GRAIL: Scalable Reachability Index for Large Graphs *

Hilmi Yıldırım
Rensselaer Polytechnic Institute
110 8th St.
Troy/NY, USA
yildih2@cs.rpi.edu

Vineet Chaoji
Yahoo! Labs
Bangalore
India
chaojv@yahoo-inc.com

Mohammed J. Zaki
Rensselaer Polytechnic Institute
110 8th St.
Troy/NY, USA
zaki@cs.rpi.edu

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 simply do not scale to very large real-world graphs. We present a very simple, but 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.

1. INTRODUCTION

Given a directed graph $G = (V, E)$ and two nodes $u, v \in V$, a *reachability query* asks if there exists a path from u to v in G. If u can reach v, we denote it as $u \rightarrow v$, whereas if u cannot reach v, we denote it as $u \not\rightarrow v$. Answering graph reachability queries quickly has been the focus of research for over 20 years. Traditional applications include reasoning about inheritance in class hierarchies, testing concept subsumption in knowledge representation systems, and checking connections in geographical information systems. However, interest in the reachability problem revived in recent years with the advent of new applications which have very large graph-structured data that are queried for reachability excessively. The emerging area of Semantic Web is composed of RDF/OWL data which 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 play a role in querying protein-protein interaction networks, metabolic pathways and gene regulatory networks. In general, given the ubiquity of large graphs, there is a crucial need for highly scalable indexing schemes.

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

Proceedings of the VLDB Endowment, Vol. 3, No. 1

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 represents a strongly connected component of the original graph, and each edge represents the fact whether one component can reach another. To answer whether node u can reach v in G, we simply look up 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. Henceforth, we will assume that all input graphs have been transformed into their corresponding DAGs, and thus we will discuss methods for reachability only on DAGs.

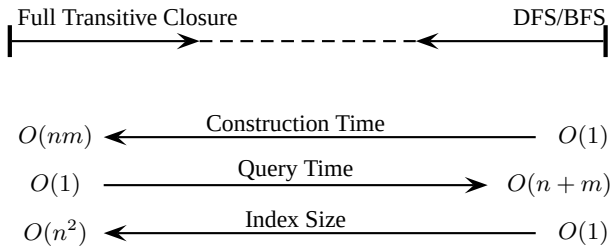
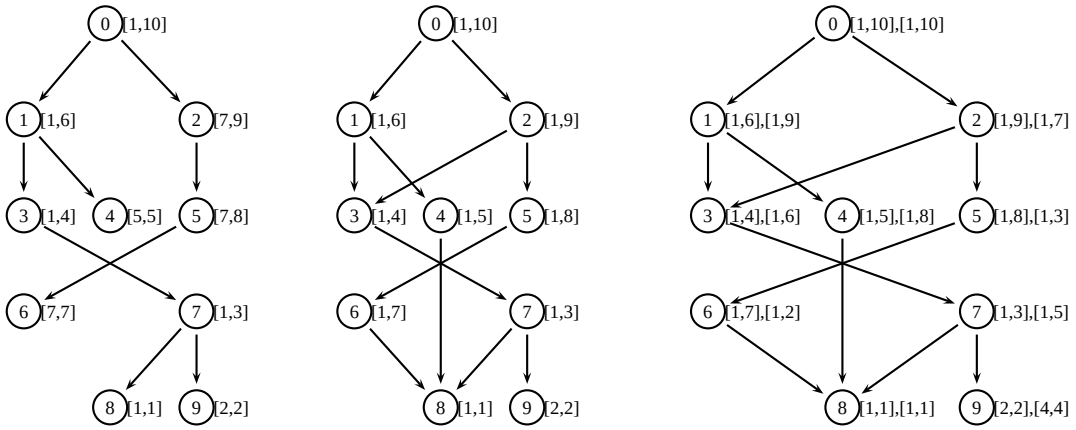


Figure 1: Tradeoff between Query Time and Index Size

There are two basic approaches to answer the reachability queries on DAGs, which lie at the two extremes of the index design space, as illustrated in Figure 1. 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 not yet a single best indexing scheme for DAGs, the reachability problem on trees can be solved effectively by *interval labeling* [11], which takes linear time and space for constructing the index, and provides constant time querying. It labels each node u with a range $L_u = [r_x, r_u]$, where r_u denotes the rank of the node u in a post-order traversal of the tree, where the ranks are assumed to begin at 1, and all the children of a node are assumed to be ordered and fixed for that traversal. Further, r_x de-



(a) Tree (b) DAG: Single Interval (c) DAG: Multiple Intervals

Figure 2: Interval Labeling: Tree (a) and DAG: Single (b) & Multiple (c)

notes the lowest rank for any node x in the subtree rooted at u (i.e., including u). This approach guarantees that the containment between intervals is equivalent to the reachability relationship between the nodes, since the post-order traversal enters a node before all of its descendants, and leaves after having visited all of its descendants. In other words, $u \rightarrow v \iff L_v \subseteq L_u$. For example, Figure 2(a) shows the interval labeling on a tree, assuming that the children are ordered from left to right. It is easy to see that reachability can be answered by interval containment. For example, $1 \rightarrow 9$, since $L_9 = [2,2] \subset [1,6] = L_1$, but $2 \not\rightarrow 7$, since $L_7 = [1,3] \not\subseteq [7,9] = L_2$.

To generalize the interval labeling to a DAG, we have to ensure that a node is not visited more than once, and a node will keep the post-order rank of its first visit. For example, Figure 2(b) shows an interval labeling on a DAG, assuming a left to right ordering of the children. As one can see, interval containment of nodes in a DAG is not exactly equivalent to reachability. For example, $5 \not\rightarrow 4$, but $L_4 = [1,5] \subseteq [1,8] = L_5$. In other words, $L_v \subseteq L_u$ does not imply that $u \rightarrow v$. On the other hand, one can show that $L_v \not\subseteq L_u \implies u \not\rightarrow v$.

In this paper we present a novel, scalable graph indexing approach for very large graphs, called GRAIL, which stands for **G**raph **R**eachability Indexing via R**A**ndomized **I**nterval **L**abeling. Instead of using a single interval, GRAIL employs multiple intervals that are obtained via random graph traversals. We use the symbol d to denote the number of intervals to keep per node, which also corresponds to the number of graph traversals used to obtain the label. For example, Figure 2(c) shows a DAG labeling using 2 intervals (the first interval assumes a left-to-right ordering of the children, whereas the second interval assumes a right-to-left ordering).

The key idea of GRAIL is to do very fast elimination for those pairs of query nodes for whom non-reachability can be determined via the intervals. In other words, if $L_v \not\subseteq L_u$, which can be checked in $O(d)$ time, we immediately return $u \not\rightarrow v$. On the other hand, if successive index lookups fail, reachability defaults to a DFS in the worst-case. The space complexity of our indexing is $O(dn)$, since d intervals have to be kept per node, and the construction time is $O(d(n+m))$, since d random graph traversals are made to obtain those labels. Since d is typically a small constant, GRAIL requires time and space linear in the graph size for index creation. For query answering, the time complexity ranges from $O(d)$ (in cases where non-reachability can be determined using the index), to $O(n+m)$ (in cases where the search defaults to a DFS). GRAIL is thus a light-weight index, that scales to very large graphs, due to its simplicity. 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. In many cases, GRAIL is the only method that can even run on such large graphs.

2. RELATED WORK

As noted above, existing approaches for graph reachability combine aspects of indexing and pure search, trading off index space for querying time. Major approaches include interval labeling, compressed transitive closure, and 2HOP indexing [1, 21, 22, 15, 13, 5, 2, 7, 20, 19, 23, 6, 12], which are discussed below, and summarized in Table 1.

	Construction Time	Query Time	Index Size
Opt. Tree Cover [1]	$O(nm)$	$O(n)$	$O(n^2)$
GRIPP [21]	$O(m+n)$	$O(m-n)$	$O(m+n)$
Dual Labeling [22]	$O(n+m+t^3)$	$O(1)$	$O(n+t^2)$
PathTree [15]	$O(mk)$	$O(mk)/O(mn)$	$O(nk)$
2HOP [7]	$O(n^4)$	$O(\sqrt{m})$	$O(n\sqrt{m})$
HOPI [20]	$O(n^3)$	$O(\sqrt{m})$	$O(n\sqrt{m})$
GRAIL (this paper)	$O(d(n+m))$	$O(d)/O(n+m)$	$O(dn)$

Table 1: Comparison of Approaches: n denotes number of vertices; m, number of edges; $t = O(m-n)$, number of non-tree edges; k number of paths/chains; and d number of intervals.

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 reachability queries, as mentioned above. 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. Thus u is guaranteed to contain all of its children's intervals. Testing reachability is equivalent to deciding whether a list of intervals subsumes another list of intervals. The worst case complexity of this is the same as a full transitive closure.

GRIPP [21] is 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* [22] processes non-tree edges in a different way. After labeling the selected tree, it computes the transitive closure of non-tree edges so that it can answer queries by a constant number of lookups (in Table 1 $t = O(m-n)$ denotes the number of non-tree edges). GRIPP and Dual Labeling thus lie on the opposite sides of the trade-off illustrated in Figure 1.

A chain decomposition approach was proposed in [13] 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 [5], where they proposed a decomposition which can be computed in $O(n^2 + kn\sqrt{k})$ time. Recently, [4] further improved this scheme by using general spanning trees in which each edge corresponds to a path in the original graph. [2] solves a variant of the reachability problem where the input is assumed to a collection of non-disjoint paths instead of a graph.

PathTree [15] is the generalization of the tree cover approach. It extracts the disjoint paths of a DAG, 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 constant time querying and fast construction times, but its index size might get very large on dense graphs (k denotes the number of paths in the decomposition). In a recent paper by the same authors, they proposed 3HOP [14] which addresses the issue of large index size. Although 3HOP has a reduced index size, the construction and query times degraded significantly. Based on our experimentation, PathTree is the best extant method in terms of query time.

The other major class of methods is based on *2HOP Indexing* [7, 20, 19, 23, 6, 12], 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 [7], 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 set-cover problem. Its quartic construction time was improved in [23] by using a geometric approach which produces slightly larger 2HOP cover than obtained in [7]. A divide-and-conquer strategy to to 2HOP indexing was proposed in [20, 19]. HOPI[20] 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. [19], by the same authors, improved the merge phase by changing the way in which cross-edges between subgraphs are processed. [6] 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 [19]. Their approach outperforms existing 2HOP approaches in large and dense datasets.

The HLSS [12] method proposes a hybrid of 2HOP and Interval Labeling. They first label a spanning tree of the graph with interval labeling and extract a remainder graph whose transitive closure is yet to be computed. In the transitive closure of the remainder graph, densest sub-matrices are found and indexed with 2HOP indexing. The problem of finding densest sub-matrices is NP-hard and they proposed a 2-approximation algorithm for it.

Despite the overwhelming interest in static transitive closure, not much attention has been paid to practical algorithms for the dynamic case, though several theoretical studies exist [18, 16, 9]. Practical works on dynamic transitive closure [10, 17] and dynamic 2HOP indexing [3] have only recently been proposed. However, scalability remains a problem. Our focus in this paper is on static indexing; extending GRAIL to the dynamic setting will be considered in the future.

3. THE 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, 22, 21] and path-tree in [15]) 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 single interval per node.

node	**exceptions** (E)	**direct** (E^d)	**indirect** (E^i)
2	$\{1, 4\}$	$\emptyset$	$\{1, 4\}$
4	$\{3, 7, 9\}$	$\{3, 7, 9\}$	$\emptyset$
5	$\{1, 3, 4, 7, 9\}$	$\emptyset$	$\{1, 3, 4, 7, 9\}$
6	$\{1, 3, 4, 7, 9\}$	$\{1, 3, 4, 7, 9\}$	$\emptyset$

Table 2: Exceptions for DAG in Figure 2(b)

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 $u \not\rightarrow v$, as per the theorem below:

THEOREM 1. *If $L_v \not\subseteq L_u$, then $u \not\rightarrow v$.*

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 $u \rightarrow v$, 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 $u \rightarrow v$ 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 $u \rightarrow v$. We conclude that $u \not\rightarrow v$.* ■

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 $u \not\rightarrow v$. We call such a false positive containment an *exception*. For example, in Figure 2(b), 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 $2 \not\rightarrow 1$. 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 Figure 2(c), 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 $2 \not\rightarrow 1$. 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.

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 below.

3.1 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

$$L_u^i = [\ L_u^i[1],\ L_u^i[2]\ ] = [r_x, r_u]$$

While the number of possible labelings is exponential, yet most graphs can be indexed very compactly with 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):
1  foreach i ← 1 to d do
2      r ← 1 // global variable:  rank of node
3      Roots ← {n : n ∈ roots(G)}
4      foreach x ∈ Roots in random order do
5          Call RandomizedVisit(x, i, G)

   RandomizedVisit(x, i, G)  :
6  if x visited before then return
7  foreach y ∈ Children(x) in random order do
8      Call RandomizedVisit(y, i, G)
9  r*_c ← min{L^i_c[1] : c ∈ Children(x)}
10 L^i_x ← [min(r, r*_c), r]
11 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. 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.

Bottom Up: Instead of processing the nodes in topological order from the roots to the sinks, in this strategy we conceptually "reverse the edges" and process the nodes in reverse topological order. With this change, $L_v \not\subseteq L_u \implies v \not\to u$. The bottom-up traversal can be done at random, or in random-pairs.

It is clear that the index construction in GRAIL takes $O(d(n+m))$, corresponding to the d traversals for the graph G. Further, the space complexity is exactly $2dn = O(dn)$, since d intervals are kept per node.

3.2 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 $u \not\to v$, by Theorem 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. The first 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:

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

For example, for the DAG in Figure 2(b), 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 $u \not\to v$. If no, then the containment is not an exception, and we answer $u \to v$. We describe how to construct exceptions lists in Appendix A.

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 exception 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 list so its construction time and index size is linear.

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 exceptions lists are explicitly maintained, either complete or memoized (see Section A): if $v \in E_u$, then GRAIL returns false, otherwise it returns true. Lines 7-10 code the default recursive DFS with pruning. If there exists a child c of u, that satisfies the condition that $L_v \subseteq L_c$, and we check and find that $c \to v$, we can conclude that $u \to v$, and GRAIL returns true (Line 9). Otherwise, if none of the children can reach v, then we conclude that $u \not\to v$, and we return false in Line 10. As an example, let us consider the single interval index in Figure 2(b). 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., $2 \not\to 4$.

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. EXPERIMENTS

We conducted extensive experiments to compare our algorithm with the best existing studies. All experiments are performed in a machine x86_64 Dual Core AMD Opteron(tm) Processor 870 GNU/Linux which has 8 processors and 32G ram. We compared our algorithm with pure DFS (depth-first search) without any pruning, HLSS [12], Interval (INT) [1], Dual Labeling (Dual) [22]), PathTree (PT) [15] and 3HOP [14]. The code for these methods was obtained from the authors, though in some cases, the original code had been reimplemented by later researchers.

Based on our experiments for GRAIL we found that the basic randomized traversal strategy works very well, with no significant benefit of the other methods. Thus all experiments are reported only with randomized traversals. Furthermore, we found that exception lists maintenance is very expensive for large graphs, so the default option in GRAIL is to use DFS with pruning.

Note that all query times are aggregate times for 100K queries. We generate 100K random query pairs, and issue the same queries to all methods. In the tables below, we use the notation –(t), and –(m), to note that the given method exceeds the allocated time (10M milliseconds (ms) for small sparse, and 20M ms for all other graphs; $M \equiv million$) or memory limits (32GB RAM; i.e., the method aborts with a `bad-alloc` error).

Dataset	Nodes	Edges	Avg Deg
agrocyc	12684	13657	1.07
amaze	3710	3947	1.06
anthra	12499	13327	1.07
ecoo	12620	13575	1.08
human	38811	39816	1.01
kegg	3617	4395	1.22
mtbrv	9602	10438	1.09
nasa	5605	6538	1.17
vchocyc	9491	10345	1.09
xmark	6080	7051	1.16

Table 3: Small Sparse Real

Dataset	Nodes	Edges	Avg Deg
arxiv	6000	66707	11.12
citeseer	10720	44258	4.13
go	6793	13361	1.97
pubmed	9000	40028	4.45
yago	6642	42392	6.38

Table 4: Small Dense Real

Dataset	Nodes	Edges	Avg Deg
citeseer	693947	312282	0.45
citeseerx	6540399	15011259	2.30
cit-patents	3774768	16518947	4.38
go-uniprot	6967956	34770235	4.99
uniprot22m	1595444	1595442	1.00
uniprot100m	16087295	16087293	1.00
uniprot150m	25037600	25037598	1.00

Table 5: Large Real

Dataset	Nodes	Edges	Avg Deg
rand10m2x	10M	20M	2
rand10m5x	10M	50M	5
rand10m10x	10M	100M	10
rand100m2x	100M	200M	2
rand100m5x	100M	500M	5

Table 6: Large Synthetic

4.1 Datasets

We used a variety of real datasets, both small and large, as well as large synthetic ones, as described below.

Small-Sparse: These are small, real graphs, with average degree less than 1.2, taken from [15], and listed in Table 3. `xmark` and `nasa` are XML documents, and `amaze` and `kegg` are metabolic networks, first used in [21]. Others were collected from BioCyc (`biocyc.org`), a collection of pathway and genome databases. `amaze` and `kegg` have a slightly different structure, in that they have a central node which has a very large in-degree and out-degree.

Small-Dense: These are small, dense real-world graphs taken from [14] (see Table 4). `arxiv` (`arxiv.org`), `citeceer` (`citeseer.ist.psu.edu`), and `pubmed` (`www.pubmedcentral.nih.gov`) are all citation graph datasets. GO is a subset of the Gene Ontology (`www.geneontology.org`) graph, and yago is a subset of the semantic knowledge database YAGO (`www.mpi-inf.mpg.de/suchanek/downloads/yago`).

Large-Real: To evaluate the scalability of GRAIL on real datasets, we collected 7 new datasets which have previously not been been used by existing methods (see Table 5). `citeseer`, `citeseerx` and `cit-patents` are citations networks in which non-leaf nodes are expected to have 10 to 30 outgoing edges on average. However `citeseer` is very sparse because of data incompleteness. `citeseerx` is the complete citation graph as of March 2010 from (`citeseerx.ist.psu.edu`). `cit-patents` (`snap.stanford.edu/data`) includes all citations in 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 (`www.uniprot.org`) database, the universal protein resource. Gene ontology is a directed acyclic graph of size around 30K, 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 datasets are obtained from the RDF graph of UniProt. `uniprot22m` is the subset of the complete RDF graph which 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 graphs ever considered for reachability testing.

Large-Synthetic: To test the scalability with different density setting, we generated random DAGs, ranging with 10M and 100M nodes, 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 the specified number of edges, we randomly pick two nodes and connect them with an edge from the lower to higher ranked node.

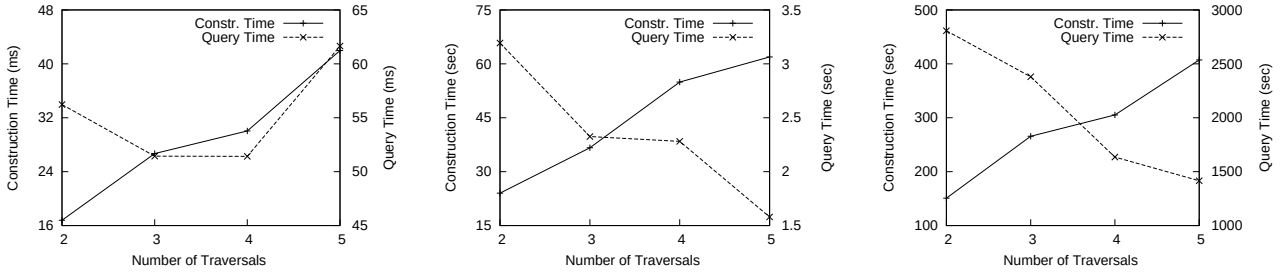
Dataset	GRAIL	HLSS	INT	Dual	PT	3HOP
agrocyc	16.13	12397	5232	11803	279	142K
amaze	3.82	703K	3215	4682	818	2304K
anthra	16	11961	4848	11600	268	142K
ecoo	16	12711	5142	12287	276	146K
human	71	135K	47772	134K	822	– (t)
kegg	3.8	1145K	3810	6514	939	3888K
mtbrv	12	3749	2630	3742	208	86291
nasa	6.3	1887	811	999	126	33774
vchocyc	12	4500	2541	3910	201	85667
xmark	7.5	70830	1547	1719	263	151856

Table 7: Small Sparse Graphs: Construction Time (ms)

4.2 Small Real Datasets: Sparse and Dense

Tables 7, 8, and 9 show the index construction time, query time, and index size for the small, sparse, real datasets. Tables 10, 11, and 12 give the corresponding values for the small, dense, real datasets. The last column in Tables 8 and 11 shows the number of reachable query node-pairs 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.

On the sparse datasets, GRAIL (using $d = 2$ traversals) has the smallest construction time among all indexing methods, though



(a) (b) (c)

Figure 3: Effect of Increasing Number of Intervals: (a) `ecoo`, (b) `cit-patents`, (c) `rand10m10x`

Dataset	GRAIL	DFS	HLSS	INT	Dual	PT	3HOP	#Pos.Q
agrocyc	57	44	71	158	65	8	235	133
amaze	764	1761	99	101	63	7	4621	17259
anthra	49	40	68	157	65	8.5	139	97
ecoo	56	52	69	160	65	8.0	241	129
human	80	36	81	238	77	14	–(t)	12
kegg	1063	2181	104	100	72	7.1	81	20133
mtbrv	49	55	81	144	75	7.2	218	175
nasa	26.5	138	96	121	80	7.8	79	562
vchocyc	49.6	56	76	145	79	7.2	206	169
xmark	79	390	86	119	92	8.2	570	1482

Table 8: Small Sparse Graphs: Query Time (ms)

Datasets	GRAIL	HLSS	INT	Dual	PT	3HOP
agrocyc	50736	40097	27100	58552	39027	87305
amaze	14840	17110	10356	433345	12701	1425K
anthra	49996	33532	26310	37378	38250	58796
ecoo	50480	34285	26986	58290	38863	97788
human	155244	109962	79272	54678	117396	–(t)
kegg	14468	17427	10242	504K	12554	10146
mtbrv	38408	30491	20576	41689	29627	74378
nasa	22420	20976	18324	5307	21894	28110
vchocyc	37964	30182	20366	26330	29310	75957
xmark	24320	23814	16474	16434	20596	14892

Table 9: Small Sparse Graphs: Index Size (Num. Entries)

Dataset	GRAIL	HLSS	INT	Dual	PT	3HOP
arxiv	21.7	– (t)	20317	450761	9639	– (t)
citeseer	43.1	120993	7682	26118	751.5	113075
go	9.5	69063	1144	4116	220.9	30070
pubmed	43.9	146807	7236	27968	774.0	168223
yago	18.2	28487	2627	4928	512	39066

Table 10: Small Dense Graphs: Construction Time (ms)

Dataset	GRAIL	DFS	HLSS	INT	Dual	PT	3HOP	#Pos.Q
arxiv	575	12179	–(t)	273	281	24.4	–(t)	15459
citeseer	82.6	408	328	227	141	24.5	263	388
go	51.4	127	273	151	136	11.6	104	241
pubmed	75.5	375	315	254	132	22.1	264	690
yago	46.9	121	258	181	88.4	13.8	157	171

Table 11: Small Dense Graphs: Query Time (ms)

Dataset	GRAIL	HLSS	INT	Dual	PT	3HOP
arxiv	24000	–(t)	145668	3695K	86855	–(t)
citeseer	64320	114088	142632	426128	91820	74940
go	27172	60287	40644	60342	37729	43339
pubmed	72000	102946	181260	603437	107915	93289
yago	26568	57003	57390	79047	39181	36274

Table 12: Small Dense Graphs: Index Size (Num. Entries)

PathTree is very effective as well. 3HOP could not run on `human`. In terms of query time, PathTree is the best; it is 3-100 times faster than GRAIL, and typically 10 times faster than HLSS and Dual. INT is not very effective, and neither is 3HOP. However, it is worth noting that DFS gives reasonable query performance, often faster than indexing methods, other than PathTree and GRAIL. Given the fact that DFS has no construction time or indexing size overhead, it is quite attractive for these small datasets. The other methods have comparable index sizes, though INT has the smallest sizes.

On the small dense datasets, 3HOP and HLSS could not run on `arxiv`. GRAIL (with $d = 2$) has the smallest construction times and index size of all indexing methods. It is also 2-20 times faster than a pure DFS search in terms of query times, but is 3-20 times slower than PathTree. Even on the dense datasets the pure DFS is quite acceptable, though indexing does deliver significant benefits in terms of query time performance.

As observed for `amaze` and `kegg` (Table 8), and for `arxiv` (Table 11), the query time for GRAIL increases as the number of reachable node-pairs (#PosQ) increase in the query set. However, the graph topology also has an influence on the query performance. Small-sparse datasets, such as `kegg` and `amaze`, have a central node with a very high in-degree and out-degree. For such graphs, for many of the queries, GRAIL has to scan all the children of this central node to arrive at the target node. This significantly increases the query time (line 7 in Algorithm 2). To alleviate this problem, one possibility is that nodes with large out-degree could keep the intervals of their children in a spatial index (e.g. R-Trees) to accelerate target node/branch lookup.

Dataset	Construction (ms)	Query Time (ms)		Index Size
	GRAIL	GRAIL	DFS	GRAIL
cit-patents	61911.9	1579.9	43535.9	37747680
citeseer	1756.2	94.9	56.6	2775788
citeseerx	19836	12496.6	198422.8	26272704
go-uniprot	32678.7	194.1	391.6	27871824
uniprot22m	5192.7	132.3	44.7	6381776
uniprot100m	58858.2	186.1	77.2	64349180
uniprot150m	96618	183	87.7	100150400

Table 13: Large Real Graphs

		Constr. (ms)	Query Time (ms)		Index Size
Size	Deg.	GRAIL	GRAIL	DFS	GRAIL
rand10m	2	128796	187.2	577.6	100M
	5	226671	5823.9	90505	100M
	10	407158	1415296.1	–(t)	100M
rand100m	2	1169601	258.2	762.7	800M
	5	1084848	20467	131306	400M

Table 14: Scalability: Large Synthetic Graphs

4.3 Large Datasets: Real and Synthetic

Table 13 shows the construction time, query time, and index size for GRAIL ($d = 5$) and pure DFS, on the large real datasets. We also ran PathTree, but unfortunately, on `cit-patents` and `citeseerx` is aborted with a memory limit error (–(m)), whereas for the other datasets it exceeded the 20M ms time limit (–(t)). It

was able to run only on citeseer data (130406 ms for construction, 47.4 ms for querying, and the index size was 2360732 entries). On these large datasets, none of the other indexing methods could run. GRAIL on the other hand can easily scale to large datasets, the only limitation being that it does not yet process disk-resident graphs. We can see that GRAIL outperforms pure DFS by 2-27 times on the denser graphs: go-uniprot, cit-patents, and citeseerx. On the other datasets, that are very sparse, pure DFS can in fact be up to 3 times faster.

We also tested the scalability for GRAIL (with $d = 5$) on the large synthetic graphs. Table 14 shows the construction time, query time and index sizes for GRAIL and DFS. Once again, none of the other indexing methods could handle these large graphs. PathTree too aborted on all datasets, except for rand10m2x with avg. degree 2 (it took 537019 ms for construction, 211.7 ms for querying, and its index size was 69378979 entries). We can see that for these datasets GRAIL is uniformly better than DFS in all cases. Its query time is 3-15 times faster than DFS. In fact on rand10m10x dataset with average density 10, DFS could not finish in the allocated 20M ms time limit. Once again, we conclude that GRAIL is the most scalable reachability index for large graphs, especially with increasing density.

Dataset	Constr. Time (ms)		Query Time (ms)		Index Size (# Entries)	
	with E	w/o E	with E	w/o E	Label	Exceptions
amaze	1930.2	3.8	454.2	758.1	22260	19701
human	24235.5	134.5	596.4	81.1	310488	11486
kegg	2320.3	6.4	404.3	1055.1	28936	18385
arxiv	64913.2	53.7	532.4	424.99	60000	3754315

Table 15: GRAIL: Effect of Exceptions

4.4 GRAIL: Sensitivity

Exception Lists: Table 15 shows the effect of using exception lists in GRAIL: "with E" denotes the use of exception lists, where as "w/o E" denotes the default DFS search with pruning. We used $d = 3$ for amaze, $d = 4$ for human and kegg (which are the sparse datasets), and $d = 5$ for arxiv. We can see that using exceptions does help in some cases, but the added overhead of increased construction time, and the large size overhead of storing exception lists (last column), do not justify the small gains. Furthermore, exceptions could not be constructed on the large real graphs.

Number of Traversals/Intervals (d): In Figure 3 we plot the effect of increasing the dimensionality of the index, i.e., increasing the number of traversals d, on one sparse (ecoo), one large real (cit-patents), and one large synthetic (rand10m10x) graph. Construction time is shown on the left y-axis, and query time on the right y-axis. It is clear that increasing the number of intervals increases construction time, but yields decreasing query times. However, as shown for ecoo, increasing d does not continue to decrease query times, since at some point the overhead of checking a larger number of intervals negates the potential reduction in exceptions. That is why the query time increases from $d = 4$ to $d = 5$ for ecoo. To estimate 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 benefits 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). Other measures based on the reduction in the number of (direct) exceptions per new traversal could also be designed.

Effect of Reachability: For all of the experiments above, we issue 100K random query pairs. However, since the graphs are very sparse, the vast majority of these pairs are not reachable. As an alternative, we generated 100K 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). Tables 16 and 17 show the query time performance of GRAIL and pure DFS for the 100K random and 100K only positive queries, on some small and large graphs. We used $d = 2$ for human, $d = 4$ for arxiv and $d = 5$ for the large graphs. The frequency distribution of number of hops between source and target nodes for the queries is plotted in Figure 4. Generally speaking, querying only reachable pairs takes longer (from 2-30 times) for both GRAIL and DFS. Note also that GRAIL is 2-4 times faster than DFS on positive queries, and up to 30 times faster on the random ones.

Dataset	GRAIL					
	Random			Positive		
	Avg	σ	Avg/σ	Avg	σ	Avg/σ
human	80.4	23.5	0.292	1058.7	146.4	0.138
arxiv	420.6	11.2	0.027	334.2	4.19	0.013
cit-patents	1580.0	121.4	0.077	3266.4	178.2	0.055
citeseerx	10275.5	3257.2	0.317	310393.2	14809	0.048
rand10m5x	5824.0	363.6	0.062	19286.8	1009.8	0.052

Table 16: Average Query Times and Standard Deviation

Dataset	DFS					
	Random			Positive		
	Avg	σ	σ/Avg	Avg	σ	σ/Avg
human	36.4	10.4	0.286	919.6	89.5	0.097
arxiv	12179.6	179.1	0.014	1374.2	30.4	0.022
cit-patents	43535.9	1081.3	0.008	6827.8	372.4	0.055
citeseerx	198422.9	10064.3	0.051	650232.0	7411.4	0.011
rand10m5x	90505.9	3303.2	0.036	49989.3	1726.9	0.035

Table 17: Average Query Times and Standard Deviation

Effect of Query Distribution: Tables 16 and 17 show the average query times and the standard deviation for GRAIL and DFS, respectively. Ten sets, each of 10K queries, are used to obtain the mean and standard deviation of the query time. In random query sets when using GRAIL the coefficient of variation (CV – the ratio of the standard deviation to the mean) is between $1/3$ and $1/40$, whereas it varies from $1/3$ to $1/70$ when using DFS. As expected, DFS has more uniform query times compared to GRAIL because GRAIL can cut some queries short via pruning while in other queries GRAIL imitates DFS. However for query sets with all reachable (positive) node-pairs, CV decreases for GRAIL since the likelihood of pruning and early termination of the query decreases. On the other hand, there is no such correlation for DFS.

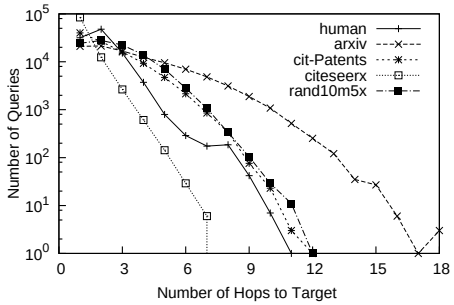


Figure 4: Reachability: Distribution of Number of Hops

Effect of Density: We studied the effect of increasing edge density of the graphs by generating random DAGs with 10 million nodes, and varying the average density from 2 to 10, as shown in Figure 5. As we can see, both the construction and query time increase with

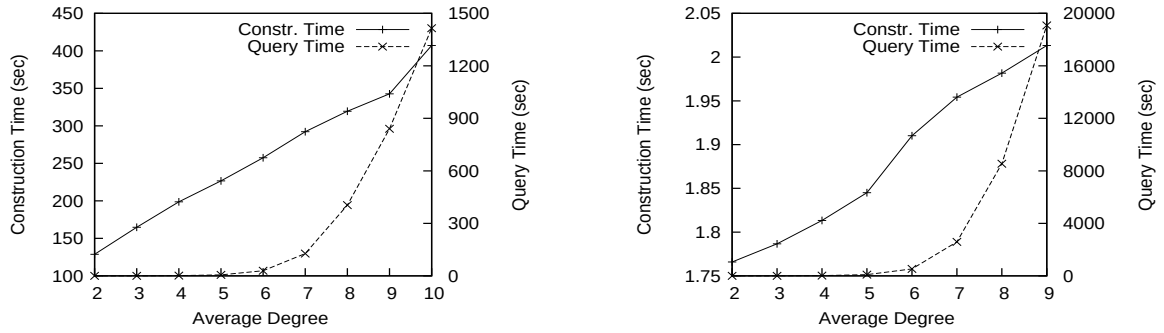


Figure 5: Increasing Graph Density: (a) GRAIL, (b) DFS

increasing density. However, note that typically GRAIL (with $d = 5$) is an order of magnitude faster than pure DFS in query time. Also GRAIL can handle dense graphs, where other methods fail; in fact, one can increase the dimensionality to handle denser graphs.

5. CONCLUSION

We proposed GRAIL, a very 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 simple DFS 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 DFS search, on large real graphs; in fact, for these large graphs existing indexing methods are simply not able to scale.

In GRAIL, we have mainly exploited a randomized traversal strategy to obtain the interval labelings. We plan to explore other labeling strategies in the future. In general, the problem of finding the next traversal that eliminates the maximum number of exceptions is open. The question whether there exists an interval labeling with d dimensions that has no exceptions, is likely to be NP-complete. Thus it is also of interest to obtain a bound on the number of dimensions required to fully index a graph without exceptions. In the future, we also plan to generalize GRAIL to dynamic graphs.

6. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *SIGMOD Rec.*, 18(2):253–262, 1989.

[2] P. Bouros, S. Skiadopoulos, T. Dalamagas, D. Sacharidis, and T. Sellis. Evaluating reachability queries over path collections. In *SSDBM*, page 416, 2009.

[3] R. Bramandia, B. Choi, and W. K. Ng. On incremental maintenance of 2-hop labeling of graphs. In *WWW*, 2008.

[4] Y. Chen. General spanning trees and reachability query evaluation. In *Canadian Conference on Computer Science and Software Engineering*, Montreal, Quebec, Canada, 2009.

[5] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, 2008.

[6] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EBDT*, 2008.

[7] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal of Computing*, 32(5):1335–1355, 2003.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[9] C. Demetrescu and G. Italiano. Fully Dynamic Transitive Closure: Breaking through the $O(n^2)$ Barrier. In *FOCS*, 2000.

[10] C. Demetrescu and G. Italiano. Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures. *Journal of Discrete Algorithms*, 4(3):353–383, 2006.

[11] P. F. Dietz. Maintaining order in a linked list. In *STOC*, 1982.

[12] H. He, H. Wang, J. Yang, and P. S. Yu. Compact reachability labeling for graph-structured data. In *CIKM*, 2005.

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

[14] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, 2009.

[15] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficient answering reachability queries on very large directed graphs. In *SIGMOD*, 2008.

[16] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. *J. Comput. Syst. Sci.*, 65(1):150–167, 2002.

[17] I. Krommidas and C. Zaroliagis. An experimental study of algorithms for fully dynamic transitive closure. *Journal of Experimental Algorithmics*, 12:16, 2008.

[18] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *STOC*, 2004.

[19] R. Schenkel, A. Theobald, and G. Weikum. HOPI: an efficient connection index for complex XML document collections. In *EBDT*, 2004.

[20] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE*, 2005.

[21] S. Trissl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, 2007.

[22] H. Wang, H. He, J. Yang, P. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, 2006.

[23] J. X. Yu, X. Lin, H. Wang, P. S. Yu, and J. Cheng. Fast computation of reachability labeling for large graphs. In *EBDT*, 2006.

APPENDIX

A. EXCEPTION LISTS

If one desires to maintain exception lists for each node, a basic property one can exploit is that if $v \in E_u$ then for each parent p of v, it must be the case that u cannot reach p. This is easy to see, since if for any parent p, if $u \rightarrow p$, then by definition $u \rightarrow v$, and then v cannot be an exception for u. Thus, the exception list for a node u can be constructed recursively from the exception lists of its parents. Nevertheless, the complexity of this step is the same as that of computing the transitive closure, namely $O(nm)$, which is impractical.

In GRAIL, we categorize exceptions into two classes. If L_u contains v [1], but none of the children of u contains L_v, then call the exception between u and v a *direct exception*. On the other hand, if at least one child of u contains v as an exception, then we call the exception between u and v as an *indirect exception*. For example, in Figure 2(b) 3 is a direct exception for 4, but 1 is an indirect exception for 2, since there are children of 2 (e.g., 5) for whom 1 is still an exception. Table 2 shows the list of direct (denoted E^d) and indirect (denoted E^i) exceptions for the DAG.

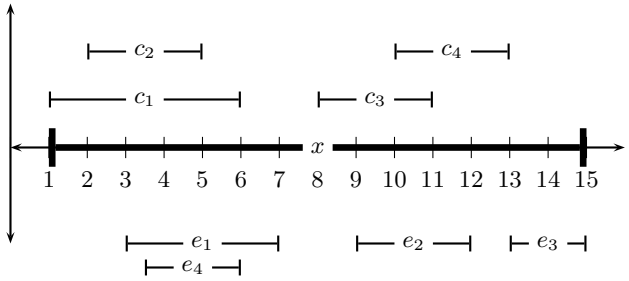


Figure 6: Direct Exceptions: c_i denote children and e_i denote exceptions, for node x.

Direct Exceptions: Let us assume that $d = 1$, that is, each node has only one interval. Given the interval labeling, GRAIL constructs the exception lists for all nodes in the graph, as follows. First all n node intervals are indexed in an *interval tree* [8], which takes $O(n \log n)$ time and $O(n)$ space. Querying the interval tree for intervals intersecting a given range of interest can be done in $O(\log n)$ time. To find the direct exceptions of node x, we first find the maximal ranges among all of its children. Next the *gap intervals* between the maximal ranges are queried to find exceptions. Consider the example in Figure 6, where we want to determine the exceptions for node x. c_i denote the children's intervals, whereas e_i denote the exceptions to be found. We can see that $L_x = [1, 15]$, and the maximal intervals among all its children are $L_{c_1} = [1, 6]$, $L_{c_3} = [8, 11]$, and $L_{c_4} = [10, 14]$. It is clear that if an exception is contained completely within any one of the maximal intervals, it cannot be a direct exception. Thus to find the direct exceptions for x, i.e., to find E^d_x, we have to query the gaps between the maximal ranges to find the intersecting intervals. In our example, the gaps are given by the following intervals: $[6, 8]$, $[11, 11 + \delta]$, and $[13, 13 + \delta]$, where $\delta > 0$ is chosen so that $L^2_{c_i} + \delta < L^2_{c_j}$ for any pair of maximal ranges. In our example, a value of $\delta = 1$ suffices, thus we query the interval tree to find all intervals that intersect $[6, 8]$, or $[11, 12]$, or $[13, 14]$, which will yield $E^d_x = \{e_1, e_2, e_3, e_4\}$.

[1] 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.

Indirect Exceptions: Given that we have the list of direct exceptions E^d_x for each node, the construction of the indirect exceptions (E^i_x) proceeds in a bottom up manner from the leaves to the roots. Let $E_{c_j} = E^d_{c_j} \cup E^i_{c_j}$ denote the list of direct or indirect exceptions for a child node c_j. To compute E^i_x, for each exception $e \in E_{c_j}$ we check if there exists another child c_k such that $L_e \subseteq L_{c_k}$ and $e \notin E_{c_k}$. If the two conditions are met e cannot be an exception for x, since $L_e \subseteq L_{c_k}$ implies that e is potentially a descendant of c_k, and $e \notin E_{c_k}$ confirms that it is not an exception. On the other hand, if the test fails, then e must be an indirect exception for x, and we add it to E^i_x. For example, consider node 2 in Figure 2(b). Assume we have already computed the exception lists for each of its children, $E_3 = E^d_3 \cup E^i_3 = \emptyset$, and $E_5 = E^d_5 \cup E^i_5 = \{1, 3, 4, 7, 8\}$. We find that for each $e \; in E_5$, 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 $E^i_2 = \{1, 4\}$, as illustrated in Table 2.

Multiple Intervals: To find the exceptions when $d > 1$, GRAIL first computes the direct and indirect exceptions from the first traversal, as described above. For computing the remaining exceptions after the i-th traversal, GRAIL processes the nodes in a bottom up order. For every direct exception in $e \in E^d_x$, remove e from the direct exception list if e is not an exception for x for the i-th dimension, and further, decrement the counter for e in the indirect exceptions list E^i_p for each parent p of x. Also, if after decrementing, the counter for any indirect exception e becomes zero, then move e to the direct exception list E^d_p of the parent p, provided $L_e \subseteq L_p$. In this way all exceptions can be found out for the i-th dimension or traversal.