_v \not\subseteq EL_u$. If yes, then we immediately return a negative answer $u \not\rightsquigarrow v$. If no, then we do a recursive DFS (or BBFS, etc.). This positive cut filter strategy allows GRAIL to substantially speedup the positive queries. In effect, GRAIL captures the core benefits of interval labels in both directions, allowing it to cut-off the search for guaranteed positive pairs (over the tree edges, the interior labels), and also allowing it to prune the non-reachable pairs (using the exterior labels). Figure 8c illustrates this positive cut filter via a Venn diagram over $d = 2$ traversals. The solid black line (rounded outer rectangle) demarcates the universal set of all possible

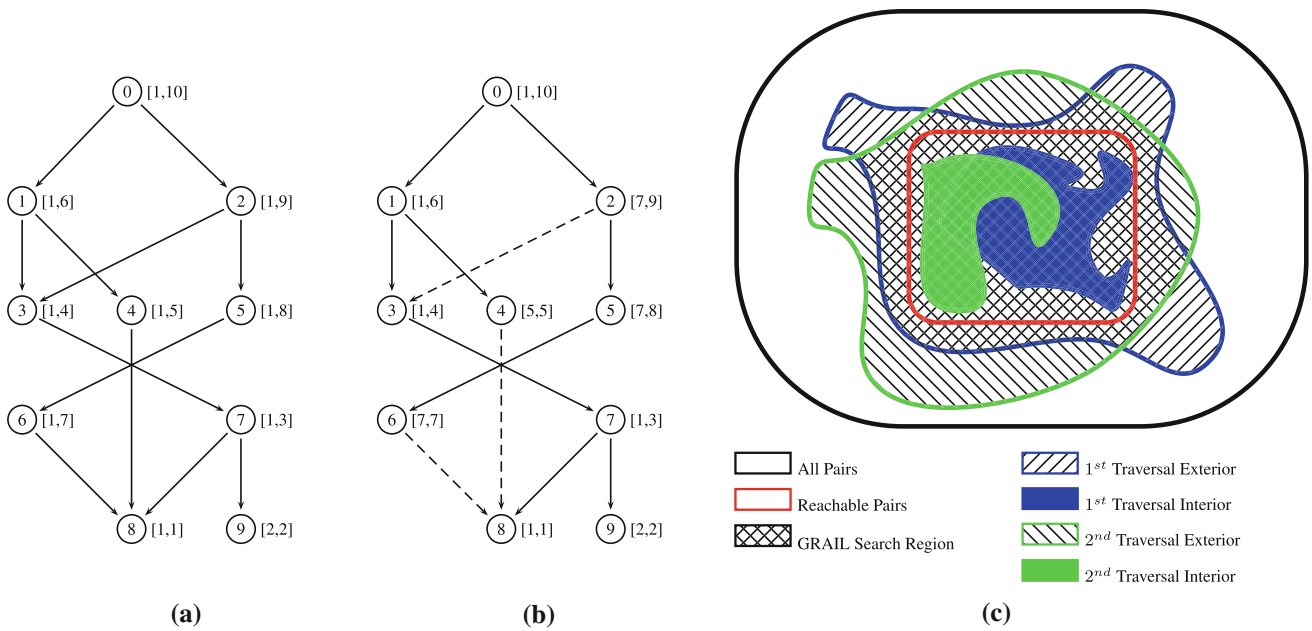

Fig. 8 Venn diagram of pairs wrt. reachability. **a** Exterior labels. **b** Interior labels. **c** Venn diagram

ordered pairs of nodes (all possible queries) in the graph. The solid red line (inner rounded rectangle) denotes the transitive closure, i.e., the set of all reachable pairs of nodes. Each GRAIL traversal is associated with two label sets. The exterior set defined by the min-post-labeling of the graph is a superset of the reachable set (denoted by the outer blue and green hatched regions). The interior set defined by the min-post-labeling of the tree is a subset of the reachable set (inner solid blue and green regions). The intersection of the exterior sets (the crosshatched region) defines the ordered pairs for whom GRAIL is not sure whether they are reachable or not. The union of the interior sets defines the set of ordered pairs which GRAIL guarantees to be reachable. If the two checks over the interior or exterior labels fail, then GRAIL must perform a recursive search to determine reachability; this corresponds to the visible crosshatched region, which excludes the interior regions. The recursive search continues as long as it is in the crosshatched region. GRAIL prunes a branch whenever it exits the region and it halts with a positive result whenever it reaches a pair inside the union of interior sets.

Experimental results in Sect. 6 show that in many cases using positive cut filter substantially improves the GRAIL query performance. It is especially very effective on positive queries. Note that it does not deteriorate the construction time as the exterior and interior labelings can be created simultaneously. Furthermore, instead of doubling the index size, the increase is only a factor of 1.5. This is because, the two intervals (interior and exterior) per node can be stored by three integers. The interval label for node u for the i^{th} dimension/traversal is given as $L_u^i = [s_u^i, m_u^i, e_u^i]$, where $EL_u^i = [s_u^i, e_u^i]$ and $IL_u^i = [m_u^i, e_u^i]$ are the exterior and interior intervals, respectively. For example, in Fig. 8, the exterior label $EL_5 = [1, 8]$ and interior label $IL_5 = [1, 7]$ for node 5 are stored as a single label $L_5 = [1, 7, 8]$. Algorithm 4 shows the GRAIL querying method using the positive cut filter. For clarity, the method is shown using separate EL_u and IL_u labels, but in reality, only L_u is maintained.

Algorithm 4: GRAIL+query: reachability testing

```
Reachable(u, v, G):
1 if IL_v ⊆ IL_u then
2     return True //u ⇝ v
3 else if EL_v ⊈ EL_u then
4     return False //u ↛ v
5 else
      //DFS with pruning
6     foreach c ∈ Children(u) such that EL_v ⊆ EL_c do
7         if Reachable(c, v, G) then
8             return True //u ⇝ v
9 return False //u ↛ v
```

6 Experiments

We conducted extensive experiments to compare our algorithm with the best existing methods. All experiments are performed in a machine with x86_64 Dual Core AMD Opteron Processor 870, which has 8 processors and 32 GB ram. We compared our algorithm with several baseline search methods like DFS (depth-first), BFS (breadth-first) and BBFS (bidirectional BFS), as well as state-of-the-art reachability indexes, such as those based on interval labeling: Interval (INT) [1], GRIPP [30] and Dual Labeling (Dual) [32], based on path decomposition: PathTree (PT) [22], and variants of 2HOP indexing: HLSS [18] and 3HOP [21]. The code for these methods was obtained from the authors, though in some cases, the original code had been reimplemented by later researchers. We implemented GRIPP in-house.

We evaluate all methods in terms of the query time, index size and index construction time. All query times reported below are aggregate times for 100 K queries. We generate 100 K random query pairs and issue the same set of queries to all methods. In the tables later, we use the notation –(t), and –(m), to note that the given method exceeds the allocated time (10 M milliseconds (ms) for small sparse, and 20 M ms—about 5.5 h—for all other graphs; $M \equiv million$) or memory limits (32 GB RAM; i.e., the method aborts with a `bad-alloc` error). All times reported are averages over three runs.

6.1 Data sets

We used a variety of real graph data sets, both small and large, as well as large synthetic ones, as described below. We present these graphs in 4 tables each of which lists the following properties: node size, edge size, average degree, average clustering coefficient (CC) and effective diameter (ED). As we are dealing with directed graphs, the average degree of a graph is equal to the ratio of the number of edges-to-the number of nodes. We used SNAP [25] software to compute the values of clustering coefficient and effective diameter.

Small sparse: These are small, real graphs, with average degree less than 1.2, taken from [22], and listed in Table 3. `xmark` and `nasa` are XML documents, and `amaze` and `kegg` are metabolic networks, first used in [30]. Others were collected from BioCyc (http://www.biocyc.org), a collection of pathway and genome databases. `amaze` and `kegg` have a different structure, in that they have a central node that has a very large in-degree and out-degree. Therefore, these two graphs have smaller effective diameter although the number of reachable pairs is very high compared to other graphs.

Table 3 Small and sparse real graphs

Data set	Nodes	Edges	Avg deg	CC	ED
agrocyc	12,684	13,657	1.06	0.33	11.98
amaze	3,710	3,947	1.06	0.31	3.79
anthra	12,499	13,327	1.07	0.32	11.89
ecoo	12,620	13,575	1.08	0.32	12.23
human	38,811	39,816	1.03	0.39	11.74
kegg	3,617	4,395	1.22	0.26	4.08
mtbrv	9,602	10,438	1.09	0.30	11.90
nasa	5,605	6,538	1.17	0.07	14.22
vchocyc	9,491	10,345	1.09	0.07	11.91
xmark	6,080	7,051	1.16	0.00	11.34

Table 4 Small and dense real graphs

Data set	Nodes	Edges	Avg deg	CC	ED
arxiv	6,000	66,707	11.12	0.35	5.48
citeseer	10,720	44,258	4.13	0.28	8.36
go	6,793	13,361	1.97	0.07	10.92
pubmed	9,000	40,028	4.45	0.10	6.32
yago	6,642	42,392	6.38	0.24	6.57

Small dense: These are small, dense real-world graphs taken from [21] (see Table 4). `arxiv`,[3] `citeceer`,[4] and `pubmed`[5] are all citation graph data sets. GO is a subset of the Gene Ontology[6] graph, and yago is a subset of the semantic knowledge database YAGO.[7] Among these graphs, `go` is significantly different from others with its low average degree and clustering coefficient and high diameter. Since it is a graph representing a taxonomy, triangles in it implies the existence of redundant edges (e.g., if node a has b and c as neighbors and b has c, the edge between a and c is redundant in a taxonomy due transitivity). Therefore, it is natural to have low clustering coefficient.

Large real: To evaluate the scalability of GRAIL on real data sets, we collected 7 new data sets which have previously not been used by existing methods (see Table 5). `citeseer`, `citeseerx` and `cit-patents` are citations networks in which non-leaf nodes have 10–30 outgoing edges on average. However, `citeseer` is very sparse because of data incompleteness. `citeseerx`[8] is the complete citation graph as of

[3] http://www.arxiv.org.

[4] http://www.citeseer.ist.psu.edu.

[5] http://www.pubmedcentral.nih.gov.

[6] http://www.geneontology.org.

[7] http://www.mpi-inf.mpg.de/suchanek/downloads/yago.

[8] http://www.citeseerx.ist.psu.edu.

Table 5 Large real graphs: sparse and dense

Data set	Nodes	Edges	AD	CC	ED
citeseer	340,945	312,282	0.92	0	5.7
uniprot22m	1,595,444	1,595,442	1.00	0	3.3
uniprot100m	16,087,295	16,087,293	1.00	0	4.1
uniprot150m	25,037,600	25,037,598	1.00	0	4.4
citeseerx	6,540,399	15,011,259	2.30	0.06	8.4
cit-patents	3,774,768	16,518,947	4.38	0.09	10.5
go-uniprot	6,967,956	34,770,235	4.99	0	4.8

Table 6 Large synthetic graphs

Data set	Nodes (M)	Edges (M)	Avg deg	CC	ED
rand10m2x	10	20	2	0	13.61
rand10m5x	10	50	5	0	8.75
rand10m10x	10	100	10	0	6.86
rand100m2x	100	200	2	0	?
rand100m5x	100	500	5	0	?

March 2010. `cit-patents`[9] includes all citations within patents granted in the US between 1975 and 1999. `go-uniprot` is the joint graph of Gene Ontology terms and the annotations file from the UniProt[10] database, the universal protein resource. Gene Ontology is a directed acyclic graph of size around 30 K, where each node is a term. UniProt annotations consist of connections between the gene products in the UniProt database and the terms in the ontology. UniProt annotations file has around 7 million gene products annotated by 56 million annotations. The remaining uniprot data sets are obtained from the RDF graph of UniProt. `uniprot22m` is the subset of the complete RDF graph that has 22 million triples, and similarly, `uniprot100m` and `uniprot150m` are obtained from 100 million and 150 million triples, respectively. These are some of the largest directed acyclic graphs ever considered for reachability testing. The reasons why these graphs have zero clustering coefficient vary. `uniprot` graphs have distinctive topology of having many roots that are connected to a single sink node via short paths. It is kind of a tree whose edges are reversed. Therefore, there exists no triangles. For `go-uniprot`, the reason is its being a taxonomy, and for citeseer, it is data incompleteness.

Large synthetic: To test the scalability with different density setting, we generated random DAGs, with 10 and 100 M nodes and with average degrees of 2, 5 and 10 (see Table 6). We first randomly select an ordering of the nodes which corresponds to the topological order of the final dag. Then, for

[9] http://www.snap.stanford.edu/data.

[10] http://www.uniprot.org.

Table 7 Small sparse graphs: construction time (ms)

Data set	GRAIL	HLSS	INT	Dual	PT	3HOP	GRIPP
agrocyc	18.33	12,832.40	5,188.56	12,092.73	269.71	108,761.47	**4.72**
amaze	5.67	684,259.17	3,074.14	4,621.47	839.31	3,387,549.83	**2.73**
anthra	17.04	11,935.63	4,916.71	11,375.35	259.08	95,874.58	**4.64**
ecoo	20.26	12,958.68	5,058.26	11,892.17	269.01	110,114.53	**4.76**
human	52.33	129,599.17	45,352.85	128,064.60	831.37	−(m)	**16.02**
kegg	5.83	1,112,604.10	3,466.63	5,381.69	968.50	4,044,018.00	**2.56**
mtbrv	14.46	3,630.32	2,517.65	3,130.82	204.91	65,721.12	**3.53**
nasa	9.29	1,748.71	805.65	1,037.23	124.67	50,839.88	**2.60**
vchocyc	15.04	4,840.56	2,543.42	4,350.59	201.27	66,401.13	**3.47**
xmark	11.31	68,687.95	1,512.11	1,788.98	260.53	197,554.38	**2.60**

Table 8 Small sparse graphs: index size (Num. Entries)

Data sets	GRAIL	HLSS	INT	Dual	PT	3HOP	GRIPP
agrocyc	88,788	40,097	**27,100**	58,552	39,027	38,631	27,314
amaze	25,970	17,110	10,356	433,345	12,701	10,221	**7,894**
anthra	87,493	33,532	**26,310**	37,378	38,250	38,112	26,654
ecoo	88,340	34,285	**26,986**	58,290	38,863	38,449	27,150
human	271,677	109,962	**79,272**	54,678	117,396	−(*m*)	79,632
kegg	25,319	17,427	10,242	504*K*	12,554	10,141	**8,790**
mtbrv	67,214	30,491	**20,576**	41,689	29,627	29,324	20,876
nasa	39,235	20,976	18,324	**5,307**	21,894	18,913	13,076
vchocyc	66,437	30,182	**20,366**	26,330	29,310	28,988	20,690
xmark	42,560	23,814	16,474	16,434	20,596	21,161	**14,102**

the specified number of edges, we randomly pick two nodes and connect them with an edge from the lower to higher ranked node. Since we randomly select the neighbors of the nodes, it is very unlikely to choose two nodes that are already connected among millions of nodes. Therefore, clustering coefficient is zero in these graphs. SPAN was not able to scale to 50 million nodes for computing effective diameter, so we show them with question mark.

6.2 Performance comparison with graph indexes

Before studying the sensitivity of GRAIL to various parameters and before evaluating the effectiveness of the various search strategies and optimizations, we compare GRAIL's performance with existing state-of-the-art graph reachability indexes, as well as the baseline search methods.

Based on our experiments (outlined later), the default strategy for GRAIL is to use randomized pairs traversal to construct the labeling (with $d = 2$ for small sparse graphs, $d = 3$ for small dense graphs and $d = 5$ for large graphs). Furthermore, GRAIL uses the topological level and positive cut filters and does not maintain exception lists. It uses DFS with recursive pruning as the default search strategy. We denote by GRAIL* the basic approach, i.e., randomized traversals, with no level or positive cut filters, and without exceptions lists (using DFS for recursive search).

6.2.1 Small real data sets: sparse and dense

Tables 7, 8 and 9 show the index construction time, query time and index size for the small, sparse, real data sets. Tables 10, 11 and 12 give the corresponding values for the small, dense, real data sets. The last column in Tables 9 and 12 shows the number of reachable query node pairs, called *positive queries*, out of the 100K test queries; the query node pairs are sampled randomly from the graphs, and the small counts are reflective of the sparsity of the graphs. The best results are shown in bold.

Small sparse graphs: On the sparse data sets (see Table 7), GRIPP has the smallest construction time among all indexing methods, though GRAIL (with $d = 2$ traversals) is a close second. PathTree is typically an order of magnitude slower than GRAIL, and the other methods are even more expensive

Table 9 Small sparse graphs: query time (ms)

Data set	GRAIL	HLSS	INT	Dual	PT	3HOP	GRIPP	DFS-L	BBFS-L	#Pos.Q
agrocyc	20.83	71.10	162.82	70.93	**8.42**	122.63	70.95	34.26	142.31	133
amaze	16.46	99.20	103.05	62.91	**7.08**	50.51	73.21	693.76	196.66	17,259
anthra	20.61	70.66	160.67	65.50	**8.26**	117.15	65.77	31.06	143.91	97
ecoo	20.74	74.35	161.41	68.22	**8.07**	120.40	74.94	35.03	143.47	129
human	22.69	80.36	237.73	76.57	**16.93**	−(m)	57.82	32.31	165.34	12
kegg	17.95	106.88	104.06	64.11	**7.25**	52.13	72.27	1,086.81	265.52	20,133
mtbrv	18.11	78.23	145.61	69.46	**7.42**	106.21	80.14	34.56	128.61	175
nasa	19.93	90.78	129.52	63.20	**7.85**	76.74	165.40	92.37	135.26	562
vchocyc	17.86	68.43	145.03	62.64	**7.22**	105.60	77.03	34.27	127.60	169
xmark	30.20	91.00	122.24	76.02	**7.50**	100.40	89.79	321.22	164.66	1,482

Table 10 Small dense graphs: construction time (ms)

Data set	GRAIL	HLSS	INT	Dual	PT	3HOP	GRIPP
arXiv	29.71	–(t)	20,097.87	386,701.59	9,770.60	8,854,231.00	**22.57**
citeseer	**46.82**	117,865.33	6,712.07	30,213.80	710.29	110,783.48	115.28
go	20.05	68,030.83	1,122.11	7,491.06	219.43	29,578.30	**4.88**
pubmed	**35.05**	142,844.19	5,043.40	25,832.03	768.78	293,161.20	181.80
yago	**24.60**	28,259.51	2,593.61	5,329.56	506.53	30,602.29	44.95

Table 11 Small dense graphs: index size (Num. Entries)

Data sets	GRAIL	HLSS	INT	Dual	PT	3HOP	GRIPP
arxiv	60,000	−(t)	145,668	14,057,239	86,855	**47,472**	133,414
citeseer	107,200	114,088	142,632	30,615,323	91,820	**51,035**	88,516
go	67,930	60,287	40,644	11,000,662	37,729	27,764	**26,722**
pubmed	90,000	102,946	181,260	15,040,251	107,915	**54,531**	80,056
yago	66,420	57,003	57,390	4,371,065	39,181	**27,038**	84,784

Table 12 Small dense graphs: query time (ms)

Data set	GRAIL	HLSS	INT	Dual	PT	3HOP	GRIPP	DFS-L	BBFS-L	#Pos.Q
arXiv	380.94	−(t)	269.64	80.42	**24.73**	355.36	4,041,302.42	6,599.51	665.56	15,459
citeseer	64.04	327.45	222.12	80.58	**24.61**	184.21	9,064.53	203.80	256.54	388
go	30.79	274.64	148.14	77.01	**11.67**	105.23	2,015.98	106.07	164.00	241
pubmed	70.07	307.48	244.04	76.79	**21.85**	179.81	7,551.03	202.00	190.14	690
yago	19.64	256.72	174.86	64.91	**14.18**	117.08	1,122.39	29.86	161.17	171

(several orders of magnitude slower). 3HOP could not run on `human`, since it exhausted the memory (denoted –(m)). INT and GRIPP have the smallest index size (see Table 8). The other methods have comparable index sizes, though GRAIL has the largest number of entries (about 3 times larger than INT).

In terms of query times on the small sparse graphs (see Table 9), PathTree is the best, with GRAIL in the second place, typically being 2–4 times slower. On these small sparse data sets, it is worth noting that, with few exceptions, DFS-L (DFS with level filter) is faster than all indexing methods other than PathTree and GRAIL. The exceptions are `amaze`, `kegg` and `xmark`, where DFS-L is not as effective; on these graphs, BBFS-L is able to improve the query times by a factor of 2–4. The reason why DFS-L suffers in these graphs is the graph topology. These graphs have a central node which has a very high in-degree and out-degree; therefore, in most of the

Table 13 Large real graphs: construction time (ms) and index size

Data set	Construction time (ms)		Index size	
	GRAIL	GRIPP	GRAIL	GRIPP
cit-patents	64,676.35	**41,453.16**	60,396,288	**33,037,894**
citeseer	**5,791.04**	234,166.72	11,103,152	**624,564**
citeseerx	**58,746.74**	1,174,080	104,646,416	**30,022,520**
go-uniprot	89,821.39	**41,851.86**	111,487,296	**69,540,470**
uniprot22m	2,767.63	**2,642.50**	11,168,108	**3,190,884**
uniprot100m	**31,919.36**	33,863.25	112,611,065	**32,174,586**
uniprot150m	**49,355.86**	59,986.65	175,263,200	**50,075,196**

Table 14 Large real graphs: query times (ms)

Data set	GRAIL	GRIPP	DFS-L	BFS-L	BBFS-L	#Pos.Q
cit-patents	**1,369.02**	31,177,982	25,774.05	3,407.33	5,064.83	39
citeseer	27.04	362.91	**25.749**	86.10	157.49	0
citeseerx	**113.43**	27,932.54	119,260.48	4,372.09	4,162.48	239
go-uniprot	**31.06**	4,864.87	37.78	157.56	271.21	0
uniprot22m	**22.65**	279.94	24.52	120.32	174.13	0
uniprot100m	**59.35**	496.59	63.47	155.44	220.87	0
uniprot150m	**52.66**	530.61	71.55	145.55	238.51	0

cases, a DFS has to scan the children of that central node to arrive the target. Whereas BBFS can terminate the search without scanning the children of the central node. Given the fact that the baseline graph search methods have no construction time or indexing size overhead, they are quite attractive for these small data sets. A more detailed comparison among DFS, BFS and BBFS (with and without the level filter) appears in Tables 18, 19 and 20.

Small dense graphs: On the small dense data sets, GRAIL (with $d = 3$) has the smallest index construction times (see Table 10), being up to 6 times faster than the closest rival GRIPP. Other methods are orders of magnitude slower, and HLSS could not run on `arxiv`. The index size (see Table 11) is comparable for all methods except DUAL. 3HOP has the smallest index.

On these small dense graphs, PathTree is still the fastest indexing method, with GRAIL in the second place, being typically 2–3 times slower than PathTree (see Table 12). Once again, DFS-L is comparable to the other indexing methods. On `arXiv`, BBFS-L is able to remedy the shortcomings of DFS-L. It is also worth noting that GRIPP's query performance acutely deteriorates as the average degree increases. One can conclude that for the small dense graphs, while pure search-based methods are quite acceptable, indexing does deliver significant benefits in terms of query time performance.

6.2.2 Large data sets: real and synthetic

Large real graphs: Table 13 shows the construction time and index size on the large real graphs. On these graphs, with the exception of GRAIL and GRIPP, none of the other indexing methods were able to run. PathTree aborted with a memory limit error (–(m)) on `cit-patents` and `citeseerx`, and it exceeded the 20 M ms time limit (–(t)) for the other data sets. It was able to run only on `citeseer` data (130,406 ms for construction, and the index size was 2,360,732 entries). While GRIPP's index size is smaller than GRAIL's, its construction time can be up to 40 times slower than GRAIL. The main index construction of GRIPP is linear on the number of edges; however, by default, it also extracts a list of special nodes (i.e., stop nodes) to speedup the querying. Therefore, in some cases such as in `citeseer`, its construction time does not reflect its linear computational complexity.

From the query times (see Table 14), we can observe that GRAIL can easily scale to large data sets (the only limitation being that it does not yet process disk-resident graphs). We can see that GRAIL outperforms GRIPP by orders of magnitude, and it is faster than BFS-L (the best among the search-based methods) by 3–40 times on the denser graphs: `go-uniprot`, `cit-patents` and `citeseerx`. On the other data sets, which are very sparse, GRAIL is still the winner, while pure DFS is the close second. On these graphs, we use $d = 2$ traversals for GRAIL. It is worth noting that

Table 15 Large synthetic graphs: construction time (ms) and index size

Size	Deg.	Construction time (ms)			Index size		
		GRAIL	GRAIL*	GRIPP	GRAIL (M)	GRAIL* (M)	GRIPP (M)
rand10m	2	**52,782.9**	53,368.0	585,639.3	70	**40**	**40**
	5	259,631.1	244,291.0	**48,338.5**	160	**100**	**100**
	10	433,898.4	412,623.3	**80,471.1**	160	**100**	200
rand100m	2	336,722.2	**313,333.4**	9,920,415.0	350	**200**	**200**
	5	1,661,943.6	1,458,824.1	**292,427.0**	800	**500**	**500**

Table 16 Large synthetic graphs: query times (ms)

Size	*Deg.*	GRAIL	GRAIL*	GRIPP	DFS-L	BFS-L	BBFS-L	#Pos.Q
rand10m	2	**174.8**	259.8	1,955.9	326.3	266.9	476.4	0
	5	6,102.9	**5,764.5**	54,212,516.0	75,732.8	13,011.2	17,449.6	17
	10	1,523,248.0	**1,417,776.5**	−(t)	35,372,076.0	2,815,736.7	1,734,676.1	5,522
rand100m	2	**229.0**	326.7	2,646.6	532.0	343.0	603.4	0
	5	9,281.9	**7,241.1**	-(t)	130,050.5	16,229.6	23,115.3	6

PathTree took 47.4 ms for querying on the sparser `citeseer` data, which is still 2 times slower than GRAIL. It is also interesting that GRIPP is still 10 times slower than GRAIL; however, it uses one-third of index size. If memory is an issue, one should definitely choose with DFS with a simple topological filter on such sparse graphs.

Large synthetic graphs: We also tested the scalability of GRAIL on the large synthetic graphs, which have 10M and 50M nodes, with different average degrees: 2, 5 and 10. We used 2 traversals for GRAIL when the average degree is 2, and 5 traversals otherwise. Table 15 shows the construction time and index sizes. Once again, none of the indexing methods other than GRAIL and GRIPP could handle these large graphs. PathTree too aborted on all data sets, except for `rand10m2x` with avg. degree 2; it took 526,004.7ms for construction, and its index size was 69,378,979 entries). The table also includes the performance of GRAIL*, the basic approach without any optimizations. We see that GRIPP construction time and index size are smaller for dense graphs. Because its construction time is linear on the number of edges and GRIPPs index size is equal to twice the number of edges. However, GRIPPs construction is again an order of magnitude slower for sparser graphs. (the reason is discussed above) Looking at the query times (Table 16), GRAIL is orders of magnitude faster than GRIPP. In fact, when querying on `rand10m10x`, GRIPP exceeded the 20 M ms time limit. Note that DFS also exceeded the time limit in that data set. We can see that for these data sets GRAIL can be orders of magnitude faster than search-based methods. However, for very dense graphs such as `rand10m10x`, BBFS gets closer to GRAIL. PathTree ran only on rand10m2x, with query time 211.7 ms, where GRAIL took 174.8 ms. It is interesting to note that for these large, and sparse synthetic graphs, the positive cut and level filters can slightly slowdown the query times for GRAIL, since the basic GRAIL* approach can be about 5–7% faster. Once again, we conclude that GRAIL is the most scalable reachability index for large graphs, especially with increasing density. The fastest index PathTree does not scale to large graphs, and the scalable index GRIPP cannot provide fast querying with increasing density. Indeed, even pure search methods with level filters are preferable to them.

Synthetic scale-free graphs: Additionally, we generated some random scale-free graphs using Albert–Barabasi preferential attachment network growth model [2] to examine the behavior of the leading methods on graphs that have power-law degree distribution. We generated 3 graphs of 10,000 nodes with average degrees of 2, 5 and 15, and 2 graphs of 100,000 nodes with average degrees of 2 and 5.

This particular set of experiments were performed on a system that has four 2.5 Ghz processors with 4 GB memory. Table 17 shows the comparison between GRAIL, GRIPP and PathTree. GRIPP is still fast at indexing, but even for average degree 2, it provides the worst query time. It gets drastically worse as the density increases. PathTree provides the fastest querying once it manages to construct the index; however, it is not scalable since it suffers memory problems as the density increases (e.g., for 100,000 node graph with average degree 5). GRAIL is still fast at indexing and querying in all cases. These results generally conform to the results in the previous experiments.

Table 17 Synthetic scale-free graphs

Data set			Construction time (ms)			Query time (ms)			
Nodes	Deg	PosQ	GRAIL	GRIPP	PathTree	GRAIL	GRIPP	PathTree	DFS
10000	2	353	11.39	6.49	281.11	23.11	4026.49	9.14	209.35
10000	5	4,287	16.22	12.13	3,537.25	117.24	561,482.62	34.16	2,561.99
10000	15	21,222	48.41	34.14	38,716.01	658.91	−(t)	67.91	16,251.73
100000	2	94	169.86	81.45	11,581.34	46.86	16,529.73	39.48	521.98
100000	5	1,842	206.82	156.96	−(m)	511.88	−(t)	−(m)	12,422.07

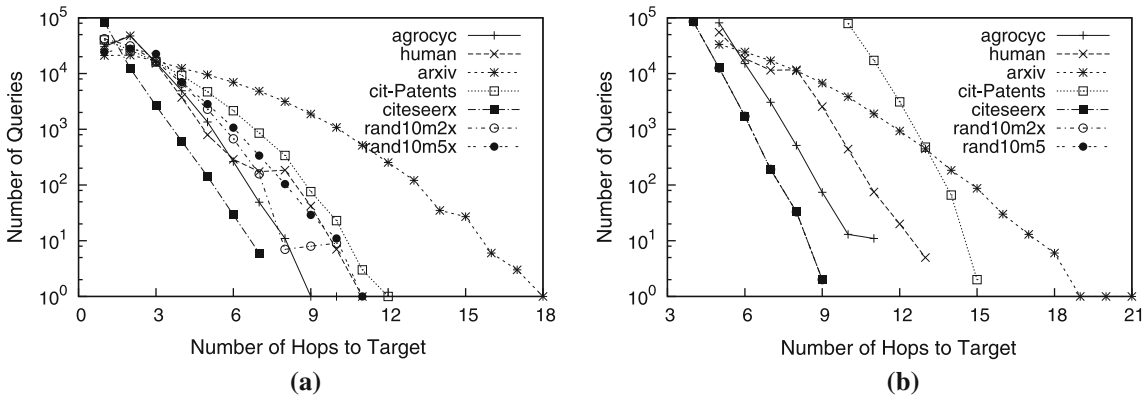

Fig. 9 Reachability queries: **a** positive, **b** deep positive distance

6.3 Graph search strategies: baseline methods

We saw previously that pure graph search methods like DFS, BFS and bidirectional BFS can be very effective in answering reachability queries, especially for the smaller graphs. In this section, we conduct a detailed evaluation of these strategies—the baseline methods—with and without the level filter.

We show results on three sets of queries. As one set we use the same 100K random query pairs used in all of the results above. However, since the graphs are very sparse, the vast majority of these pairs are not reachable. As an alternative, we generated 100 K reachable pairs by simulating a random walk (start from a randomly selected source node, choose a random child with 99% probability and proceed, or stop and report the node as target with 1% probability). Finally, we generated an additional set of 100K deep positive queries, where by deep we mean longer path lengths between the source and destination nodes in the query. The frequency distribution for the number of hops between source and target nodes for the positive queries and deep positive queries is plotted in Fig. 9. We can see the random positive queries have hop lengths from 1 to 18, but the deep positive pairs start and end at longer hop lengths. For example, for `cit-patents`, the hop length ranges from 1 to 12, but for the deep positive queries, the hop length ranges from 10 to 15.

We compare the query time performance of depth-first, breadth-first and bidirectional breadth-first approaches both with and without the topological level filter in Tables 18, 19 and 20. We use DFS, BFS and BBFS for the three search strategies without the level filter, and DFS-L, BFS-L and BBFS-L with the level filter.

Comparing the query times on the random query pairs (Table 18), we can observe that the simple topological level filter is extremely effective in pruning. It improves the performance of all the search methods. On the small graphs, the improvement is typically 2–4 times, whereas on the large graphs the improvement can be up to a factor of 10. One can also observe that DFS-L is invariably the preferred method for the sparse graphs. However, there are a few exceptions, such as `amaze`, `kegg`, `xmark` and `arxiv`, where BBFS-L can be an order of magnitude faster. The distinction of these graphs is their smaller effective diameter. On the large graphs, BFS-L seems to be the preferred method, since it is either the fastest method (by a factor of 2 or higher) or is not very far from the best method. BBFS-L is a close second.

On the positive and deep positive queries (Tables 19, 20), the effectiveness of the level filter is not very much. This is because of the fact that these are all reachable pairs of nodes and cannot be filtered out. Thus, the overhead of applying the level check consistently slows down the query times for all the three search methods, with

Table 18 Query time comparison of the baseline methods: random queries

Data set	DFS	DFS-L	BFS	BFS-L	BBFS	BBFS-L
agrocyc	43.68	**33.06**	105.23	76.35	141.32	138.65
amaze	1,737.61	692.88	1,844.58	406.93	283.17	**191.74**
anthra	38.79	**29.83**	100.68	71.65	139.55	139.62
ecoo	50.29	**33.83**	110.59	78.01	142.37	141.85
human	34.09	**32.43**	93.27	80.39	156.31	164.86
kegg	2,150.63	1,085.38	2,049.80	606.48	388.31	**267.29**
mtbrv	54.70	**33.32**	117.74	77.56	126.74	125.47
nasa	131.12	93.12	190.31	**85.47**	142.67	132.49
vchocyc	55.43	**33.80**	104.10	64.50	126.68	124.34
xmark	388.60	324.93	370.84	214.58	204.89	**158.41**
arxiv	13,237.37	6,816.73	10,635.92	1,513.44	3,491.59	**664.77**
go	121.75	**106.21**	161.35	121.39	179.10	164.64
pubmed	323.84	207.70	443.40	**94.84**	202.40	188.04
yago	116.43	**29.52**	133.26	90.70	152.03	162.44
cit-Patents	38,616.12	30,317.47	40,983.66	**3,403.68**	7,997.03	5,045.11
citeseer	46.20	**25.97**	101.31	87.58	139.00	153.54
citeseerx	224,954.60	153,682.93	259,932.93	4,428.25	6,098.53	**4,218.07**
go-uniprot	**37.87**	156.11	268.58	372.05	320.92	235.17
rand10m2x	553.33	391.02	625.60	**266.27**	537.53	477.06
rand10m5x	91,262.17	74,963.43	96,374.10	**12,598.14**	29,365.89	17,110.69

Table 19 Query time comparison of the baseline methods: positive queries

Data set	DFS	DFS-L	BFS	BFS-L	BBFS	BBFS-L
agrocyc	392.27	456.49	226.55	296.55	**150.49**	176.07
human	924.84	933.10	323.98	528.19	**182.20**	255.21
arxiv	1,390.69	1,399.79	858.13	1,006.30	**240.03**	296.98
citeseerx	79,843.72	80,639.62	2,133.04	3,160.10	**147.55**	183.47
cit-Patents	6,851.40	6,859.36	2,852.23	3,623.24	**605.37**	944.77
go-uniprot	**151.69**	168.31	221.77	243.11	86,679.75	2,058,323.75
rand10m2x	430.95	461.72	299.84	369.90	**292.26**	384.79
rand10m5x	49,874.03	50,063.26	3,570.48	4,738.75	**1,106.92**	1,627.96

Table 20 Query time comparison of the baseline methods: deep positive queries

Data set	DFS	DFS-L	BFS	BFS-L	BBFS	BBFS-L
agrocyc	699.56	710.95	1,306.06	1,785.99	**299.60**	387.30
human	1,359.66	1,263.17	2,105.32	3,164.45	357.66	**309.99**
arxiv	1,084.50	1,010.98	2,188.39	2,348.89	**380.77**	503.91
citeseerx	1,510,195.37	1,481,738.00	375,087.90	471,321.15	**9,488.83**	17,209.19
cit-Patents	50,584.21	49,820.53	96,294.84	128,953.70	**11,692.85**	18,848.20
go-uniprot	**173.61**	188.34	256.48	290.80	184,129.15	3,214,053.00
rand10m2x	**1,069.12**	1,120.13	1,413.06	1,801.97	1,124.22	1,512.55
rand10m5x	174,483.20	174,269.43	213,657.48	268,942.18	**62,386.87**	81,867.81

Table 21 Query time comparison of the GRAIL methods

Data set	d	GRAIL*	GRAIL*+LF	GRAIL*+BI	GRAIL*+BILF	GRAIL−LF	GRAIL	GRAIL+BI	GRAIL+BILF
agrocyc	2	66.76	66.08	151.75	151.62	51.38	**21.99**	151.39	150.05
amaze	2	818.53	834.61	340.41	318.06	25.17	**15.81**	135.78	136.08
anthra	2	62.50	62.23	148.21	146.85	49.17	**22.37**	153.76	154.35
ecoo	2	68.52	64.44	175.18	158.57	54.63	**22.95**	156.08	154.04
human	2	86.15	84.44	160.19	161.84	64.03	**23.45**	161.95	177.25
kegg	2	1105.01	1079.61	503.31	496.06	25.60	**17.57**	150.93	159.23
mtbrv	2	61.40	58.80	151.47	151.52	47.42	**20.40**	148.28	165.92
nasa	2	37.75	35.68	176.42	146.64	35.70	**21.98**	147.30	146.45
vchocyc	2	60.00	57.74	156.44	155.75	46.43	**19.85**	138.99	137.54
xmark	2	97.61	88.75	180.60	176.62	42.46	**28.04**	148.17	165.71
arxiv	4	457.89	436.09	1,149.78	1,126.67	369.36	**331.72**	732.75	718.59
go	3	54.55	50.27	159.23	142.51	52.65	**34.11**	141.77	144.07
pubmed	3	89.56	85.77	216.76	215.18	94.07	**66.23**	219.38	219.36
yago	3	50.29	45.62	216.07	213.03	53.95	**19.10**	221.40	233.04
cit-Patents	5	1,425.87	**1,363.89**	1,995.60	1,982.56	1,537.45	1,412.75	2,021.37	1,999.32
citeseer	2	99.99	93.29	224.15	225.36	105.15	**26.30**	248.23	245.85
citeseerx	5	8,173.68	8,259.32	2,583.46	2,381.76	193.41	**115.42**	1,044.88	1,044.86
go-uniprot	2	163.10	131.59	465.90	507.32	171.82	**29.63**	489.76	469.97
rand10m2x	3	222.25	210.52	338.70	347.04	234.54	**155.93**	350.90	344.73
rand10m5x	5	6,479.56	6,322.14	7,914.99	6,969.14	6,196.16	**5,990.32**	7,446.50	7,376.99

some exceptions. On these (deep) positive queries, BBFS is the best overall method, being 2–100 times faster than alternative methods. The only exception is `go-uniprot`, where DFS/BFS does much better. This graph has around seven million roots that are connected to the remaining 30 K nodes. Bidirectional search is slow in this graph because the reverse search that starts from the target node has to explore many irrelevant roots before finding a solution.

We can conclude that for small graphs, DFS-L is the best overall method, whereas for the large graphs BFS-L is the best. However, BBFS is the clear winner if one expects more positive queries. It is for this reason that BBFS-L is the fastest on data sets like `kegg`, `amaze` and `xmark` on the 100 K random queries, since there are a relatively larger number of positive pairs in those query sets.

6.4 GRAIL: effect of parameters and optimizations

In this section, we study the behavior of GRAIL under optimizations like the level and positive cut filter, as well as the choice of the graph search strategy (DFS/BFS/BBFS). We also evaluate the effectiveness of maintaining exceptions lists and of the various traversal strategies to construct the interval labels.

6.4.1 Effect of optimizations

We first consider the effect of the filters in combination with the recursive graph search strategy on the random, positive and deep positive query pairs. We compared eight variations of GRAIL, which are shown in Tables 21, 22 and 23. Here, GRAIL is the version which uses positive cut filter, whereas GRAIL* is the version without this filter. The suffix of LF is used when level filter is used, and BI is used when a bidirectional search is performed instead of DFS. Thus, GRAIL is the approach that uses positive cut and the level filter, whereas GRAIL-LF is with level filter removed, GRAIL+BI uses bidirectional BFS instead of DFS, and GRAIL+BILF uses both BBFS and level filter. Similar extensions for GRAIL* show the variants of the basic approach, which uses no optimizations, and GRAIL*+LF uses the level filter. The column d represents how many traversals are used to index that graph.

It is clear that the positive cut filter is extremely effective, since GRAIL variants are invariably superior to the GRAIL* counterparts, by up to 100 times in some cases. The positive cut filter is very helpful for positive queries since if at any point of the search a node contains the guaranteed interval, it is sufficient to terminate the query with positive answer. However, it is also helpful in sparse graphs because most of the reachable pairs can be covered with the interior label set. In some exceptional cases, GRAIL* is better than GRAIL, when the distance from the source to target nodes is high as

Table 22 Query time comparison of the GRAIL methods with positive queries

Data set	d	GRAIL*	GRAIL*+LF	GRAIL*+BI	GRAIL*+BILF	GRAIL−LF	GRAIL	GRAIL+BI	GRAIL+BILF
agrocyc	2	479.69	561.59	325.44	330.82	**36.47**	38.55	150.99	159.66
human	2	1,377.98	1,163.86	745.24	748.22	**35.84**	37.39	146.89	148.35
arxiv	4	321.82	330.89	1,070.86	1,222.06	**253.14**	**253.09**	816.84	826.17
cit-Patents	5	3,109.31	3,303.92	3,037.28	2,694.92	3,195.76	3,089.42	**2,452.24**	2,475.95
citeseerx	5	27,400.50	26,402.77	327.15	317.41	123.55	**117.37**	232.21	234.51
go-uniprot	2	**262.75**	275.12	−(t)	−(t)	319.57	327.29	−(t)	−(t)
rand10m2x	3	550.03	549.80	775.11	842.06	546.64	**522.34**	685.14	698.08
rand10m5x	5	17,997.88	17,555.28	3,417.46	3,840.27	19,593.36	19,371.47	**3,152.93**	3,175.09

Table 23 Query time comparison of the GRAIL methods with deep positive queries

Data set	d	GRAIL*	GRAIL*+LF	GRAIL*+BI	GRAIL*+BILF	GRAIL−LF	GRAIL	GRAIL+BI	GRAIL+BILF
agrocyc	2	492.98	512.95	1,001.90	840.36	**66.32**	69.89	459.63	467.71
human	2	670.90	709.45	547.53	508.68	**49.64**	50.91	242.98	247.03
arxiv	4	371.42	376.47	2,484.63	2,513.01	**282.28**	292.93	1,512.91	1,488.77
cit-Patents	5	30,969.72	**30,174.54**	46,411.89	45,814.12	33,551.28	33,093.57	43,740.33	44,198.10
citeseerx	5	1,131,971.50	1,090,715.25	63,204.19	62,324.88	3,711.59	**3,529.66**	15,287.94	15,432.34
go-uniprot	2	**290.22**	301.95	−(t)	−(t)	349.56	361.48	−(t)	−(t)
rand10m2x	3	1,294.02	1,323.16	2,363.52	2,343.72	1,195.18	**1,177.69**	1,914.48	1,951.57
rand10m5x	5	64,076.53	**63,784.27**	90,399.75	80,405.13	68,951.63	70,447.32	74,483.97	75,279.82

for `cit-patents` and `rand10m5x` in Table 23. Due to the density of these data sets, positive cut filter only works after some level until which the search compares interior intervals in vain. For instance, for `rand10m5x`, the deep positive query lengths range from 10 to 16, whereas the search utilizes the positive cuts only after depth 5 on average, and thus up to depth 5, the search makes twice the number of comparisons. Even in these cases, it is worth noting that the performance difference between GRAIL and GRAIL*+LF is not high.

We also observe that bidirectional search is not effective when used with GRAIL variants because the overhead of maintaining two queues and comparing twice the number of intervals at every step adds extra cost, which is not offset by additional pruning. We conclude that GRAIL (with positive cut and level filters, and DFS search) is the preferred method.

6.4.2 Exception lists

Table 24 shows the effect of using exception lists. Here, GRAIL is the default GRAIL strategy with reversed pairs traversals, level and positive cut filters, and without exception lists, GRAIL* is the basic approach without any optimizations, and GRAIL*+E is the basic approach with exception lists. Results are shown only for small, real, sparse and dense graphs, since computing the exception lists on the large graphs is too expensive. We used $d = 2$ for sparse and $d = 3$ for dense graphs.

On both the sparse and dense graphs, the construction time for exception lists blows up; GRAIL*+E is 2–3 orders of magnitude slower than GRAIL/GRAIL*. It is interesting to note that for the spare graphs, the number of exceptions is not very large. In fact, the total index size for GRAIL*+E (which is equal to the index size of GRAIL* plus the number of exceptions) can be smaller than the index size for GRAIL (which has to keep additional information for the positive cut and level filters). However, computing the exception lists is very expensive. On the dense graphs, the number of exceptions blows up by over a factor of 100, and thus, the construction time for GRAIL*+E increases even more rapidly. In terms of query time, for the sparse graphs, it is interesting to note that GRAIL is faster than GRAIL*+E by a factor of 2. However, GRAIL*+E is faster than the basic GRAIL* method (sometimes by a factor of 40). This means that using the positive cut and level filters is much more effective than using exception lists, though exception lists can deliver better performance than DFS search. On the other hand, on the dense graphs, the exception lists lead to faster query times, even when compared to positive cut and level filters. For example, on `arXiv`, GRAIL*+E is faster than GRAIL by a factor of 6. However, this comes at the cost of 3 orders of magnitude slowdown in construction times. We can conclude that whereas using exceptions does help in some cases, the

Table 24 GRAIL: effect of exceptions

Data set	Construction time (ms)			Index size				Query time (ms)		
	GRAIL	GRAIL*	GRAIL*+E	GRAIL	GRAIL*	GRAIL*+E	# Exceptions	GRAIL	GRAIL*	GRAIL*+E
agrocyc	18.37	18.33	7615.42	88,788	50,736	79,378	28,642	**20.81**	72.10	53.10
amaze	5.72	5.72	2,221.14	25,970	14,840	45,720	30,880	**16.91**	867.85	25.36
anthra	17.15	17.17	14,521.61	87,493	49,996	65,320	15,324	**20.62**	61.22	52.87
ecoo	20.27	20.36	7,865.13	88,340	50,480	79,576	29,096	**20.89**	64.38	53.86
human	52.75	52.71	15,756.67	271,677	155,244	176,663	21,419	**22.90**	81.84	69.88
kegg	5.89	5.87	2,409.59	25,319	14,468	56,717	42,249	**18.84**	1,137.29	25.81
mtbrv	14.53	14.60	4,764.51	67,214	38,408	57,085	18,677	**18.39**	64.99	46.90
nasa	9.31	9.38	2,952.96	39,235	22,420	191,771	169,351	**20.16**	35.73	30.95
vchocyc	15.12	15.29	4,790.82	66,437	37,964	54,234	16,270	**17.91**	56.55	46.80
xmark	9.87	9.83	7,134.75	42,560	24,320	1,368,326	1,344,006	**30.30**	99.72	37.71
arXiv	29.71	29.68	57,595.03	60,000	36,000	4,203,463	4,167,463	380.94	485.79	**57.56**
citeseer	46.82	47.19	169,422.86	107,200	64,320	8,399,563	8,335,243	64.04	88.66	**58.89**
go	20.05	20.09	14,653.48	67,930	40,758	656,031	615,273	**30.79**	47.69	39.39
pubmed	35.05	34.95	94,894.25	90,000	54,000	5,980,867	5,926,867	70.07	92.96	**54.92**
yago	24.60	24.53	56,401.45	66,420	39,852	2,179,247	2,139,395	**19.64**	49.51	46.08

substantially higher overhead of construction time, and the large size overhead of storing exception lists, do not justify the relative small gains in query times. Furthermore, exceptions could not be constructed on the large real graphs.

6.4.3 Label traversal strategies

We implemented different traversal strategies for interval labeling, as explained in Sect. 3.2. `Random` is the default randomized strategy, whereas `ReversePairs` denotes the randomized pairs strategy. *MaxVol*, *MaxInt*, *MaxAdjVol* and *MaxAdjInt* denote the deterministic methods that prioritize the nodes that have larger volume, minimum interval, adjusted volume and minimum adjusted interval, respectively. For the adjusted methods, we implemented the transitive closure size estimation algorithm in [9] to get the approximate size of the reachable set for each node. In these experiments, we used GRAIL*.

As seen in Table 25, random labeling has the worst performance. Though the differences among the methods are not substantial, they are more pronounced for the large graphs. On the small sparse data sets, deterministic methods provide up to 20% improvement over the randomized labeling. MaxVol gives best results most of the time, closely followed by ReversePairs. We use ReversePairs as the default strategy for GRAIL, since it is usually better for large graphs and does not need the estimation of the transitive closure.

In Table 26, we take a closer look at the impact of traversal strategy by comparing the number of exceptions remaining after performing the traversals. It is interesting to note that Random labeling produces significantly more exceptions compared to other labelings whereas the difference is not that apparent in query time comparison. It also suggests that the query time does not necessarily have to be directly correlated with the number of exceptions remained. For instance, in Table 26, ReversePairs have about 20% more exceptions than MaxAdjInt for arxiv graph; however, its query time is still 20% faster than MaxAdjInts as seen in Table 25.

6.4.4 Number of traversals/intervals (d)

In Fig. 10, we plot the effect of increasing the dimensionality of the index, i.e., increasing the number of traversals d, on three small sparse (`agrocyc`, `amaze`, `ecoo`), two small dense (`arxiv`, `pubmed`), two large real (`cit-patents`, `citeseerx`) and two large synthetic (`rand 10m2x`, `rand10m5x`) graphs. In each plot, we show the query times of GRAIL and GRAIL*. Since GRAIL is usually much faster than GRAIL*, we had to use two different y-axis for some of the results. The y-axis on the left hand side shows the query time of GRAIL, while the one on the right side displays the query time for GRAIL*. Note that the plots for GRAIL and GRAIL* appear similar in most of the graphs even though their query times are in different scales. For instance, in `amaze` GRAIL rises to 26 ms from 19 ms, whereas GRAIL* rises to 940 ms from 840 ms in a very similar manner. It is clear that increasing the number of intervals increases the construction time and index size, while decreasing the query time. However, increasing d does not progressively decrease query times, since at some point, the overhead of checking a larger number of intervals negates the potential reduction in exceptions. In small sparse graphs,

Table 25 Comparison of different traversal strategies in GRAIL: query times (ms)

Data set	d	Random	ReversePairs	MaxVol	MaxInt	MaxAdjVol	MaxAdjInt
agrocyc	2	78.39	72.33	65.45	65.53	**64.92**	**64.93**
anthra	2	66.44	**60.98**	65.55	65.53	65.07	65.67
amaze	2	838.09	872.46	726.80	**713.04**	726.26	717.27
mtbrv	2	73.07	64.57	**62.24**	**62.19**	**62.16**	**62.13**
arxiv	3	535.05	**490.00**	595.52	568.92	673.33	589.49
arviv	4	477.71	**417.76**	495.20	452.59	598.03	497.92
pubmed	3	98.15	92.20	**86.52**	88.90	**86.07**	**86.91**
pubmed	4	89.33	86.32	**79.68**	82.36	80.64	82.48
yago	3	56.25	49.33	**46.66**	47.15	**46.98**	**46.82**
yago	4	53.01	48.32	**45.80**	46.35	**45.86**	46.38
go	3	60.93	46.99	**45.25**	**45.62**	48.86	49.20
go	4	48.19	45.25	**43.50**	**43.26**	46.61	46.45
citeseerx	4	8,905.04	**7,797.91**	7,848.54	7,850.85	8,142.71	8,093.69
citeseerx	5	8,404.13	8,148.96	**7,374.71**	7,455.17	7,942.13	7,680.86
cit-Patents	4	1,694.08	**1,626.33**	1,748.10	1,926.49	1,890.82	1,915.62
cit-Patents	5	1,458.75	**1,350.29**	1,387.13	1,620.15	1,593.68	1,617.17
go-uniprot	4	158.72	137.59	136.71	136.52	136.65	**135.37**
go-uniprot	5	149.08	141.82	**135.72**	**135.62**	**135.89**	**135.50**

Table 26 Comparison of different traversal strategies in GRAIL: number of exceptions remained

Data set	d	Random	ReversePairs	MaxVol	MaxInt	MaxAdjVol	MaxAdjInt
agrocyc	2	1,514,319	**28,642**	29,943	29,943	29,672	29,672
amaze	2	685,366	**30,880**	108,472	108,472	41,280	41,280
anthra	2	78,560	15,324	14,874	14,874	**14,436**	**14,436**
ecoo	2	2,701,160	29,096	28,475	28,475	**27,576**	**27,576**
human	2	2,003,874	**21,419**	23,756	23,746	21,720	21,720
kegg	2	651,196	42,249	100,449	100,449	**36,604**	**36,604**
mtbrv	2	1,612,124	**18,677**	20,240	20,240	19,734	19,734
nasa	2	674,175	**169,351**	225,161	225,161	208,253	208,253
vchocyc	2	961,784	16,270	16,131	16,131	**15,759**	**15,759**
xmark	2	2,298,605	1,344,006	1,376,535	1,376,535	**1,233,041**	**1,233,041**
arxiv	3	4,944,155	4,167,463	3,387,488	3,452,679	3,251,799	**3,174,875**
citeseer	3	12,645,509	8,335,243	6,560,907	6,777,400	**6,547,945**	6,801,282
go	3	1,279,303	615,273	640,928	**608,102**	663,908	662,851
pubmed	3	8,255,165	5,926,867	4,643,569	4,768,787	**4,621,694**	4,714,258
yago	3	4,729,341	2,139,395	**1,054,214**	1,198,431	1,055,417	1,198,552

the minimum query time is obtained at some point between 2 and 4. However, we decided to use $d = 2$ traversals for these graphs since adding each dimension increases the index size significantly, while the gain in query time is comparatively very small. As the graphs get denser, the optimum number of traversals increases. For example, for `arxiv`, we get the minimum query times between 12 and 16, and adding more traversals degrades query performance. Considering this graph has an average degree of 11.12, using 10 traversals seems acceptable. That makes the index size close to the order of the graph size and the query time close to the minimum. However, we used much smaller d in our experiments to be able to compete with other algorithms in terms of index size as well. Therefore, one can choose the value d according to the needs by balancing the index size and query time. On very large sparse graphs such as `citeseerx` and `rand10m2x`, the query times approach to the minimum level after 4 traversals. However, for the denser large graphs, we

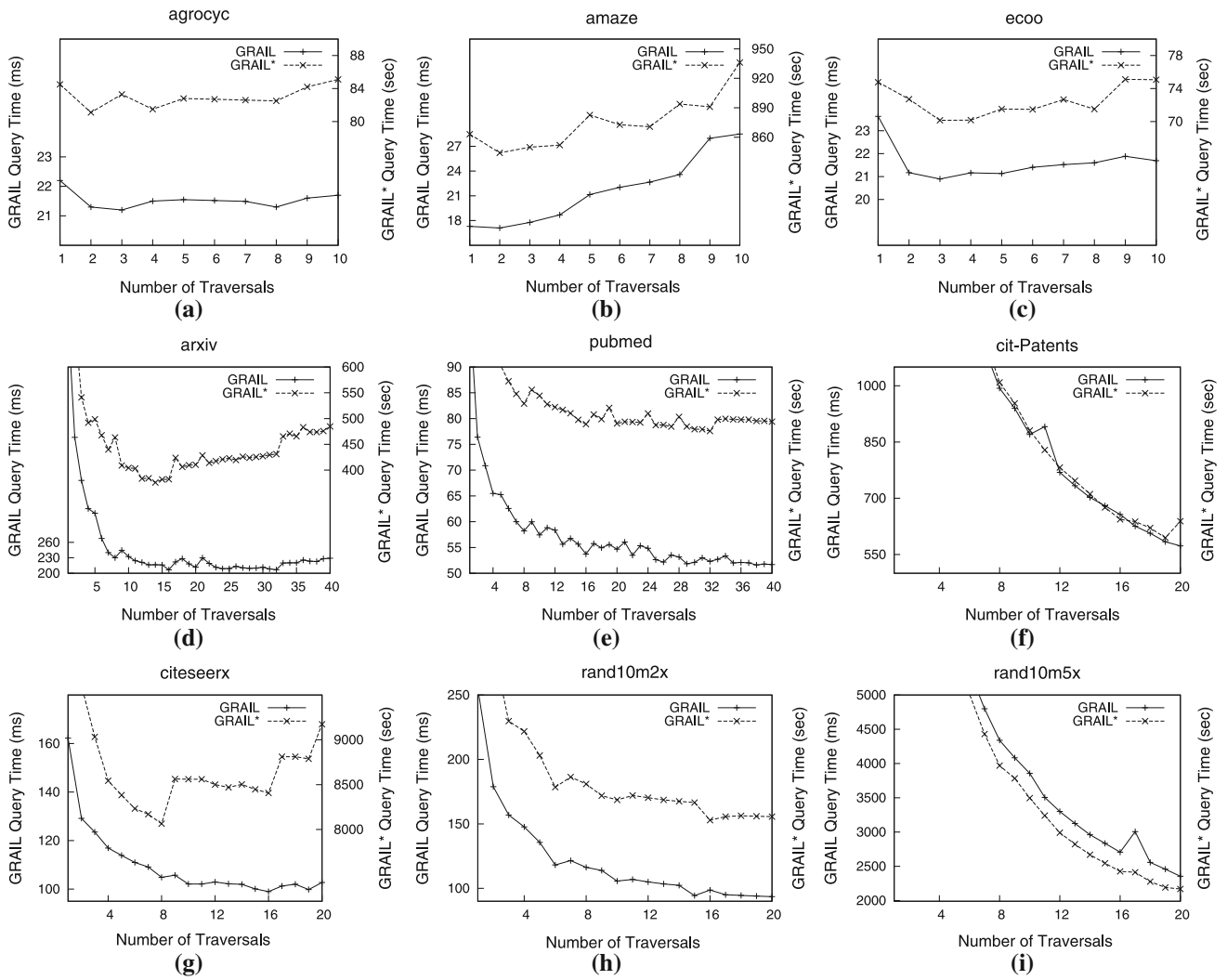

Fig. 10 Effect of increasing number of intervals: **a** `agrocyc`, **b** `amaze`, **c** `ecoo`, **d** `arxiv`, **e** `pubmed`, **f** `cit-patents`, **g** `citeseerx`, **h** `rand10m2x`, **i** `rand10m5x`

see that the query times continue to decrease up to 25 traversals. It is also interesting to see that for these graphs, positive cut filter does not improve query performance and GRAIL has almost the same timing with GRAIL*. For `cit-Patents`, we obtained 1,369 ms with 5 traversals and this plot suggests that it can be improved to 650 ms by increasing its index size by 4 times. Therefore, even in these cases, it may not worth to keep adding traversals unless the memory is not a constraint or the query performance is critical.

Consequently, estimating the number of traversals that minimize the query time, or that optimize the index size/query time trade-off, is not straightforward. However, for any practical benefit, it is imperative to keep the index size smaller than the graph size. This loose constraint restricts d to be less than the average degree. In our experiments, we found out that the best query time is obtained when $d = 5$ or smaller (when the average degree is smaller).

7 Conclusion

We proposed GRAIL, a relatively simple indexing scheme, for fast and scalable reachability testing in very large graphs, based on randomized multiple interval labeling. GRAIL has linear construction time and index size, and its query time ranges from constant to linear time per query. Based on an extensive set of experiments, we conclude that for the class of smaller graphs (both dense and sparse), while more sophisticated methods give a better query time performance, a DFS/BFS search is often good enough, with the added advantage of having no construction time or index size overhead. On the other hand, GRAIL outperforms all existing methods, as well as pure search-based methods on large real graphs; in fact, for these large graphs existing indexing methods are simply not able to scale. Although GRIPP is scalable in indexing, it is not

able to compete even with pure search methods in denser graphs.

As a future work, we plan to address reachability labeling in dynamic graphs. GRAIL seems promising for the problem of dynamic reachability since its labeling has relaxed invariants. The updates on the graph such as edge addition and deletion could be reflected on the labeling via local modifications to the labels. The reachability problem in labeled graphs is also an interesting direction.

References

1. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. SIGMOD Rec. **18**(2), 253–262 (1989)
2. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)
3. Bouros, P., Skiadopoulos, S., Dalamagas, T., Sacharidis, D., Sellis, T.: Evaluating reachability queries over path collections. In: SSDBM, p. 416 (2009)
4. Bramandia, R., Choi, B., Ng, W.K.: On incremental maintenance of 2-hop labeling of graphs. In: WWW (2008)
5. Chen, L., Gupta, A., Kurul, M.E.: Stack-based algorithms for pattern matching on dags. In: VLDB (2005)
6. Chen, Y.: General spanning trees and reachability query evaluation. In: Canadian Conference on Computer Science and Software Engineering, Montreal (2009)
7. Chen, Y., Chen, Y.: An efficient algorithm for answering graph reachability queries. In: ICDE (2008)
8. Cheng, J., Yu, J.X., Lin, X., Wang, H., Yu, P.S.: Fast computing reachability labelings for large graphs with high compression rate. In: EBDT (2008)
9. Cohen, E.: Estimating the size of the transitive closure in linear time. In: 35th Annual Symposium on Foundations of Computer Science, pp. 190–200 (1994)
10. Cohen, E.: Size-estimation framework with applications to transitive closure and reachability. J. Comput. Syst. Sci. **55**(3), 441–453 (1997)
11. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. SIAM J. Comput. **32**(5), 1335–1355 (2003)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2001)
13. Demetrescu, C., Italiano, G.: Fully dynamic transitive closure: breaking through the $O(n^2)$ barrier. In: FOCS (2000)
14. Demetrescu, C., Italiano, G.: Dynamic shortest paths and transitive closure: algorithmic techniques and data structures. J. Discret. Algorithms **4**(3), 353–383 (2006)
15. Dietz, P.F.: Maintaining order in a linked list. In: STOC (1982)
16. Gene Ontology. http://www.geneontology.org/ (2010). Accessed 4 Dec 2010
17. Gene Ontology. Go Database Guide. http://www.geneontology.org/GO.database.shtml#schema_notes (2010). Accessed 4 Dec 2010
18. He, H., Wang, H., Yang, J., Yu, P.S.: Compact reachability labeling for graph-structured data. In: CIKM (2005)
19. Herman, I.L: W3c semantic web faq. http://www.w3.org/2001/sw/SW-FAQ# (2010). Accessed 4 Dec 2010
20. Jagadish, H.V.: A compression technique to materialize transitive closure. ACM Trans. Database Syst. **15**(4), 558–598 (1990)
21. Jin, R., Xiang, Y., Ruan, N., Fuhry, D.: 3-hop: a high-compression indexing scheme for reachability query. In: SIGMOD (2009)
22. Jin, R., Xiang, Y., Ruan, N., Wang, H.: Efficient answering reachability queries on very large directed graphs. In: SIGMOD (2008)
23. King, V., Sagert, G.: A fully dynamic algorithm for maintaining the transitive closure. J. Comput. Syst. Sci. **65**(1), 150–167 (2002)
24. Krommidas, I., Zaroliagis, C.: An experimental study of algorithms for fully dynamic transitive closure. J. Exp. Algorithmics **12**, 16 (2008)
25. Leskovec, J.: Snap Network Analysis Library. http://snap.stanford.edu/snap/index.html (2010). Accessed 4 Dec 2010
26. Roditty, L., Zwick, U.: A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In: STOC (2004)
27. Schenkel, R., Theobald, A., Weikum, G.: HOPI: an efficient connection index for complex XML document collections. In: EBDT (2004)
28. Schenkel, R., Theobald, A., Weikum, G.: Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In: ICDE (2005)
29. Steve Harris, G.: Sparql 1.1 Query Language. http://www.w3.org/TR/sparql11-query/#propertypaths (2010). Accessed 4 Dec 2010
30. Trissl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: SIGMOD (2007)
31. UniProt. http://www.uniprot.org/ (2010). Accessed 4 Dec 2010
32. Wang, H., He, H., Yang, J., Yu, P., Yu, J.X.: Dual labeling: answering graph reachability queries in constant time. In: ICDE (2006)
33. Wu, Z., Eadon, G., Das, S., Chong, E.I., Kolovski, V., Annamalai, M., Srinivasan, J.: Implementing an inference engine for rdfs/owl constructs and user-defined rules in oracle. In: International Conference on Data Engineering, pp. 1239–1248 (2008)
34. Yildirim, H., Chaoji, V., Zaki, M.J.: Grail: scalable reachability index for large graphs. PVLDB **3**(1), 276–284 (2010)
35. Yu, J.X., Lin, X., Wang, H., Yu, P.S., Cheng, J.: Fast computation of reachability labeling for large graphs. In: EBDT (2006)