GRIPP - Indexing and Querying Graphs based on Pre- and Postorder Numbering

Silke Trißl, Ulf Leser
Institute for Computer Science, Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
{trissl, leser}@informatik.hu-berlin.de

Abstract

Many applications require querying graph-structured data. As graphs grow in size, indexing becomes essential to ensure sufficient query performance. We present the GRIPP index structure (GRaph Indexing based on Pre- and Postorder numbering) for answering reachability and distance queries in graphs. GRIPP requires only linear space and can be computed very efficiently. Using GRIPP, we can answer reachability queries on graphs with 5,000,000 nodes on average in less than 5 milliseconds, which is unrivaled by previous methods. We can also answer distance queries on large graphs more efficiently using the GRIPP index strucutre. We evaluate the performance and scalability of our approach on real, random, and scale-free networks using an implementation of GRIPP inside a relational database management system. Thus, GRIPP can be integrated very easily into existing graph-oriented applications.

Contents

1 Introduction

Managing, analyzing, and querying graph-like data is important in many areas such as geographic information systems [13], web site analysis [10], and XML documents with XPointers [23]. In addition, the semantic web builds on RDF, a graph-based data model, and graph-based query languages such as RQL [18] or SparQL[1]. Thus, querying graphs is likely to become even more important in the near future.

In our area of research we mainly deal with data from the life sciences. In every living cell there exist complex mechanisms involving DNA, proteins, and chemical compounds that are constitutive for the functioning of the cell. It is now commonly acknowledged that further progress in understanding the complex mechanisms inside a living cell can only be achieved if the interplay of many components, organized in networks, is understood [4].

The size of the networks under consideration can be very large. Typical biological networks, such as gene regulation or protein interaction networks, are currently in the range of tens of thousands of nodes. This number increases dramatically as activity in measuring interactions moves from bacteria to higher organisms, such as humans [3]. Already today, networks of biomedical entities (genes, diseases, drugs, etc.) extracted from large publication databases contain up to 6 million edges[2]. Every network can also be considered as graph.

Querying large graphs is a challenge. Important types of queries in labeled, directed graphs are *reachability*, *distance*, and *path queries*. We assume that the graph is stored in a relational database management system. Thus, all queries need to be translated into SQL queries. Using a naive approach, the queries can be answered by traversing the graph at query time, starting from node v and performing a depth-first or breadth-first search [8]. This method does not need any precomputed index, but must traverse the entire graph.

As a second option, we can pre-compute the transitive closure (TC) of the graph. Stored in a table, we can use the TC as an index with which reachability queries can be answered by a single table lookup. But on the downside, the size of the TC is in worst-case quadratic in the number of nodes of the graph [2]. This renders its computation and storage infeasible for large graphs.

There is a need for new index structures to efficiently answer reachability and distance queries on large graphs. In this paper we present such an index structure, called GRIPP. Its main idea is an adaptation of the pre- and postorder numbering scheme – so far only applied to trees [9] and DAGs [24] – to (cyclic, possibly unrooted) graphs. The GRIPP index can be computed very efficiently and requires only linear space in the size of the graph. Querying GRIPP requires multiple queries, but typically orders of magnitude less queries than graph traversal. Thus, in general the query performance of GRIPP compares favorably with graph traversal and can be used on graphs far beyond the scope of TC. The properties of TC, recursive query strategies, and GRIPP are compared qualitatively in Table 1.

Clearly, indexing trees is simpler than indexing general graphs. In general, the performance of different approaches to indexing and querying graphs largely depends on the structure of the graphs under study, for instance, whether they are random or scale-free,

[1]http://www.w3.org/TR/rdf-sparql-query

[2]See http://www.pubgene.org.

	Query time	Index creation
Transitive closure	very fast	infeasible for graphs >10,000 nodes
Recursive strategy	very slow	no index needed
GRIPP	fast	fast

Table 1: Different strategies for answering reachability queries on graphs, separating efforts for indexing and querying.

and whether they are dense or sparse. These differences are often not sufficiently recognized when new methods are developed. We are especially interested in an index structure that exploits the structure of biological networks, which are, like many other real-world networks, scale-free. This means that the distribution of the node degree follows a power-law [15], resulting in very few well connected nodes and many nodes having only one incoming or outgoing edges (see Figure 1).

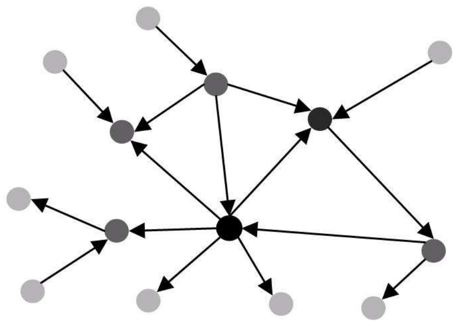


Figure 1: A scale-free graph. The darker the nodes, the higher their degree.

Our paper is organized as follows. In the next section we describe our graph data model and common ways to query graph structured data. In Section 3 we present the GRIPP index structure itself. In Section 4 we show how to efficiently evaluate reachability and distance queries using GRIPP. In Section 5, we describe heuristics for indexing scale-free graphs. Section 6 gives implementation details for the presented methods. In Section 7, we give experimental results for synthetic random, synthetic scale-free, and real biological networks, with graph sizes ranging from 1,000 to five million nodes and different graph densities. Section 8 discusses related work and Section 9 concludes the paper.

2 Background

We adopt notation from Cormen et al. [8]. A graph $G = (V, E)$ is a collection of nodes V and edges E. We only consider connected graphs with labeled nodes and directed, unlabeled edges. The *size* of a graph, $|G|$, is the number of nodes $|V|$ plus the number of edges $|E|$. The *degree* of a node is the number of incoming and outgoing edges of a node. Given a graph G, a *path* p is a sequence of nodes that are connected by directed edges.

We want to answer reachability and distance queries on graphs.

Definition 2.1 (Reachability) *Let $G = (V, E)$ and $v, w \in V$. w is* reachable *from v iff at least one path from v to w exists.*

Definition 2.2 (Distance) *Let $v, w \in V$. The length of the shortest path is called the* distance *between v and w. If no path between v and w exists, the path length is infinite.*

Of course, for a given pair of nodes v, w there can exist several paths that are shortest.

2.1 Querying Graphs in Databases

We analyze the problem of answering reachability and distance queries on graphs stored in a relational database system.

We assume that graphs are stored as a collection of nodes and edges. The information on nodes includes a unique identifier and possibly additional information. Edges are stored as binary relationship between two nodes, i.e., as adjacency list. Additional attributes on edges can be stored as well.

Reachability is concerned with the question if a path between two nodes exists. Given two nodes v and w, the function $reach(v, w)$ returns `true` if a path from v to w exists, otherwise `false`. For distance queries we are interested in the length of the shortest path. The function $dist(v, w)$ returns the distance, i.e., the length of the shortest path, between nodes v and w. If no path exists, the function returns `null`.

The simplest way to answer questions on graphs is to traverse the graph at query time using depth- or breadth-first search [8]. This requires time proportional to the number of traversed edges, i.e., in worst-case the size of the graph. In a relational database system depth- and breadth-first search can not be expressed by standard SQL in all database systems, but must be implemented using user-defined functions.

The commercial database management systems Oracle and IBM DB2 have implemented recursive query strategies. IBM DB2 supports the SQL 2003 standard, while Oracle uses its own syntax. The implementations aim at hierarchy data, i.e., mainly tree structured data. Starting with version 10g Oracle also provides methods to handle cycles. Oracle's implementation traverses graph structured data in depth-first order. When answering reachability or distance questions, Oracle enumerates all cycle free paths between the start and end node. This behavior makes the current implementation inefficient for reachability and distance queries as is discussed in Section 7. We did not evaluate the implementation of the SQL 2003 standard in IBM DB2.

Another option to answer some queries in graphs is to pre-compute the transitive closure TC. The TC of a graph is the set of node pairs (u, v) for which a path from u to v exists. Efficient algorithms for computing the TC in relational databases have been developed [2, 21]. But the size of the TC is $O(|V|^2)$, which makes it inapplicable to large graphs. In addition to that, the transitive closure is only capable of answering reachability questions. Distance questions can be answered if, in addition to node pairs, the distance between the nodes is stored as well. Answering questions about path lengths or actual paths is not possible using the transitive closure alone.

A different indexing strategy is to label nodes using the pre- and postorder labeling scheme. But this indexing scheme was only described for tree structured data [9]. As it allows to maintain the order of child nodes in the tree it is well suited to index XML documents [11]. In previous work we extended this indexing scheme to index large ontologies that are structured as directed acyclic graphs (DAGs). We used an ’unfolding’ technique, where each added ’non-tree’ edge introduces new entries to the index structure [24]. The target node of the additional edge as well as all successor nodes get additional pre- and postorder ranks. Thus, each node has as many pre- and postorder values as there are paths from the root node to this node. Using this technique the index size grows tremendously with increasing number of edges, making it only feasible for tree-like DAGs. For highly connected DAGs as well as for graphs we have to apply different indexing methods.

2.2 Pre- and postorder labeling

Our indexing scheme for graphs is based on the pre- and postorder indexing scheme for trees. We will therefore first explain this indexing scheme for trees in more detail.

Given a tree, in the pre- and postorder indexing scheme each node in the tree receives three values, a preorder value, a postorder value, and the depth of the node in the tree. Pre- and postorder values are assigned to a node according to the order in which the nodes are visited during a depth-first traversal of the tree. The preorder value v_{pre} is assigned the first time node v is encountered during the traversal. The postorder value v_{post} is assigned after all successor nodes of v have been traversed. Originally, two counters are used, one for the preorder value and one for the postorder value. Both are incremented after each assignment. In our implementation we use only one counter for both values as this is advantageous for querying. We will explain this in the following.

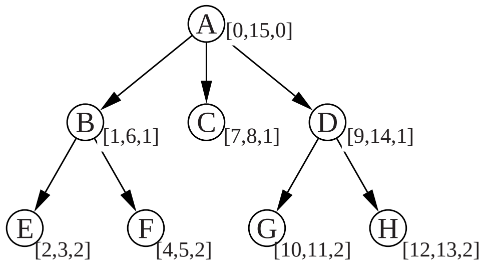
The depth of v, v_{depth}, is also assigned during the depth-first traversal. The depth of the root node of the tree is 0. The depth of any node v in the tree is the distance to the root node.

Example 2.1 *A pre- and postorder labeled tree with depth information can be seen in Figure 2(a).*

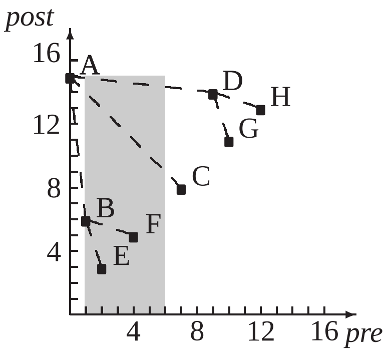
The the list of nodes together with assigned pre- and postorder values and depth information form an index through which reachability and distance queries on trees can be answered with a single SQL query. If w is reachable from v, w must have a higher preorder and lower postorder value than v, i.e., $w_{pre} > v_{pre} \wedge w_{post} < v_{post}$. However, the evaluation of this condition in a RDBMS is prohibitively slow due to the two non-equijoins [12]. Fortunately, the test condition can be restricted to a single value using the following observation. During the creation of the index a node v always receives its preorder value before its successors get their pre- and postorder values. The postorder value of node v is assigned after all successor nodes have pre- and postorder values. As the counter is incremented after every assignment, the pre- as well as postorder values of any successor node w of v must lie within the borders given by the pre- and postorder values of v, i.e., $[v_{pre}, v_{post}]$. Thus, $reach(v, w) \Leftrightarrow v_{pre} < w_{pre} < v_{post}$.

If $reach(v, w)$ evaluates to true we know there exists a path from v to w. As in trees only one path between two nodes may exist this is also the shortest path. The length of that path is $w_{depth} - v_{depth}$, i.e., $dist(v, w) = w_{depth} - v_{depth}$.

Example 2.2 *In Figure 2(b), the gray area shows the preorder range in which all reachable nodes from node B are located.*



(a) Pre- and postorder labeling of a tree.



(b) Pre-/ postorder plane. In gray: Preorder range between B_{pre} and B_{post}.

Figure 2: Indexing trees by pre- and postorder labeling.

The method as described only works for trees. As soon as nodes have multiple incoming edges they are visited multiple times during a traversal and thus no unique pair of pre- and postorder values can be assigned.

3 GRIPP – A Graph Index Structure

The main idea of the GRIPP index structure is intriguingly simple. In GRIPP every node in the graph receives at least one pair of pre- and postorder values and depth information. However, as nodes can have multiple parents, one pair is not sufficient to encode the entire graph structure. Some nodes in the graph have to be encoded by more than one pair of pre- and postorder values and depth information.

For now, we assume that the graph has exactly one root node, i.e., one node without incoming edges. We also assume that an arbitrary, yet fixed, order among nodes exists, e.g., an order based on node labels. In Section 5 we explain a suitable order for graphs and we also show how to deal with graphs with multiple or without root nodes.

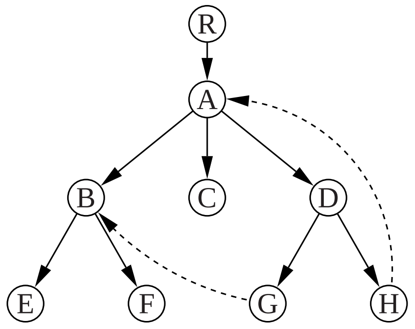
For the creation of the GRIPP index we start at the root node of G. During a depth-first traversal of G we assign pre- and postorder values and depth information. We always traverse child nodes of a node according to their order. A node v with $n \geq 1$ incoming edges is reached n times during the traversal on edges e_i, $1 \leq i \leq n$. The edge e_i on which we reach v for the first time is called a *tree edge*. We assign a preorder value and depth to v and proceed the depth-first traversal. After all successor nodes of v have a value pair, v receives its postorder value. Later, we will reach node v over edges e_j, $e_j \neq e_i$. We call those edges e_j *non-tree edges*. Each time we reach v we assign a new triple (preorder value, postorder value, and depth) to node v. But we do not traverse child nodes of v.

We store the pre- and postorder values and the depth together with the node identifier as *instances* in the *index table*, $IND(G)$. Every node will have as many instances in $IND(G)$ as it has incoming edges in G. Analogously to the distinction of tree and non-tree edges we distinguish between tree and non-tree instances in $IND(G)$.

Definition 3.1 (Tree and non-tree instances) *Let $IND(G)$ be the index table of graph G. Let $v \in V$ be a node of G and v' be an instance of v in $IND(G)$. v' is a* tree instance *of v, iff it was the first instance created for v in $IND(G)$. Otherwise v' is a* non-tree instance *of v.*

In the following, we refer to any instance in $IND(G)$ of a node v as v', to a tree instance as v^T, and to a non-tree instance as v^N. The set of tree instances in $IND(G)$ is I^T and the set of non-tree instances is I^N. In analogy, the set of tree edges is E^T and the set of non-tree edges E^N. We shall need the distinction of instances for querying as explained in Section 4.

Example 3.1 *Figure 3(a) shows a graph and Figure 3(b) shows its index table, resulting from a traversal in lexicographic order of node labels. Nodes A and B have two instances in $IND(G)$ because they have two incoming edges.*



(a) A graph G.

node	pre	post	depth	type
R	0	21	0	tree
A	1	20	1	tree
B	2	7	2	tree
E	3	4	3	tree
F	5	6	3	tree
C	8	9	2	tree
D	10	19	2	tree
G	11	14	3	tree
B	12	13	4	non-tree
H	15	18	3	tree
A	16	17	4	non-tree

(b) Index table $IND(G)$.

Figure 3: Graph G and its GRIPP index table $IND(G)$. Solid lines in the graph represent tree edges, dashed lines are non-tree edges.

The GRIPP index structure resembles a rooted tree, which we call the *order tree*, $O(G)$.

Definition 3.2 (Order tree) *Let $G = (V, E)$ and let $IND(G)$ be its index table. The* order tree, *$O(G)$, is a tree that contains all instances of $IND(G)$ as nodes connected by all edges of G.*

Intuitively, $O(G)$ consists of a spanning tree $T(G)$ and a 'non-tree' part $N(G)$. The spanning tree contains the tree instance of every node in the graph and is connected by only tree edges. The non-tree part of $O(G)$ contains one node for every non-tree instance in $IND(G)$ connected by a non-tree edge to a node in the spanning tree $T(G)$. Therefore, every non-tree instance is a leaf node in $O(G)$, while tree instances can be inner or leaf nodes. Note that the structure of $O(G)$ depends on the order in which G is traversed. In Section 5 we shall explain how we can select an order that is specifically well suited.

Definition 3.3 (Partitioning) *Let $G = (V, E)$ be a graph with the index table $IND(G)$ and resulting order tree $O(G)$. $O(G)$ can be* partitioned *into two disjoint graphs: a spanning tree $T(G) = (I^T, E^T)$ and a disconnected non-tree part $N(G) = (I^N, E^N)$, with $|I^T| = |V|$, $I^T \cup I^N = IND(G)$, $E^T \cup E^N = E$.*

Example 3.2 *In Figure 4 the instances of $IND(G)$ shown in Figure 3(b) are plotted. Nodes A and B have two nodes in $O(G)$ as they have two instances in $IND(G)$, one tree and one non-tree instance.*

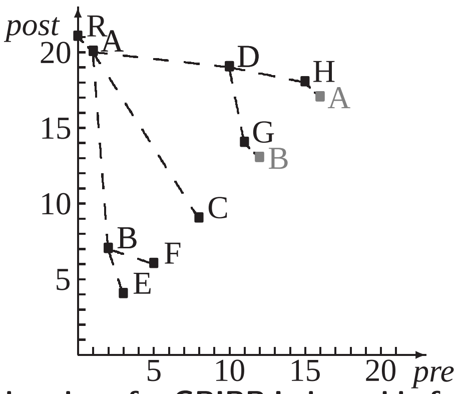


Figure 4: Pre-/ postorder plane for GRIPP index table from Figure 3(b). Dotted lines indicate $O(G)$. Non-tree instances are displayed in gray.

3.1 Properties of the GRIPP index

3.1.1 Time and Space Requirements

The space requirements to store the GRIPP index table is linear in the size of the graph. The GRIPP index table has as many entries as G has edges plus one entry for the root node, because (a) every edge traversal generates one instance in $IND(G)$ and (b) every edge is traversed exactly once.

To create the GRIPP index structure we perform a depth-first search over a graph G. The depth-first search has a time complexity of $O(|G|)$ (see [8]). We will analyze the time complexity to create GRIPP in more detail now. During the index creation we basically perform four steps for every edge.

These steps are

- return the next child node v of a node,
- check if v has already been seen during the traversal,
- if not add v to the list of traversed nodes, and
- insert v as instance in $IND(G)$.

We assume we can search a specific tuple in a table containing n tuples in $log(n)$ time and that insertion is constant. To get the next child node for a node we require $log(|E|)$ time. To get all child nodes for one node we reqire $n * log(E)$, with n being the outdegree of that node. To get the child nodes for all nodes we therefore need $|E| * log(|E|)$ time as we have in total $|E|$ edges. To check if we have already traversed that child node takes $log(|V|)$ time, i.e., for all child nodes $|E| * log(|V|)$ time. During the traversal we will add all nodes once to the list of traversed nodes (stored as relational table), which takes in total $|V|$ time. In addition we add an instance for every child node to $IND(G)$, which takes $|E|$ time. Therefore, the total required time is $|E| * log(|E| * |V|) + |V| + |E|$ to create the GRIPP index structure in a relational database system.

3.1.2 Properties of $O(G)$

Preorder of tree instance In the GRIPP index structure the tree instance of a node v has a lower preorder rank than all non-tree instances of that node. Intuitively, we traverse G in depth-first order. When we reach v for the first time, the traversed edge becomes a tree edge and v is added with a tree instance to the GRIPP index table. The next time we reach v it is added with a non-tree instance to the index table. As the counter for the pre- and postorder values is never decreased the preorder value of the non-tree instance must be higher than that of the tree instance.

Distance of nodes in $O(G)$ Let $v, w \in V$ and $v', w' \in O(G)$ be an instance of v and w, respectively. If v' is ancestor to w' in $O(G)$ we can determine the distance of v' and w' in $O(G)$ by calculating $w'_{depth} - v'_{depth}$. Note that this is not the distance between v and w in G. To aquire the distance between two nodes we have to do more work (see Section 4.3).

Example 3.3 *Figure 5 shows an order tree $O(G)$ for a scale-free graph G with 100 nodes and 200 edges. The child nodes in the order tree are ordered according to their preorder values from left to right. During the index creation we traverse the graph in depth-first order. We stop extending a path if (a) the node has no child nodes or (b) the node has already been traversed. This means we traverse the graph as 'deep' as possible. This is reflected in Figure 5. The tree instance c^T of the first traversed child node c during the index creation is the left-most child node of the root in Figure 5. As c has many reachable nodes in G, c^T has many successor nodes in $O(G)$. The remaining child nodes of the root node have only few successor nodes. These nodes (a) either had no instance in $IND(G)$ when they were traversed or (b) are non-tree instances of already traversed nodes.*

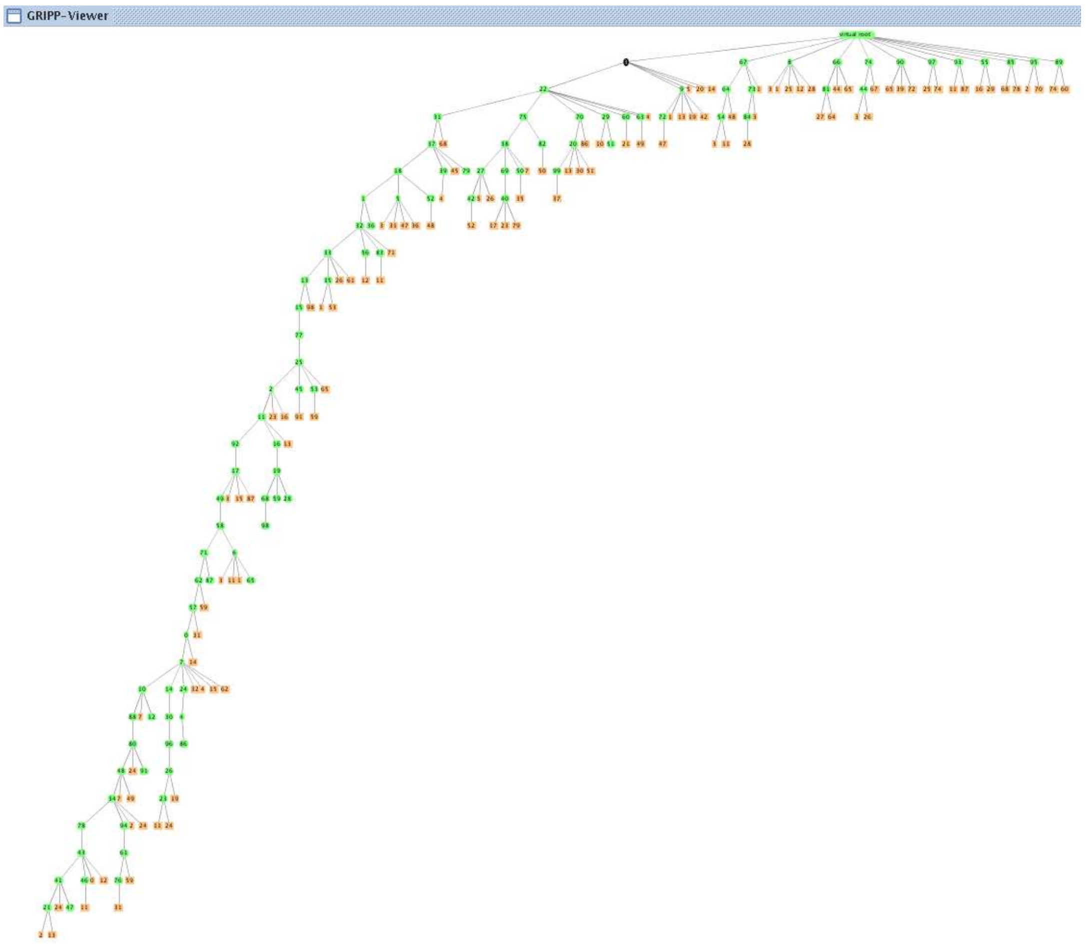


Figure 5: Order tree created by GRIPP for a graph of 100 nodes and 200 edges.

4 Querying GRIPP

In the following chapter we show how to use GRIPP to efficiently answer reachability and distance queries for a fixed pair of nodes. As answering distance queries for a fixed pair of nodes first requires to know if a path between the two nodes exists we first concentrate on reachability queries and then proceed to answering distance queries.

Recall, in trees both query types can be answered with a single lookup because all reachable nodes of a node v have a preorder value that is contained within the borders given by v_{pre} and v_{post} and $dist(v, w) = w_{depth} - v_{depth}$.

When querying the GRIPP index structure in this way, we face two problems. First, v has multiple instances in $IND(G)$, each with its individual pre- and postorder value. Second, in the preorder range of an instance v' we will only find instances of nodes that are reachable from v' in $O(G)$. Nodes reachable from v in G but not from v' in $O(G)$ will be missed. Thus, to find all reachable nodes in G, we have to extend the search by using the *hop technique*.

4.1 Hop technique

To evaluate $reach(v, w)$ and $dist(v, w)$ we use the GRIPP index table $IND(G)$. Every non-tree instance of v in $IND(G)$ is a leaf node in $O(G)$ and therefore has no successors in $O(G)$. But every node v also has one tree instance v^T in $IND(G)$. If v^T is an inner node in $O(G)$, v^T has reachable nodes w' in $O(G)$ such that $v^T_{pre} < w'_{pre} < v^T_{post}$. Those can be retrieved with a single query. We call this set *reachable instance set* of v.

Definition 4.1 (Reachable instance set) *Let $v \in V$ be a node of graph G and $v^T \in IND(G)$ its tree instance. The* reachable instance set *of v, written $RIS(v)$, is the set of all instances that are reachable from v^T in $O(G)$, i.e., have a preorder value in $[v^T_{pre}, v^T_{post}]$.*

Thus, the first step to answer $reach(v, w)$ is as follows. We first find the tree instance v^T of v and retrieve its reachable instance set. If $w' \in RIS(v)$, with w' instance of w, we finish and return $true$, otherwise we have to extend the search.

Recall that in $RIS(v)$ we only find instances that are reachable from v^T in $O(G)$, because during the creation of $IND(G)$ with reaching an already visited node we insert a non-tree instance in $IND(G)$ and do not traverse the child nodes of that node. Therefore, if $RIS(v)$ contains non-tree instances of nodes their child nodes might not have an instance in $RIS(v)$, i.e., these nodes are reachable from v in G, but not from v' in $O(G)$. To account for those we have to examine all non-tree instances of nodes in $RIS(v)$. We call those nodes *hop nodes* for v.

Definition 4.2 (Hop node) *Let $v, w \in V$ and w^N be a non-tree instance of w. If $w^N \in RIS(v)$ then w is called a* hop node *for v.*

Example 4.1 *Figure 6 shows the GRIPP index structure for the graph in Figure 3(a) plotted in a two-dimensional co-ordinate plane. When we query for $reach(D, C)$ we initially consider the reachable instance set of D. $RIS(D)$ contains non-tree instances of A and B, i.e., both are hop nodes for D.*

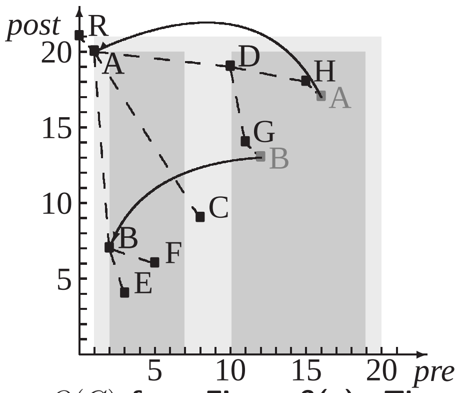


Figure 6: The figure shows $O(G)$ from Figure 3(a). The preorder ranges of $RIS(D)$ and $RIS(B)$ are in darkgray, the range of $RIS(A)$ is in lightgray. Nodes A and B are hop nodes for D. .

Every hop node in $RIS(v)$ has a reachable instance set in $O(G)$. The nodes in that set are reachable from v in G, but not from v^T in $O(G)$. Thus, we have to identify all hop nodes and recursively check their reachable instance sets. Therein, we basically perform a depth-first search over $O(G)$ using hop nodes in ascending order of their preorder values. We stop traversing $O(G)$ if we find an instance of node w or if there exists no further non-traversed hop node.

In $IND(G)$ there exist $|E| - |V|$ non-tree instances, each of which can be a hop node. Thus, querying GRIPP for $reach(v, w)$ requires in worst case $|E| - |V|$ queries. This is better than a depth-first traversal of G, as this requires in worst case $|E|$ traversals. Furthermore, we can save most of those queries by intelligent pruning.

4.2 Reachability queries

Example 4.2 *Consider Figure 6 and* $reach(D, R)$. *We find non-tree instances of nodes* A *and* B *in* $RIS(D)$. *If we first use* A *as hop node, we find non-tree instances of* A *and* B *in* $RIS(A)$. *Clearly, we do not need to use* A *as hop node again. Therefore, we next use* B *as hop node. The tree instance of* B *is a successor of the tree instance of* A *in* $O(G)$. *This implies that* $RIS(B)$ *is contained in* $RIS(A)$, *i.e., we will not find new instances in* $RIS(B)$ *that are not already contained in* $RIS(A)$. *Therefore, retrieving* $RIS(B)$ *is not necessary and can be pruned.*

In general we want to avoid posing queries for preorder ranges which we have already checked. During our search we keep a list U of all nodes that were used to retrieve a reachable instance set, i.e., the start node and the hop nodes. Now assume we have found a new hop node h. The decision whether we need to consider the reachable instance set of h entirely, partly, or not at all, depends on the location of the tree instance h^T of h relative to the tree instances of nodes in U.

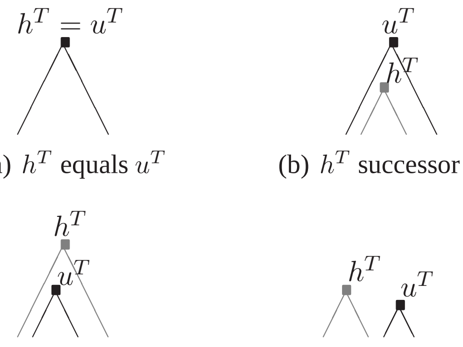
4.2.1 Pruning strategies for reachability queries

There are four possible locations of h^T in relation to the tree instance u^T of any node $u \in U$. These are shown in Figure 7. h^T either is

- (a) equal to,
- (b) a successor of,
- (c) an ancestor of, or
- (d) a sibling to u^T.

Since we may consider all nodes in U for pruning, these results in four possible cases: (a) h^T is equal to the tree instance of any node in U; (b) h^T is successor of the tree instance of at least one node in U; (c) h^T is ancestor of the tree instance of at least one node in U and neither (a) nor (b) is true; and (d) h^T is sibling to the tree instances of all nodes in U. Note that the pre- and postorder ranges of two instances can never overlap. They are either disjoint or one is entirely contained in the other.

In case (d), no pruning is possible and we have to consider the entire reachable instance set of h, as there exists no previous reachable instance set that covers instances in



(a) h^T equals u^T (b) h^T successor of u^T

(c) h^T ancestor of u^T (d) h^T sibling to u^T

Figure 7: Possible locations of h^T of hop node h relative to u^T, $u \in U$.

$RIS(h)$. For the remaining three cases we can apply pruning to ensure that no instance is considered twice during the evaluation of $reach(v, w)$.

In the first case (see Figure 7(a)), we can skip h entirely. A non-tree instance of h has already been used as hop node and therefore the reachable instance set of the tree instance of h has been checked.

In the second case (see Figure 7(b)) we also can skip h. In this case there exists $u \in U$ such that h^T is successor of u^T, i.e., $h^T \in RIS(u)$. Thus, the entire reachable instance set of hop node h is contained in $RIS(u)$ and has already been considered.

In the third case we have to be more careful.

Example 4.3 *Consider Figure 6 and the query $reach(D, R)$. Assume, we have retrieved $RIS(D)$ and $RIS(B)$ and have expanded the search using A as hop node. $RIS(A)$ contains the tree instance of B and D and therefore also contains $RIS(B)$ and $RIS(D)$. Thus, when we consider $RIS(A)$ we can 'skip' the range of $RIS(B)$ and $RIS(D)$.*

Skip Strategy We first assume that only one u^T exists that is a successor of h^T. Thus, $RIS(u)$ is contained in $RIS(h)$. This situation is displayed in Figure 7(c). Considering the entire reachable instance set of h leads to duplication of work. To avoid this we use the *skip strategy* working as follows. For every node $u \in U$ we store the pre- and postorder value of u^T, i.e., the borders of $RIS(u)$. In that range all instances are covered by $RIS(u)$ and we can skip the preorder range without missing instances. We only have to consider instances from $RIS(h)$ whose preorder values lie outside the pre- and postorder range of u^T.

If there is more than one successor node of h in U, the situation is slightly more complicated. Essentially, we can skip all their ranges when searching $RIS(h)$. This could be optimized by merging ranges iteratively during the search, thus reducing the number of necessary interval operations. However, we currently do not merge ranges.

We could merge ranges in U only for cases (c) and (d). In case (c) the tree instance of the hop node h is ancestor to tree instances of nodes in U. We could shorten U by deleting all nodes u that have a tree instance in $RIS(h)$. But as deletion operations are expensive in RDBMS we currently do not merge ranges in that context. In case (d) ranges can be adjoining, i.e., the $u^T_{1post} + 1 = u^T_{2pre}$. In that case we could merge those two entries. But as this is computationally more expensive than to skip both ranges separately we also do not merge ranges in that case. In addition, we search list U only a few times during a reachability query (shown in Section 7), i.e., the cost to merge ranges might not account for the gain of merging.

Stop Strategy When querying graphs for reachability between nodes v and w we can stop extending the search as soon as we have found an instance of w in the reachable instance set of the current hop node h. But if $w \notin RIS(h)$ we must find every hop node in $RIS(h)$ and start a recursive search. It would be advantageous if we knew in advance that in $RIS(h)$ does not exist a hop node that will extend the search, because in that case we do not have to query for the tree instance of every hop node. We now show cases where this property can be pre-computed.

Recall that a hop node for node s is a node h that has a non-tree instance in $RIS(s)$. h is not used as hop node if the tree instance of h is in $RIS(s)$ (Figures 7(a), 7(b)). We can precompute a list of nodes S for which all hop nodes have this property. We call those nodes *stop nodes* as their reachable instance sets will not extend the search.

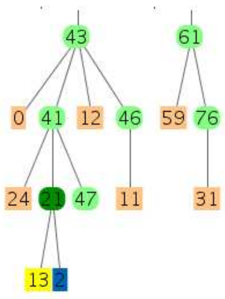
Definition 4.3 (Stop node) *Let $s \in V$ be a node of graph G and let $RIS(s)$ be its reachable instance set in $O(G)$. s is called a* stop node *iff all non-tree instances in $RIS(s)$ also have their corresponding tree instances in $RIS(s)$ or are a non-tree instance of s.*

Intuitively, a stop node s is a node in G for which in $RIS(s)$ for every non-tree instance there exists a corresponding tree instance. This means, that all nodes reachable from s in G are reachable from s^T in $O(G)$, i.e., have an instance in $RIS(s)$. Clearly, nodes reachable from s in G can also have non-tree instances in other reachable instance sets than $RIS(s)$.

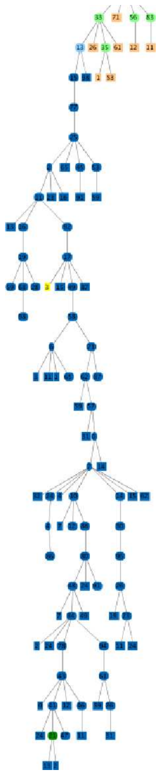
When we reach the tree instance of s during the search we immediately know that we need not extend the search further using hop nodes of $RIS(s)$. We only have to check if $w \in RIS(s)$. The GRIPP index structure in Figure 3 contains several stop nodes, namely nodes R, A, B, E, F, and C. As heuristic, during the search we prefer stop nodes as hop nodes over non-stop nodes.

Example 4.4 *As an illustration for a complex search process Figure 8 shows the evaluation of the reachability query $reach(21, 52)$ on a graph with 100 nodes and 200 edges. The query starts by considering the reachable instance set of node 21. In $RIS(21)$ there are two hop nodes, namely 13 and 2. As 13 has the lower preorder value we use this node as next hop node. $RIS(13)$ contains the tree instance of 21, i.e., we skip that range during the search. In $RIS(13)$ there are several non-tree instances, including a non-tree instance of stop node 3. Therefore, we use that node as next hop node. $RIS(3)$ contains an instance of node 52, i.e., we can return `true`.*

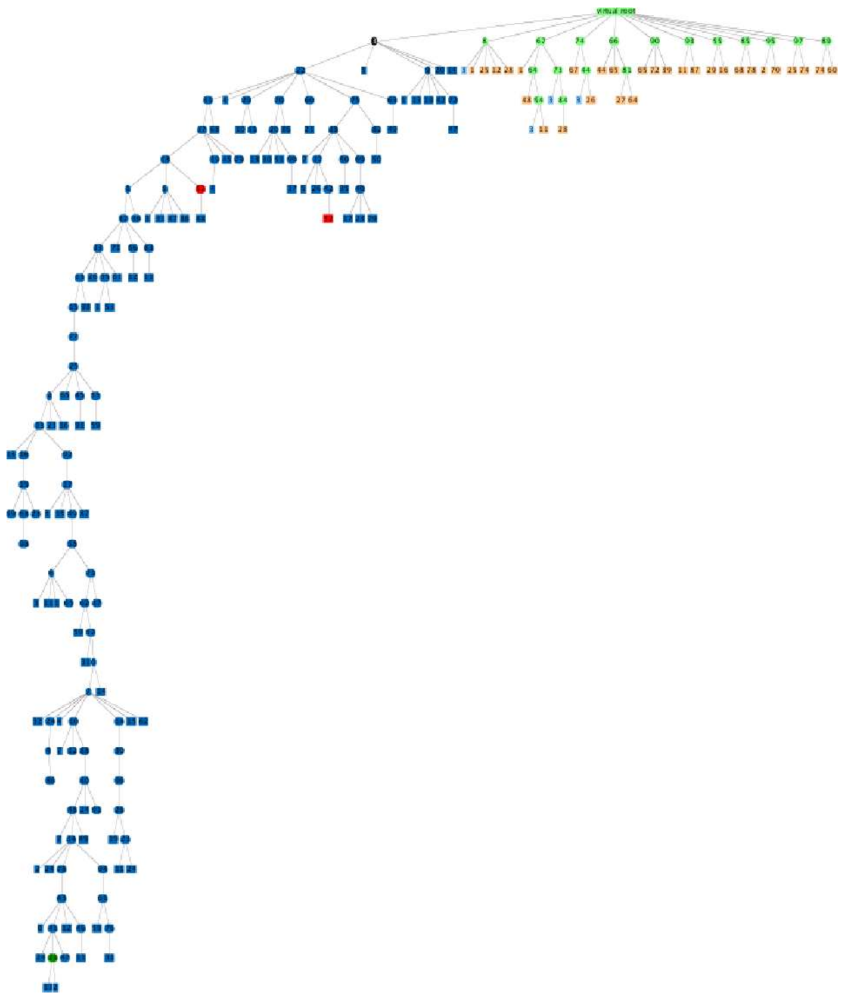
If we had not found node 52 in $RIS(3)$ we could also stop our search in this case, as node 13 as well as 21 are successor nodes of 3 in the order tree. This means no non-tree instance in a reachable instance set would point to a tree instance outside $RIS(3)$, i.e., we could not find an instance of node 52.



(a) Start at node 21 (in dark).



(b) Hop node 13.



(c) Hop node 3 (also stop node).

Figure 8: $reach(21,52)$ **on a generated scale-free graph with 100 nodes and 200 edges. In (a)** $RIS(21)$ **is dark. The non-tree instance of the next hop node 13 is light-colored. In (b)** $RIS(13)$ **is dark. The one non-tree instance of stop node 3 is light-colored, which is used as next hop node. In (c)** $RIS(3)$ **is dark. Two instances of the end node 52 are in** $RIS(3)$**.**

4.3 Distance queries

To answer $dist(v, w)$ using GRIPP we begin at node v and traverse the index structure using hop nodes. During the traversal we search for an instance of w in the reachable instance set of the start node or of hop nodes. If we find an instance of w we can determine the path length from v to w using GRIPP. As this path may not be the shortest, we have to traverse the index structure further. Applying a naive approach we have to systematically use every non-tree instance as hop node. We stop when no more unused non-tree instance is available. The length of the shortest path found is the distance betwen v and w.

In the following we first explain how to determine path lengths between two nodes using GRIPP. Later we will show how to apply different pruning strategies to make the evaluation of distance queries more efficient.

4.3.1 Determine path lengths

To determine the length of a path we need the depth of nodes in the GRIPP order tree $O(G)$. Assume two nodes v and w. If an instance w' of w is element of $RIS(v)$ we know that (a) w is reachable from v and (b) one path between v and w has the length $w'_{depth} - v^T_{depth}$ with v^T tree instance of v and w' any instance of $w \in RIS(v)$. This is not necessarily the distance between the two nodes, as a shorter path may exist through hop nodes.

Example 4.5 *Figure 9 shows $O(G)$ for the graph in Figure 3(a) with the depth of the nodes. When querying for $dist(D, E)$ we first retrieve $RIS(D)$, which contains two hop node, namely A and B. The path length from the tree instance D^T of D to the non-tree instances A^N and B^N of A and B, respectively, is in both cases 2 ($A^N_{depth} - D^T_{depth} = 2$ and $B^N_{depth} - D^T_{depth} = 2$). $RIS(D)$ does not contain the end node E, but we can extend the search using A or B as hop nodes.*

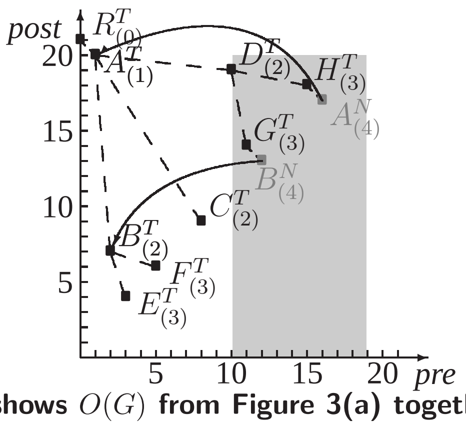


Figure 9: The example shows $O(G)$ from Figure 3(a) together with the depth of the nodes. The preorder range of $RIS(D)$ is in darkgray. Nodes A and B are hop nodes.

If w is not in $RIS(v)$ we have to extend the search using hop nodes. We can determine the path length from the tree instance v^T of v to the non-tree instance h^N_1 of the first hop node h_1. If there exists no instance of w in $RIS(h_1)$ we proceed with traversing $O(G)$

using further hop nodes h_i until we find an instance w' of w. To determine the path length of the path p starting at v, containing hop nodes $h_1 ... h_n$ and ending at w we have to sum up the path lengths for every part of the path as shown in Equation 1.

$$plen(v, w) = len(v^T, h_1^N) + \sum_{i=1}^{n-1} len(h_i^T, h_{i+1}^N) + len(h_n^N, w') \tag{1}$$

with $h_i \in p, 1 \leq i \leq n$, $len(a, b) = b_{depth} - a_{depth}$ with b successor of a in $O(G)$.

Example 4.6 *To evaluate $dist(D, E)$ on the GRIPP index structure shown in Figure 9 we first use node A as hop node. We find $E^T \in RIS(A)$ with $len(D^T, A^N) + len(A^T, E^T) = 2 + 2 = 4$. As next step we use B as hop node. We also find $E^T \in RIS(B)$, in this case with $len(D^T, B^N) + len(B^T, E^T) = 2 + 1 = 3$.*

There are no further unused hop nodes in $RIS(D)$. There are two non-tree instances in $RIS(A)$, i.e., A and B, which we have already used with a lower distance. We will prune hop nodes A and B and therefore $dist(D, E) = 3$.

4.3.2 General query strategy for distance queries

To determine the distance between nodes v and w we use the following query strategy. We first answer $reach(v, w)$ as described in Section 4.2. If $reach(v, w) = false$ we stop and return $dist(v, w) = null$. Otherwise, we determine the length of the path found when computing $reach(v, w)$ as first upper bound for the distance.

In the second step we perform a breadth-first search over $O(G)$ starting at v. We traverse $O(G)$ by using hop nodes h_i in ascending order of the path length between v and h_i. Be aware, this does not mean that we use hop nodes in the order they are found during the search (see also Example 4.7). We stop traversing $O(G)$ as soon as no further hop node can be used. The length of the shortest path is the distance between v and w.

Example 4.7 *Figure 10 shows a distance query from node v to w. We first answer $reach(v, w)$. We use h_1 as first hop node and find two instances of w in $RIS(h_1)$.*

We now start the breadth-first search by using hop nodes in ascending order of the path length between v and hop nodes. We first use h_1 as hop node as $plen(v, h_1) < plen(v, h_3)$. $RIS(h_1)$ contains a non-tree instance of h_2. We use the node as hop node that has the shortest path length to v^T. As $plen(v, h_2) = 5$ and $plen(v, h_3) = 7$ we use h_2 as next hop node. Finally, we use h_3 as hop node. As there are no further hop nodes the distance between v and w is the shortest path length found.

4.3.3 Pruning strategies for distance queries

For reachability queries only the location of the tree instance is important to decide if we can prune a hop node h. In Section 4.2 we identified four possible locations of the tree instance of a hop node h' in relation to reachable instance sets of used hop nodes U (Figure 7).

For distance queries in addition to the location of the tree instance of hop node h we also have to consider the path length between the start node and h. We have to compare

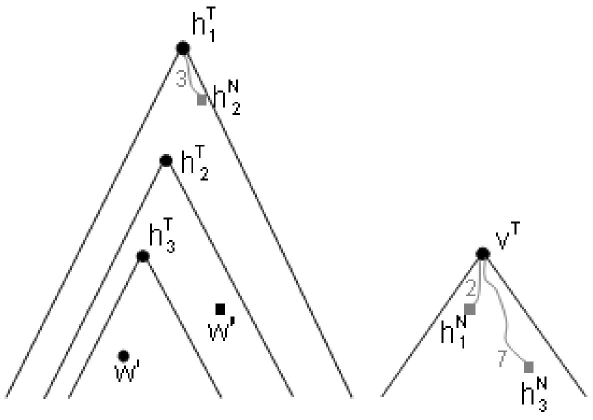


Figure 10: Evaluating distance queries. We use hop nodes in order of their subscripts. $len(v^T, h_1^N) = 2$**,** $len(v^T, h_3^N) = 7$**, and** $len(h_1^T, h_2^N) = 3$

$plen(v, h)$ to all path lengths between v and h over nodes in U. For that reason we store for every $u \in U$ the depth in $O(G)$ (u_{depth}) and the path length (u_{plen}) between v and u.

Example 4.8 *Consider Figure 9 and $dist(D, E)$. $RIS(D)$ contains non-tree instances of nodes A and B, both with the same path length to D. We use node A as first hop node and find E with a path length of 4. When querying for reachability we will not use B as hop node, as the tree instance is successor to a used hop node. However, for distance queries we have to use B as hop node. The path length between D and B is two, the currently shortest path between D and E is 4. Thus, using B as hop node can result in a shorter path. In this case the path between D and E over B is 3.*

We now show for all four cases individually when we can prune hop nodes.

h^T* equals *u* $\in$ *U In case that the tree instance h^T of the hop node h is equal to the tree instance u^T of a node $u \in U$ we have already seen all instances in $RIS(h)$. But if $plen(v, h) \neq plen(v, u)$ the path lengths between v and nodes in $RIS(u)$ are incorrect. If $plen(v, h) \geq plen(v, u)$ we do not have to use h as hop node for distance queries as the path lengths between v and nodes in $RIS(u)$ would only increase. Otherwise, if $plen(v, h) < plen(v, u)$ we must use h as hop node, as we have to adjust the previously computed path lengths between v and nodes in $RIS(h)$.

During a breadth-first search we will never get the situation that $plen(v, h) < plen(v, u)$ as we use non-tree instances in ascending order of their path length to v, i.e., $plen(v, h) \geq plen(v, u)$ always holds and we can therefore always prune.

Example 4.9 *That case is displayed in Figure 11. $RIS(v)$ contains two non-tree instances, i.e., u^N and h^N, with $len(v^T, u^N) < len(v^T, h^N)$. If we use u^N first we add u to the list of used nodes U and retrieve $RIS(u)$. As next non-tree instance we consider h^N and find that $h = u$. As $len(v^T, u^N) < len(v^T, h^N)$ we do not have to use h to retrieve $RIS(u)$ again. Otherwise, if we used h^N first we also had to use u^N as we had to adjust the path lengths to nodes in $RIS(h)$.*

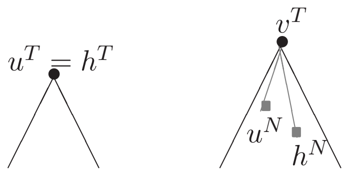


Figure 11: Case $h = u$ for distance queries.

h^T successor of $u \in U$ In the second case h^T is successor of the tree instance u^T of at least one node in U. For reachability queries we can prune that case entirely, as $RIS(h)$ is contained in at least one $RIS(u)$. For distance queries we also have to compare the path length from v directly to h to the path length from v over u to h, i.e., $plen(v, h)$ and $plen(v, u) + len(u^T, h^T)$.

If $plen(v, h) \neq plen(v, u) + len(u^T, h^T)$ we have to adjust the path lengths in $RIS(u)$. If $plen(v, h) \geq plen(v, u) + len(u^T, h^T)$ the path lengths between v and nodes in $RIS(u)$ will remain constant or even increase. To answer distance queries we are not interested in longer path and therefore we will not use h as hop node. Otherwise, if $plen(v, h) < plen(v, u) + len(u^T, h^T)$ we must use h as hop node and adjust the path lengths between v and nodes in $RIS(u)$.

Example 4.10 *Consider Figure 12. $RIS(v)$ contains two hop nodes, namely u and h with $len(v^T, u^N) < len(v^T, h^N)$. We use u as first hop node. In the next step we consider h^N. We find that h^T is successor of u^T. We reach h^T over two different paths, one directly from v to h, and one from v over u to h. Therefore there exist two different path lengths from v to h.*

- $plen(v, h) = plen(v^T, h^N)$
- $plen(v, h) = plen(v^T, u^N) + len(u^T, h^T)$

We have to use h as hop node only if the path between v and h over u is longer than the path directly to h. In every other case we can prune.

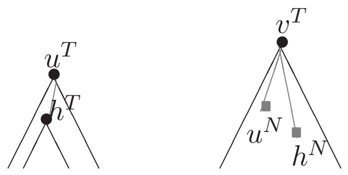


Figure 12: Case h^T successor of u^T for distance queries.

h^T ancestor of $u \in U$ In the third case is h^T is ancestor of the tree instance u^T of a node in U and neither of the previous two cases are true. For reachability queries we exclude the range between pre- and postorder value of every node $u \in RIS(h)$. For distance queries we must consider the path length between v and h to decide if we can skip a preorder range.

If $plen(v,h) + len(h^T, u^T) \geq plen(v,u)$ the path lengths between v and nodes in $RIS(u)$ will remain constant or even increase, i.e., we can skip the area. Otherwise, if $plen(v,h) + len(h^T, u^T) < plen(v,u)$ we cannot skip the area and must adjust the path lengths between v and nodes in $RIS(u)$.

Example 4.11 *Consider Figure 13. Here again, $RIS(v)$ contains two hop nodes, u and h with $len(v^T, u^N) < len(v^T, h^N)$. We use u as first hop node. In the next step we use h as hop node. As h^T is ancestor of u^T we reach u^T over two different paths, one directly from v to u, and one from v over h to u. Therefore there exist two different path lengths from v to u.*

- $plen(v,u) = plen(v^T, u^N)$
- $plen(v,u) = plen(v^T, h^N) + len(h^T, u^N)$

If $len(v^T, u^N) < len(v^T, h^N) + len(h^T, u^T)$ we can skip the preorder range between u^T_{pre} and u^T_{post}, otherwise not.

During a breadth-first search of the index structure $plen(v,u) \leq plen(v,h)$ as we used u before we used h as hop node. As we use hop nodes in ascending order of their path lengths to v the non-tree instance u^N of u must have an equal or lower distance than the non-tree instance of h^N. Therefore during a breadth-first search we can always skip the preorder range of used hop nodes in $RIS(h)$.

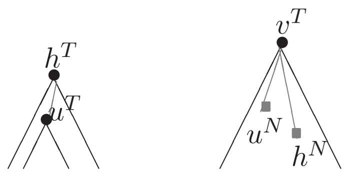


Figure 13: Case h^T ancestor of u^T for distance queries.

h**^{T}* **sibling to all *u* $\in$ *U In the last case is h^T sibling to all nodes $u \in U$. In this case we have to retrieve $RIS(h)$ regardless of the path length between v and h as we know nothing about instances in $RIS(h)$.

***plen(v,h)* $>$ *plen(v,w)*-2** When querying for $dist(v,w)$ we can also prune hop nodes if $plen(v,h) \geq plen(v,w)$ regardless of the location of the tree instance of the hop node. If we used h as hop node the path length between v and w would only increase. Actually, we can prune hop nodes if $plen(v,h) > plen(v,w) - 2$. If $plen(v,h) = plen(v,w)$ we can only find a path from v to w over h of $plen(v,w) = plen(v,w) + 1$, as we require at least one step to reach w in $RIS(h)$. Similarly, if $plen(v,h) = plen(v,w) - 1$ we can only find a path length over h of $plen(v,w) = plen(v,w)$, which is not shorter than the currently shortest path. Therefore we can prune if $plen(v,w) > plen(v,w) - 2$. In contrast, if $plen(v,h) = plen(v,w) - 2$ we could find a path between v and w of $plen(v,w) = plen(v,w) - 1$, i.e., we can not prune in that case.

In addition for a breadth-first search over GRIPP we have to put nodes on a stack. As we know that only hop nodes with $plen(v, h) > plen(v, w) - 2$ might contribute to shorter paths we do not have to put any hop node on the stack whose path length to v exceeds the current upper bound. Therefore, performing first a reachability query and returning an initial upper bound for the distance reduces the number of nodes that are put on the stack.

4.3.4 Distance queries in GRIPP – depth-first vs. breadth-first search

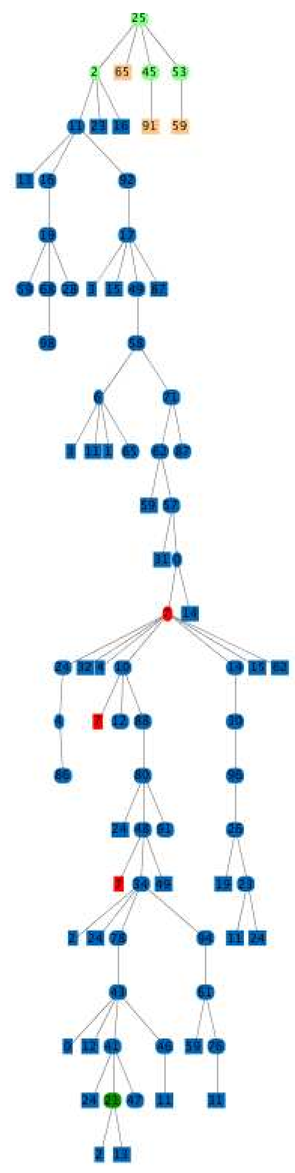
We can use two different search strategies for distance queries in GRIPP - depth-first or breadth-first search. Using depth-first search we can answer $reach(v, w)$ very fast using few hop nodes (experimentally verified in Section 7). Therefore for $dist(v, w)$ we can quickly determine a first upper bound for the distance and proceed the depth-first traversal. After we have found and instance w' of w we proceed using hop nodes whose non-tree instances are sibling to w' in $O(G)$. For such a hop node h it could be the case that $plen(v, h) > dist(v, w)$. This means that using h will not contribute to the result – as we will find shorter paths by using successive hop nodes. Concluding, using a depth-first search might lead to unnecessarily used hop nodes.

In contrast, during a breadth-first search we use hop nodes in ascending order of their distance to the query node. This means that we always use a hop node h with $plen(v, h) < dist(v, w)$, i.e., h might be on the shortest path between v and w. In addition, when using breadth-first search the pruning strategies for the cases $h^T = u^T$ and h^T ancestor of u^T are simpler, as we do not have to compare path lengths.

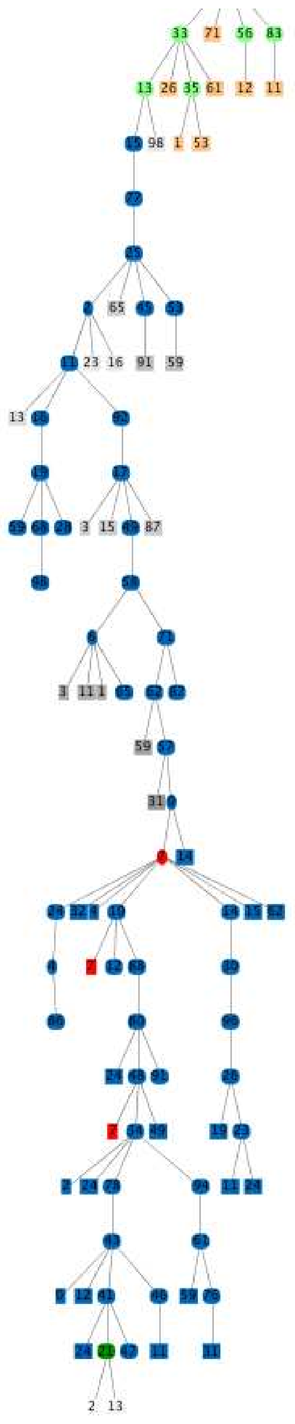
Example 4.12 *Figures 14 and 15 on pages 21 and 22 show the evaluation of a distance query on a graph with 100 nodes and 200 edges.*

To evaluate $reach(21, 7)$ *we first perform a reachability query. We start at node* 21 *and use node* 2 *as first hop node. We find three instances of node* 7 *in* $RIS(2)$*. The shortest path length is* 11.

In the next step we start the breadth-first search over the GRIPP index tree. During the search we add non-tree instances to the list of not traversed non-tree instances. The added non-tree instances up to path length 7 *are shown in Table 2. The table also reflects the progression during the breadth-first traversal.*

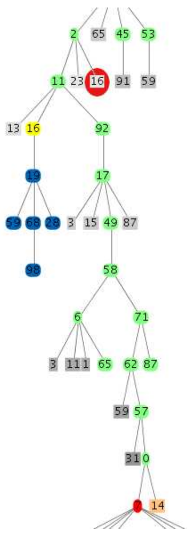


(a) Insert non-tree instances in $RIS(2)$.

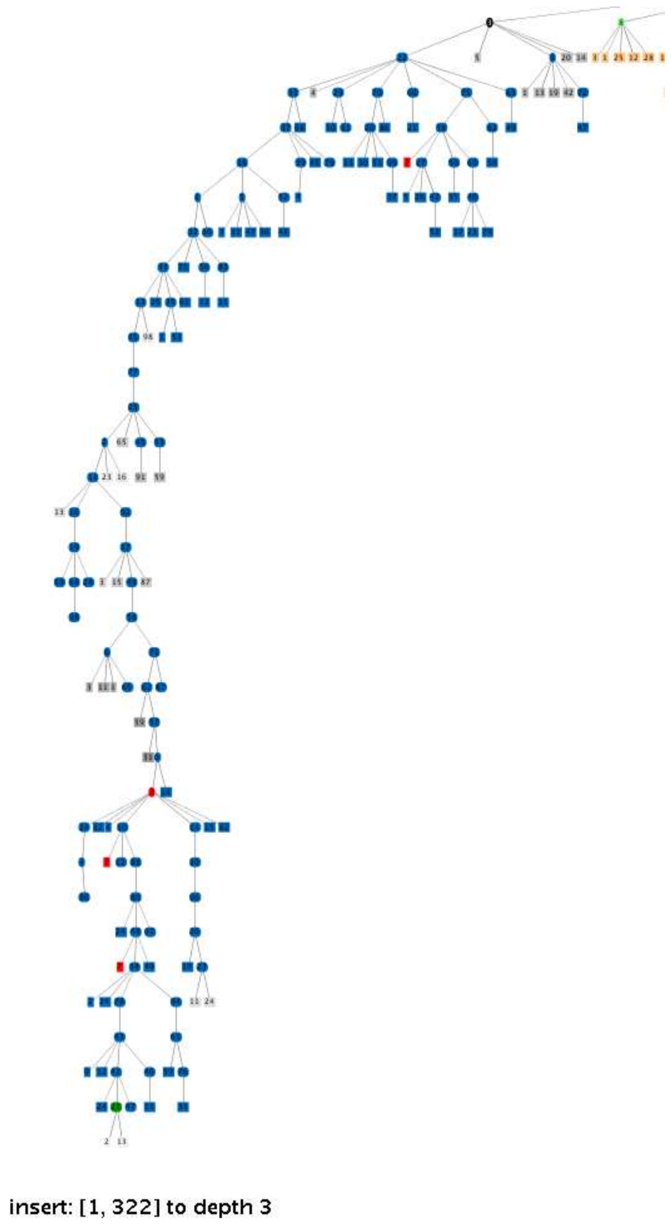


(b) Skip the already searched area of $RIS(2)$ and add remaining non-tree instances.

Figure 14: First two steps on the evaluation of $dist(21, 7)$.



(a) The tree instance of 16 is successor to the tree instance of 2 and 13. But we have to use 16 as hop node as the path lengths between 21 and nodes in $RIS(16)$ decreases.



(b) In $RIS(3)$ there is another instance of 7. The path length over hop node 3 is 9.

Figure 15: Two further hop steps during $dist(21, 7)$. The shortest path is $21 - 2 - 11 - 92 - 17 - 3 - 22 - 75 - 38 - 7$

plen	Search steps starting in with node 21													
1	2										13			
2	23		16								98			
3	11^{suc}	24		13^{eq}										
4														
5					3				15	87^{o}		65^{o}		
6					5	20	14						91^{o}	59^{o}
7					3^{eq}, 31, 47^{o}, 36^{o}	13^{eq}, 30^{suc}, 51^{o}		4, 1, 13^{eq}, 19, 42						

Table 2: Added non-tree instances to the list of not traversed non-tree instances. The table also shows the progression of the search. We did not use non-tree instances with superscript o = without successors, eq = equals a previous hop node, and suc = successor of a previous hop node and path length correct.

5 Heuristics for GRIPP

In this section we show that GRIPP is especially well suited for dense graphs. In GRIPP, mostly two criteria influence the performance of queries: (1) The order of child nodes during the index creation, and (2) the order in which hop nodes are used during the search phase.

5.1 Order of child nodes

Consider a query $reach(v, w)$. Clearly, the best GRIPP index structure would contain all reachable nodes from v in G in $RIS(v)$ and therefore the query could be answered with a single lookup. This is only the case when v has been traversed before all of its successors. We obviously cannot compute a special index structure for every possible start node. However, we can learn from this observation that a ’good’ order is one where nodes with many reachable nodes in G also should have large reachable instance sets in $O(G)$, i.e., that these nodes should be traversed early during index creation. With such nodes, we scan large fractions of the graph with few queries. This helps in pruning hop nodes.

This criteria can be satisfied easily in scale-free graphs, which contain few highly connected nodes (called hubs in the following) and many sparsely connected nodes. Hubs have many incoming and outgoing edges and a high chance of having a large set of successor nodes. To ensure that hubs also get a large reachable instance set we need to traverse them early during the GRIPP index creation. We achieve this goal by choosing child nodes in the order of their degree during index creation. As another positive effect, hubs are also reached by many nodes. Thus, they tend to appear early as hop node in the search phase, even if the start node v of a query is not a hub. Thereby, the search quickly reaches a node very close to the root of the order tree. Ordering nodes according to their degree is advantageous for all types of graphs, not only for scale-free graphs. In Section 7 we show the influence of the graph type on query performance empirically.

5.2 Order of hop nodes

The second criteria that influences the query performance is the order in which hop nodes are used during the search phase. Given node v, $RIS(v)$ can contain several hop nodes h. Following our explanation above the best strategy is to use the hop node that has the largest reachable instance set first. Clearly, this would be the best order in which to use hop nodes. But this strategy has a major disadvantage. In order to decide which hop node has the largest reachable instance set we need the pre- and postorder values of the tree instances for all hop nodes. As this is also time consuming we currently follow a different strategy, i.e., we use hop nodes in order of their preorder values of the non-tree instances. Clearly, we could precompute and store the size of the reachable instance set for every hop node, but experimental evidence shows that the number of recursive calls for this strategy increases only marginally.

5.3 Effect of node order on distance queries

For distance queries we perform a breadth-first search over GRIPP. We use hop nodes in the order of the path length to the query node. As explained in Section 4 this has the advantage, that we only use hop nodes that could be on a shortest path to the target node.

The weak point during the evaluation of distance queries is the index structure itself. Consider the query $dist(v, w)$. If we find an instance of w in $RIS(v)$ the path from v to w is not necessarily the shortest path. We still have to use all hop nodes in $RIS(v)$ with $plen(v, h) < plen(v, w)$. It would be advantageous to have an index structure where we knew for at least for some paths that these are the shortest. Appendix A shows such an index structure. For that structure we first perform a breadth-first search starting at any node and then create the index structure during a depth first search using the information from the breadth-first search. This has the advantage that every path in $O(G)$ between two tree instances is shortest, i.e., we could prune even more hop nodes. But there are two disadvantages, namely that with growing graph sizes the time required to execute the breadth-first search does not grow linear but exponential. In addition, querying this index structure for reachability requires more recursive calls and is therefore on average about 100 % slower than querying the index struture created by depth-first search alone (data not shown).

6 Implementation

In this section we present details on our implementation of GRIPP as stored procedures in a RDBMS. We explain how to deal with graphs with multiple or no root, describe how we compute the list of stop nodes, and sketch the search algorithms.

6.1 GRIPP index table

Before we create the GRIPP index we add a virtual root node r to the graph. We add an edge between r and the node that has the highest degree among all nodes. We then traverse and label the nodes as explained in Section 3 starting from r using the degree of nodes as order criteria. However, some nodes are not reached during this traversal, e.g., nodes without incoming edges or nodes in not connected subgraphs. We find those nodes and add another edge from r to the node with the highest degree. This is repeated until all nodes have at least one instance in the index table. This way, we uniformly handle graphs with none, one, or multiple root nodes.

Algorithm 1 shows the algorithm to compute the GRIPP index table $IND(G)$.

Example 6.1 *Figure 16(b) shows the GRIPP index strucutre that is created after applying Algorithm 1 to the graph in Figure 16(a) using child nodes ordered by node degree descending and node label ascending.*

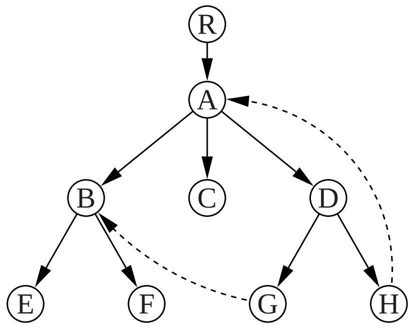
Algorithm 1: The GRIPP algorithm to compute $IND(G)$

```
pre_post ← 0 seen ← ∅
PROCEDURE compute_GRIPP()
    while ¬empty(node \ seen) do
        pre_node ← pre_post
        pre_post ← pre_post + 1
        next_node ← next(node \ seen) // order by degree
        traverse(next_node, 0)
        GRIPP ← GRIPP ∪ (next_node, pre_node, pre_post, 0, T)
        pre_post ← pre_post + 1
    end
end
PROCEDURE traverse(next_node, cur_dist)
    seen ← seen ∪ next_node
    while child ← next(children(next_node)) // order by degree
    do
        pre_node ← pre_post
        pre_post ← pre_post + 1
        if child ∉ seen then
            node_inst ← T
            traverse(child, cur_dist +1)
        else
            node_inst ← N
        end
        GRIPP ← GRIPP ∪ (child, pre_node, pre_post, cur_dist +1, node_inst)
        pre_post ← pre_post + 1
    end
end
```

6.2 Stop node list

To create the list of stop nodes would we have to check the reachable instance set of every node. As this is too time consuming we currently test only selected nodes. We are especially interested in nodes whose reachable instance set covers many instances. Therefore, we only consider child nodes c of the virtual root node as stop node candidates. In addition for every c we compute the size of $RIS(c)$, $|RIS(c)|$. We only consider c as stop node candidate if $|RIS(c)| \geq t$, with t being the cut-off value. For our experiments we use the cut-off value $t = 0.0005 * max(|RIS(c)|)$, which we determined empirically as tradeoff between the number of nodes we must evaluate during the stop node list generation and the number of stop nodes found. Furthermore, we only consider a node as stop node if it is a potential hop node, i.e., if it has a non-tree instance in $IND(G)$. For a stop node candidate s we check if the tree instance h^T of any hop node in $RIS(s)$ has a preorder value that is lower than that of the tree instance s^T of s. In that case, h^T is sibling to s^T in $O(G)$ and s is not a stop node; otherwise, s is a stop node and is added to the list of stop nodes.

Example 6.2 *Applying that heuristic to the GRIPP index structure from Figure 3(b) the*



(a) A graph G.

node	pre	post	depth	type
A	0	19	0	tree
B	1	6	1	tree
E	2	3	2	tree
F	4	5	2	tree
D	7	16	1	tree
G	8	11	2	tree
B	9	10	3	non-tree
H	12	15	2	tree
A	13	14	3	non-tree
C	17	18	1	tree
R	20	23	0	tree
A	21	22	3	non-tree

(b) $IND(G)$ created by Algorithm 1

Figure 16: Graph G and its GRIPP index table $IND(G)$. Solid lines in the graph represent tree edges, dashed lines are non-tree edges.

only stop node for the graph is node A.

Algorithm 2 shows the procedure to compute the list of the stop nodes. The child nodes to the root node are retrieved according to the size of their reachable instance sets.

Algorithm 2: The algorithm to compute the stop node list

```
PROCEDURE compute_stop_nodes(root_node)
    t ← 0
    while cand ← next(children(root_node)) // order by $|RIS|$
    do
        if t = 0 then
            t ← $post$(cand) $-pre$(cand)
        end
        if $post$(cand) $-pre$(cand) >t
          AND hasNon-tree(cand)
          AND stopNodeCond(cand) then
            STOP_NODES ← STOP_NODES ∪ ($node$(cand), $pre$(cand), $post$(cand));
        end
    end
end
FUNCTION stopNodeCond(cand)
    forall non_tree_inst ∈ RIS(cand) do
        tree_inst ← getTree(non_tree_inst)
        if tree_inst ∉ RIS(cand) then return false
    end
    return true
end
```

6.3 Search algorithm – Reachability.

The search phase is implemented as a stored procedure in a RDBMS. The GRIPP index as well as all temporary information (stop nodes, visited hop nodes, etc.) is stored in relational tables. The instance type of a node, i.e., tree or non-tree, is stored as special attribute. We created b-tree indexes on relevant attributes, including a combined index on the attributes preorder, node, and instance type. Given a query $reach(v, w)$, Algorithm 3 starts by adding v to the list U of used nodes. It then tests if $w \in RIS(v)$ with a query over the index table. If that is true the algorithm immediately returns `true`. Otherwise, it checks if v is a stop node. If that is the case, we know that (a) $RIS(v)$ does not contain the end node and (b) no hop node will extend our search and therefore return `false`.

If v is no stop node the algorithm checks if $RIS(v)$ contains a non-tree instance of a stop node. If so, the algorithm performs a depth-first search using this node as next hop node.

In the next step the algorithm searches for hop nodes u in $RIS(v)$. As the algorithm has already retrieved $RIS(u)$ we do not want to search the non-tree instances again. Knowing the pre- and postorder values of these instances u the algorithm can determine the preorder ranges for which non-tree instances have to be retrieved. These non-tree instances are used in ascending order of their preorder rank as next hop nodes to perform a depth-first search. For every hop node h we determine the location of its tree instance h^T and test if $RIS(h)$ is completely covered from reachable instance sets from nodes in U. If not, we pursue, using h as next hop node. We stop once we found an instance of w or if there are no more non-traversed hop nodes. All checks are implemented as relational queries.

Algorithm 3: Reachability queries on GRIPP index structure.

```
FUNCTION reachability(query, target)
    if target $\in$ RIS(query) then
        return true
    else
        used_hop $\leftarrow$ used_hop $\cup$ ($node$(query), $pre$(query), $post$(query))
        if query $\in$ STOP_NODES then
            used_stop $\leftarrow$ used_stop $\cup$ ($node$(query), $pre$(query), $post$(query))
            return false
        else
            while non_tree_inst $\leftarrow$ nextStop(RIS(query)) do
                tree_inst $\leftarrow$ getTree(non_tree_inst)
                result $\leftarrow$ reachability(tree_inst, target)
                if result $=$ true then return true
            end
            if query $\in$ RIS(used_stop) then return false
            used_hop_in_RIS $\leftarrow$ getUsedHopInRIS(query)
            i_left $\leftarrow$ $pre$(query)
            repeat
                next_used_hop $\leftarrow$ next(used_hop_in_RIS) // order by preorder
                if next_used_hop $\neq \emptyset$ then i_right $\leftarrow$ $pre$(next_used_hop)
                else i_right $\leftarrow$ $post$(query)
                if i_left $<$ i_right then
                    // get non-tree instances ordered by preorder
                    non_tree_instances $\leftarrow$ getNonTree(i_left, i_right)
                    foreach non_tree_inst $\in$ non_tree_instances do
                        tree_inst $\leftarrow$ getTree(non_tree_inst)
                        if hasChildren(tree_inst)
                         AND tree_inst $\neq$ used_hop
                         AND tree_inst $\notin$ RIS(used_hop) then
                            result $\leftarrow$ reachability(tree_inst, target)
                            if result $=$ true then return true
                        end
                        if query $\in$ RIS(used_stop) then return false
                    end
                end
                i_left $\leftarrow$ $post$(next_used_hop)
            until i_right $=$ $post$(query)
        end
        return false
    end
end
```

6.4 Search algorithm – Distance.

The search phase for $dist(v, w)$ is implemented as a stored procedure in a RDBMS. As for reachability queries the GRIPP index and the list of stop nodes as well as all temporary information (visited hop nodes, not used non-tree instances etc.) is stored in relational tables. The type and the depth of an instance in $O(G)$ are stored as special attributes in the index table. Given a query $dist(v, w)$, Algorithm 4 starts by first computing $reach(v, w)$ using an extended reachability search algorithm. If a path between v and w exists the algorithm proceeds in a second step with a breadth-first search to determine the distance between v and w.

The basic algorithm to determine $reach(v, w)$ shown in Algorithm 3 was extended to return a path length between v and w. The procedure shown in Algorithm 5 has as additional parameter the path length $plen$ between v and the query node. For the first call this path length is 0. The path length between the query node and the next hop node is $plen(query, hop) = plen + len(query, hop)$, with $len(query, hop) = hop_{depth} - query_{depth}$. As soon as the algorithm finds an instance of w in a reachable instance set it returns the path length between v and w. If the algorithm finds more than one instance of w in a set it returns the shortest path length.

If a path between v and w exists Algorithm 4 proceeds with a breadth-first search using the path length returned from the reachability search as first upper bound for the distance. For the breadth-first search it adds all non-tree instances in $RIS(v)$ together with the length of the path to v to the list of not used non-tree instances. As pruning criteria only non-tree instance are added to the list that have a path length that is shorter than the upper bound.

During the breadth-first search Algorithm 6 uses non-tree instances in that list in ascending order of their distance to v. For every non-tree instance it first retrieves the corresponding tree instance of the node. In the next step the algorithm checks if that node can be pruned. It first checks if the node has already been used as hop node (regardless the path length as we perform a breadth-first search). If yes, that node is pruned and the algorithm proceeds with the next non-tree instance. Otherwise, it checks if the tree instance of that node is successor to a previously used hop node. If yes, the algorithm also has to consider the path lengths. If the path length over the used hop node is shorter than this path the algorithm can prune that hop node. Otherwise, if the hop node is no successor or the path is longer that node is used as next hop node.

When Algorithm 6 uses a node as hop node that node is added to the list of used hop nodes and it is checked if its reachable instance set contains instances of the target node. If that is the case, the algorithm determines the shortest path length between the query and an instance of the target node. If that path is shorter than the previously shortest path it corrects the upper bound. In the next step the algorithm adds all non-tree instances of the reachable instance set of the hop node to the list of not used non-tree instances. But we do not want to add all instances, i.e., we want to leave out non-tree instances that are already covered by a reachable instance set of a used hop node and we do not add non-tree instances that are further away from the query node than the currently shortest path length. After the algorithm has added the remaining non-tree instances it proceeds with the next non-tree instance.

The algorithm terminates if there are no more non-tree instances in the list or the found path length between the query and the target node is lower than the path length between the next non-tree instance and the query node. In both cases the algorithm returns the currently shortest path length as distance between the query and the target node.

Algorithm 4: Breadth-first search for distance between two nodes.

```
FUNCTION distance(query, target)
    plen = plenReachability(query, null, target)
    if plen $\neq$ null then
        used_hop_plen $\leftarrow$ used_hop_plen $\cup$ ($node$(query), $pre$(query), $post$(query),
        $depth$(query), 0)
        foreach non_tree_inst $\in$ getNonTree($pre$(query), $post$(query)) do
            if len(query, non_tree_inst) $<$ plen then
                not_used_non_tree $\leftarrow$ not_used_non_tree $\cup$ ($node$(non_tree_inst),
                len(query, non_tree_inst))
            end
        end
        return distance_breadth(not_used_non_tree, used_hop_plen, plen, target)
    else
        return null
    end
end
```

Algorithm 5: Extended algorithm for reachability queries on GRIPP index structure.

```
FUNCTION plenReachability(query, plen, target)
    if target $\in$ RIS(query) then
        return min(plen + len(query, target)
    else
        used_hop $\leftarrow$ used_hop $\cup$ ($node$(query), $pre$(query), $post$(query))
        if query $\in$ STOP_NODES then
            used_stop $\leftarrow$ used_stop $\cup$ ($node$(query), $pre$(query), $post$(query))
            return null
        else
            while non_tree_inst $\leftarrow$ nextStop(RIS(query)) do
                tree_inst $\leftarrow$ getTree(non_tree_inst)
                plen $\leftarrow$ plenReachability(tree_inst, plen +len(query, non_tree_inst),
                target)
                if plen $\neq$ null then return plen
            end
            if query $\in$ RIS(used_stop) then return null
            used_hop_in_RIS $\leftarrow$ getUsedHopInRIS(query)
            i_left $\leftarrow$ $pre$(query)
            repeat
                next_used_hop $\leftarrow$ next(used_hop_in_RIS)
                if next_used_hop $\neq \emptyset$ then i_right $\leftarrow$ $pre$(next_used_hop)
                else i_right $\leftarrow$ $post$(query)
                if i_left $<$ i_right then
                    non_tree_instances $\leftarrow$ getNonTree(i_left, i_right)
                    foreach non_tree_inst $\in$ non_tree_instances do
                        tree_inst $\leftarrow$ getTree(non_tree_inst)
                        if hasChildren(tree_inst)
                         AND tree_inst $\neq$ used_hop
                         AND tree_inst $\notin$ RIS(used_hop) then
                            plen $\leftarrow$ plenReachability(tree_inst, plen +len(query,
                            non_tree_inst), target)
                            if plen $\neq$ null then return plen
                        end
                        if query $\in$ RIS(used_stop) then return null
                    end
                end
                i_left $\leftarrow$ $post$(next_used_hop)
            until i_right $=$ $post$(query)
        end
        return null
    end
end
```

Algorithm 6: Breadth-first search.

```
FUNCTION distance_breadth(not_used_non_tree, used_hop_plen, plen, target)
    while next_non_tree ← next(not_used_non_tree) do
        if plen < $plen$(next_non_tree)+1 then break
        next_tree ← getTree(next_non_tree)
        if next_tree ∉ used_hop_plen then
            if next_tree ∉ RIS(used_hop_plen) OR (next_tree ∈ RIS(used_hop_plen)
            AND $plen$(next_non_tree) < $plen$(used_hop_plen)+len(used_hop_plen,
            next_tree)) then
                used_hop_plen ← used_hop_plen ∪ ($node$(next_tree), $pre$(next_tree),
                $post$(next_tree), $depth$(next_tree), $plen$(next_non_tree))
                if target ∈ RIS(next_tree) then
                    new_len = $plen$(next_non_tree) + len(next_tree, target)
                    if new_len < plen then
                        plen = new_len
                        if plen < $plen$(next_non_tree)+1 then break
                    end
                end
                used_hops_in_RIS ← getUsedHopInRIS(next_tree)
                i_left ← $pre$(next_tree)
                repeat
                    next_used_hop ← next(used_hops_in_RIS)
                    if next_used_hop ≠ ∅ then i_right ← $pre$(next_used_hop)
                    else i_right ← $post$(next_tree)
                    if i_left < i_right then
                        non_tree_instances ← getNonTree(i_left, i_right)
                        foreach non_tree_inst ∈ non_tree_instances do
                            // do not add non-tree instances further away
                               from the query node than plen
                            if $plen$(next_tree) + len(next_tree, non_tree_inst) < plen
                            then
                                not_used_non_tree ← not_used_non_tree ∪
                                ($node$(non_tree_inst), $plen$(next_tree) + len(next_tree,
                                non_tree_inst))
                            end
                        end
                    end
                    i_left ← $post$(next_used_hop)
                until i_right = $post$(next_tree)
            end
        end
    end
    return plen
end
```

7 Experimental Results

To evaluate our approach we use synthetic as well as real-world data. We compare GRIPP to two other well known methods. For the index creation we compare GRIPP with the transitive closure. Clearly, querying the transitive closure would be fastest, but as we can not compute the transitive closure for large graphs, we also compare GRIPP with recursive query strategies.

We created random as well as scale-free synthetic graphs in the size of 1,000 to 5,000,000 nodes and 0 to 450% more edges than nodes using the method described in [3]. For real-world data we took data from metabolic and protein-protein interaction networks. We used the data from the metabolic networks of KEGG [17], aMAZE [19], and Reactome [16]. Nodes represent enzymes, chemical compounds or reactions, while edges represent the participation of an enzyme or compound in a reaction. For protein-protein interaction networks we used STRING [25]. Nodes are chemical compounds or biomolecules, i.e., DNA, RNA, or proteins and edges represent interactions between compounds or biomolecules. Edges in STRING are labeled with a confidence value for the protein-protein interaction. In STRING 95 we included edges with a confidence of 95 % or higher. In STRING 90 and STRING 75 we included edges with 90 % and 75 % confidence, respectively. We only included nodes with at least one edge in all three datasets. Table 3 shows the size of the different graphs. Note that in STRING 75 there are 7 times more edges than nodes.

Database	No. nodes	No. edges	Density
Metabolic networks			
Reactome	3,677	14,447	3.9
aMAZE	11,876	35,846	3.0
KEGG	14,269	35,170	2.5
Protein-protein interaction networks			
STRING 95	75,132	207,764	2.8
STRING 90	135,145	952,940	7.1
STRING 75	196,493	1,383,134	7.0

Table 3: Number of nodes and edges in biological networks.

We have implemented all algorithms as stored procedures in ORACLE 9i. Tests were performed on a DELL dual Xeon machine with 4 GB RAM. Queries were run without rebooting the database. We created b-tree indexes on all selection predicates of the GRIPP index table, including a combined index on the attributes preorder, node, instance type, and depth.

For every number of nodes, edges, and graph type we generated five different graphs. For every graph we created a GRIPP index structure and noted the time required to create the index structure and size of the generated structure.

7.1 Index Creation

We compare the time required to compute the GRIPP index with the time required to compute the transitive closure using the semi-naive algorithm from [21]. Note that in our experience the logarithmic algorithm is not faster in a RDBMS (data not shown).

No. nodes	**Scale-free graphs**			**Random graphs**		
	TC	GRIPP	Stop nodes	TC	GRIPP	Stop nodes
1,000	47.3	2.3	0.1	49.8	2.2	0.1
5,000	2,007.8	11.3	0.1	2,277.0	11.4	0.1
10,000	12,555.1	23.0	0.1	14,694.3	23.3	0.1
50,000	-	119.5	0.2	-	127.6	0.3
100,000	-	235.8	0.4	-	237.4	0.4
500,000	-	1,196.6	2.6	-	1,203.9	2.6
1,000,000	-	2,539.8	5.8	-	2,588.7	6.0
5,000,000	-	16,062.5	38.2	-	16,901.0	37.2

Table 4: Average time (sec) to compute the GRIPP index table and the transitive closure for synthetic graphs with 100 % more edges than nodes.

Table 4 shows the results for scale-free graphs with 1,000 to one million nodes and 100 % more edges than nodes. For graphs of 50,000 or more nodes we could not compute the transitive closure. For instance, for graphs with 50,000 nodes and 100,000 edges the computation did not complete within 24 hours. In contrast, computing the GRIPP index table for the same graphs took less than 120 seconds. The time for the stop node list for those graph is under one second.

The data show that GRIPP scales roughly linear in the number of nodes for a fixed density. For example, we computed the GRIPP index table for a scale-free graph with 5,000,000 nodes and 10,000,000 edges in less than 5 hours. This means that we can compute the GRIPP index table even for much larger graphs as we did.

No. edges	**Scale-free graphs**		**Random graphs**	
	GRIPP	Stop nodes	GRIPP	Stop nodes
100,000	168.3	120.0	169.1	185.4
150,000	199.8	0.6	200.3	0.6
200,000	235.8	0.4	237.4	0.4
250,000	277.1	0.4	276.8	0.4
300,000	313.8	0.5	316.0	0.5
350,000	349.1	0.6	353.7	0.5
400,000	388.0	0.7	390.3	0.6
450,000	505.1	0.7	554.3	0.7

Table 5: Average time (sec) to compute the GRIPP index table and the stop node list for synthetic graphs with 100,000 nodes and increasing number of edges.

Table 5 shows that GRIPP also scales roughly linear with increasing number of edges. For example, the computation of the GRIPP index table for 100,000 nodes and 400,000 edges took less than 400 seconds, compared to about 240 seconds for a graph with 100,000 nodes and 200,000 edges.

For the creation of the GRIPP index structure we also have to take into account the time required to compute the stop nodes as presented in Section 6.2. Table 4 shows that even for large graphs with fixed density of 2 the computation takes less than 40 seconds. For graphs with a fixed number of nodes and increasing density the time to compute the stop nodes decreases with increasing density (shown in Table 5). The reason for this is the number of stop node candidates that have to be evaluated. Graphs with extremely low density have many child nodes to the root node, i.e., many nodes have to be evaluated, while graphs with higher density have fewer child nodes to the root.

No. nodes	**Scale-free graphs**			**Random graphs**		
	TC	GRIPP	Stop nodes	TC	GRIPP	Stop nodes
1,000	619,231.6	2,181.2	1.0	637,401.6	2,151.6	1.0
5,000	15,137,809.8	10,885.0	1.0	15,686,250.8	10,766.6	1.0
10,000	60,918,470.4	22,006.5	1.0	62,858,373.2	21,784.3	1.0
50,000	-	110,199.3	1.0	-	109,149.3	1.0
100,000	-	218,482.8	1.0	-	215,554.4	1.0
500,000	-	1,092,203.6	1.0	-	1,092,203.6	1.0
1,000,000	-	2,184,524.6	1.0	-	2,156,309.2	1.0
5,000,000	-	10,922,541.4	1.0	-	10,782,940.6	1.0

Table 6: Average size (tuples) of the transitive closure, GRIPP index table, and stop node list for synthetic scale-free and random graphs with 100 % more edges than nodes.

Table 6 shows that the size of the GRIPP index table grows linear with the size of the graph. The GRIPP index table of a scale-free graph with 10,000 nodes and 20,000 edges contains about 22,000 instances. In contrast, the transitive closure of the same graph contains more than 60 million node pairs. For random graphs GRIPP requires about the same time and size as for scale-free graphs of the same size.

Table 7 shows the time and space required to compute the GRIPP index table on real-world graphs. The time required to compute the GRIPP index table for metabolic networks of Reactome, aMAZE, and KEGG and for protein-protein interaction networks of STRING corresponds well with the time required for synthetic networks of the same size.

The time to compute the stop node list for metabolic networks also complies with the time for synthetic networks of the same size. In contrast, for protein-protein interaction networks the time required to compute the stop node list is much higher than for generated graphs. The main reason is that all three networks of STRING are comprised of many unconnected subgraphs, i.e., the virtual root node has many child nodes. But this also means that during the stop node list generation we have to check many stop node candidates. This explains the high time consumption, as this step is time consuming.

Database	No. nodes	No. edges	GRIPP index		Stop nodes	
			Time	Size	Time	Size
Metabolic networks						
Reactome	3,677	14,447	14.1	14,910	0.3	23
aMAZE	11,876	35,846	37.2	37,636	0.1	1
KEGG	14,269	35,170	39.2	36,591	0.1	2
Protein-protein interaction networks						
STRING 95	75,132	207,764	225.9	225,868	163.1	3,178
STRING 90	135,145	952,940	851.1	967,838	139.7	477
STRING 75	196,493	1,383,134	1,237.0	1,404,139	196.4	492

Table 7: Time in seconds and storage space in tuples required to compute and store the GRIPP index table and the stop node list for real world graphs.

7.2 Query times for reachability queries

We compare querying GRIPP to answer reachability queries with a recursive depth-first search stopping as soon as the target node is found. For the comparison we randomly selected 1,000 pairs of nodes for every graph and computed $reach(v, w)$.

We also tested Oracle's 10g implementation of recursive SQL queries. It outperforms our own recursive function for very small and sparse graphs. However, it is extremely slow already for medium-sized graphs. A single query on a graph with 1,000 nodes and 1,500 edges took more than 7 hours to complete. The reason seems to be that Oracle enumerates all paths in the graph beginning from the start node and this number grows exponentially.

No. nodes	TC	recursive	GRIPP
1,000	1.0 ± 0.00	372.0 ± 297.37	2.2 ± 0.95
5,000	1.0 ± 0.00	1,810.9 ± 1,509.88	2.2 ± 1.01
10,000	1.0 ± 0.00	3,676.8 ± 3,010.51	2.3 ± 1.02
50,000	-	18,345.5 ± 14,989.95	2.3 ± 1.03
100,000	-	-	2.3 ± 1.04
500,000	-	-	2.3 ± 1.05
1,000,000	-	-	2.3 ± 1.05
5,000,000	-	-	2.3 ± 1.03

Table 8: Average number of calls to answer reach(v,w) for the three different query strategies on scale-free graphs.

Table 8 shows the average number of recursive calls for the different query strategies on scale-free graphs with 1,000 to 5,000,000 nodes and 100 % more edges than nodes. Clearly, we need only one lookup to answer reachability using the transitive closure. The number of recursive calls for the recursive query strategy depends on the size of the graph. For graphs of 1,000 nodes and 2,000 edges we required on average 372 recursive calls, ranging from 1 call for a node without child nodes to 795 calls in worst case. This also explains the high standard deviation.

When querying graphs using GRIPP the number of recursive calls remains almost

No. nodes	TC	recursive	GRIPP
1,000	0.4 ± 0.08	242.1 ± 201.11	2.8 ± 1.23
5,000	0.5 ± 0.11	$1{,}383.4 \pm 1{,}193.34$	3.0 ± 1.44
10,000	0.5 ± 0.67	$3{,}283.1 \pm 2{,}777.78$	3.0 ± 1.43
50,000	-	$34{,}062.9 \pm 28{,}210.27$	3.6 ± 1.87
100,000	-	-	3.2 ± 1.44
500,000	-	-	3.6 ± 1.65
1,000,000	-	-	3.8 ± 1.77
5,000,000	-	-	4.5 ± 3.02

Table 9: Average query time (ms) to answer reach(v,w) for the three different query strategies on scale-free graphs.

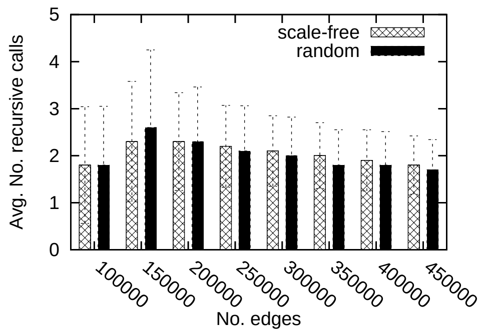
constant over different sizes of graphs. The maximum number of recursive calls ranges from 6 to 9 for different sizes of scale-free graphs. This is surprising, as we would expect that the number of calls depends on the number of non-tree instances in $IND(G)$, i.e., that for GRIPP the number of recursive calls increases with growing size of the graph.

We can explain that behavior by the following consideration. When querying for $reach(v, w)$ we start with $RIS(v)$ and extend the search using hop nodes. We only use hop nodes whose tree instance (a) is sibling to or (b) ancestor of the tree instance of v. This also means, that we constantly exclude more and more nodes from being used as hop node. As we preferably use a stop node as hop node we quickly cover the vast majority of the instances in $IND(G)$. Clearly, in worst case we have to use as many hop nodes as unique nodes have non-tree instances in $IND(G)$. But our results show that in synthetic as well as real-world networks this is not the case.

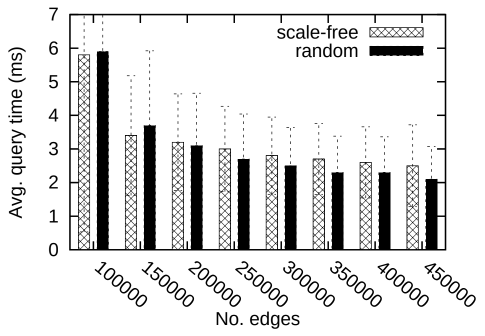
The query times (shown in Table 9) for the different strategies correspond well with the number of recursive calls. For GRIPP the average query times range from 2.8 to 4.5 ms for scale-free graphs. For example for 50,000 nodes and 100,000 edges querying GRIPP requires on average about 3.6 ms compared to 34,100 ms for querying the graph recursively. The time difference between GRIPP and recursive query strategies grows as the size of G increases.

Figure 17 shows the average number of calls and average query times on scale-free and random graphs of 100,000 nodes and 100,000 to 450,000 edges. For both types of graphs the average number of calls and average time decreases with increasing graph density. This can be explained as follows. With increasing graph density the number successor nodes of the node with the highest degree also increases. Remember, we traverse this node first during the index creation. If this node has incoming edges it is a stop node. Therefore, when we reach a stop node during a reachability search we cover more and more nodes with increasing graph density. And as the number of edges increases it is more and more likely to find an instance of the stop node in a reachable instance set.

For graphs up to 150,000 edges querying GRIPP has advantages on scale-free graphs. For denser graphs GRIPP performs better on random graphs. This behaviour can also be explained with the number of successor nodes of the node with the highest degree. During the generation of scale-free graphs a node with many incoming and outgoing edges is likely to get more edges, while in random graphs nodes for new edges are chosen randomly. In sparse scale-free graphs most highly connected nodes are reachable from the first traversed node, but this also means that more nodes are reachable in sparse scale-free graphs than in random graphs. In denser graphs this reverses as in scale-free graphs it is more likely that a new edge is added between (well connected) nodes that are both already reachable from the node with the highest degree. In contrast, in random graphs the nodes for the new edge are chosen randomly, i.e., giving the possibility to enlarge the set of successor nodes. Therefore, the number of successor nodes of the first traversed node grows faster for random graphs than for scale-free graphs with increasing graph density and this means that queries can be answered faster.

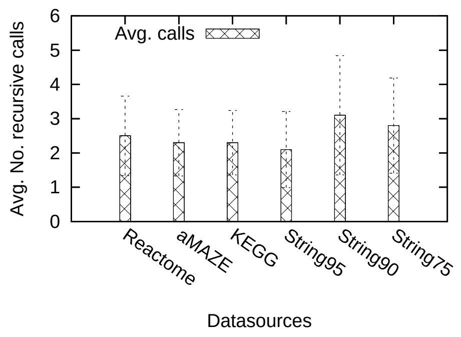


(a) Calls.

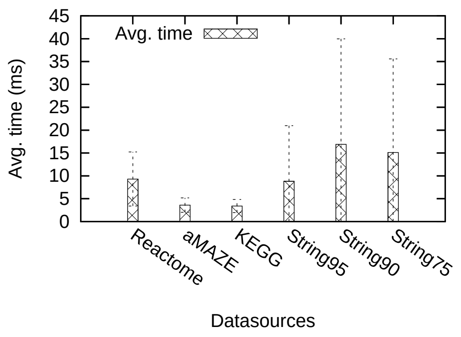


(b) Query time.

Figure 17: Average query time and average number of calls for synthetic scale-free and random networks of 100,000 nodes and increasing number of edges using GRIPP.



(a) Calls.



(b) Query time.

Figure 18: Average query time and average number of calls for real-world networks using GRIPP.

Figure 18 shows the average number of calls and average time for reachability queries on real-world networks. The average number of calls and average query time for the metabolic networks of Reactome, aMAZE, and KEGG is slightly higher than the number for synthetic scale-free graphs. This indicates that, although the networks are scale-free, they still have a different structure than our synthetic graphs. For the protein-protein interaction database STRING the number of recursive calls is only slightly higher than the number for synthetic scale-free or random graphs of comparable size while the average query time is much higher. This can be explained by the following observation. In STRING every interaction between two proteins is represented as two directed edges, i.e., one leading from protein 1 to protein 2 and one from protein 2 to protein 1. In the order tree of GRIPP we therefore always find a non-tree instance of protein 1 in the reachable instance set of protein 1. Clearly, we must evaluate if we need protein 1 as hop node, which is not the case. As this testing also takes time the average time for reachability queries increases while the number of calls remains low.

7.3 Query times for distance queries

We measured query performance for distance queries on generated random and scale-free graph of different sizes. We compared GRIPP with recursive query strategies. We have implemented the query strategy for GRIPP as described in Section 6.4. We compare that approach with two different breadth-first search strategies as stored procedures in Oracle.

7.3.1 GRIPP against breadth-first search

We have implemented two different approaches for the breadth-first search. The first approach (breadth-first single) is the standard implementation of a breadth-first search. Given a query node, all child nodes of that node are added to the stack in arbitrary order. The nodes on the stack are processed according to their order on the stack. We add the child nodes of every processed node to the stack if that node is or has not been on the stack. The algorithm terminates as soon as we find the target node as child node or if no more nodes are on the stack.

The second approach is a set based approach (named as breadth-first set). In the first step we add all child nodes of the query node together with the distance 1 to the stack. Instead of processing every node separately we process all nodes with the same distance to the query node at once. We use a single SQL statement to process all nodes with distance i on the stack and add the child nodes of these nodes that are not already on the stack to the stack with distance $i + 1$. In the next step we process all nodes with distance $i + 1$. The algorithm terminates if no more nodes are on the stack or if a child node is the target node and then the algorithm returns the distance.

Table 10 shows the average number of calls for 1,000 randomly selected node pairs for the different methods. For GRIPP the number of recursive calls consists of the number of hop nodes required to determine reachability plus the number of hop nodes required during the breadth-first search. The number of calls for the standard breadth-first search is the number of nodes for which we retrieved and added child nodes to the stack and the number of calls for the set based breadth-first search is the number of SQL queries.

The comparison between GRIPP and the standard breadth-first search shows that on average queries on GRIPP require an order of magnitude less calls than using breadth-first search. This can be explained as follows. For a standard breadth-first search we have to use every node in the graph for querying i.e., in worst-case the total number of nodes in the graph. In contrast, during a breadth-first search in GRIPP we use every hop node at most once, i.e., in worst-case as many hop nodes as unique nodes in the graph have

No. nodes	Average distance	GRIPP	breadth-first single	breadth-first set
Scale-free networks				
1,000	6.25	22.0 ± 37.9	370.1 ± 297.3	6.3 ± 4.0
10,000	7.38	192.4 ± 354.6	3,724.2 ± 2,993.7	7.7 ± 4.9
50,000*	8.42	1,046.7 ± 1,925.8	19,229.3 ± 15,290.7	9.0 ± 5.9
Random networks				
1,000	8.26	40.3 ± 60.7	380.0 ± 298.4	8.2 ± 5.0
10,000	10.67	402.5 ± 625.4	3,783.6 ± 3,035.0	10.4 ± 6.0
50,000	12.52	2,081.9 ± 3,167.0	-	-

Table 10: Average number of calls and standard deviation for synthetic graphs with 100 % more edges than nodes.

non-tree instances in GRIPP. In addition during the search in GRIPP we can prune hop nodes. We do not use hop nodes if the hop node has no successor nodes in $O(G)$ or if the hop node is successor of a used hop node in $O(G)$ and the path lengths between the query node and node in the reachable instance set of the hop node will not decrease. Therefore querying GRIPP requires fewer calls than querying the graph directly.

The set based approach requires the fewest number of calls. This is clear, as we only perform one SQL query for every distance. But the database system must compute more for every single call. Therefore not only the number of calls is important but also the time required to get the distance. Clearly, GRIPP could also be searched that way, but it is not yet implemented.

The table also shows that the number of calls for all three methods is higher for random graphs than for scale-free graphs. The reason is that the average distance is higher for random graphs than for scale-free graphs. A higher distance also means that more nodes must be queried during the search.

No. nodes	Avg. distance	GRIPP	breadth-first single	breadth-first set
Scale-free networks				
1,000	6.25	70.9 ± 110.9	166.3 ± 149.9	93.3 ± 94.2
10,000	7.38	957.1 ± 1,475.8	1,657.7 ± 1,475.1	4,320.0 ± 4,585.8
50,000*	8.42	12,010.1 ± 18,966.1	8,535.5 ± 7,692.7	114,993.9 ± 129,553.1
Random networks				
1,000	8.26	104.5 ± 140.8	173.6 ± 148.8	93.5 ± 77.5
10,000	10.67	2,043.9 ± 2,813.6	1,738.5 ± 1,517.5	3,920.7 ± 4,105.8
50,000	12.52	31,377.5 ± 42,587.0	-	-

Table 11: Average time in ms and standard deviation for synthetic graphs with 100 % more edges than nodes.

Table 11 shows the average query times for distance queries for 1,000 randomly selected node pairs. The figures show that for small, scale-free graphs , i.e., scale-free graphs with up to 10,000 nodes and 20,000 edges querying GRIPP is fastest. For larger graphs the standard breadth-first search is fastest.

The following observation helps to understand that behavior. In GRIPP the larger the graph becomes, the more nodes are reachable from the first node traversed during the creation of GRIPP. In GRIPP this also means that the length of the longest path from the root to a leave node increases. The target node in a reachable instance set of a large graph might therefore also be further away than in a small graph. During the search we first perform a reachability query on GRIPP to determine if a path exists and return the upper bound for the distance. With increasing size of the graph this upper bound also increases. During the breadth-first search we add all non-tree instances to the list of non-traversed nodes that have a path length to the query node that is shorter than the upper bound. As the upper bound for large graphs is high we add many non-tree instances to the list of not traversed non-tree instances that will never be considered as hop nodes as we find a shorter upper bound afterwards during the traversal. This explains the steep increase in time between 10,000 and 50,000 nodes.

The set based approach is only faster for graphs with 1,000 nodes, still in the same range as the other two approaches for 10,000 nodes, but much slower for 50,000 nodes. There are two reasons, namely (a) increasing average distance, and (b) entire execution of the last query. First, with increasing average distance the number of calls also increases. In every call we retrieve the child nodes for all nodes with distance i from the query node on the stack. For every child node the database system has to check if it is already on the stack or if it has to be added. For every call we use only one SQL statement with a division operation, i.e., select nodes, that are in the set of child nodes, but not already in the stack relation. As division operations are very costly in a RDBMS the distance query takes much more time with increasing path length and graph size.

The second reason is that we have to execute the last query entirely. Consider the case where the distance between two nodes is i. We look for child nodes of nodes with distance $i-1$ to the query node. In the standard breadth-first search we will consider the nodes one at a time. If we find the target node immediately we can terminate the search, i.e., in best case execute only one additional query. In contrast, in the set based approach we have to retrieve all child nodes and afterwards look for the target node. Therefore the set based approach clearly has disadvantages against the standard implementation of a breadth-first search.

7.3.2 Breadth-first search combined with GRIPP reachability

For the GRIPP distance search we first perform a reachability query to determine if a path between the query and the target node exists, i.e., we can answer distance queries where no path exists very fast. In contrast, using breadth-first search in worst case we have to traverse the entire graph to determine if a path exists. For example, for a scale-free graph with 10,000 nodes and 20,000 edges for almost 40 % of the randomly selected node pairs $reach(v, w) = \texttt{false}$. The standard breadth-first search (breadth-first single) requires on average 1,700 ms to return $dist(v, w) = \texttt{null}$. We can split those node pairs in two groups, one group where the query node has no outgoing edges ($\tilde{4}0$ % of the node pairs), i.e., no recursive queries are necessary, and one group where the query node has outgoing edges ($\tilde{6}0$ %). For the group with no outgoing edges queries require on average 1.3 ms to return an answer, while for the group with outgoing edges a query reqires on average

2,846 ms. Using GRIPP we can reduce that to 20 ms on average.

No. nodes	$reach(v, w)$	GRIPP	breadth-first single	breadth-first set
Scale-free networks				
1,000	yes	70.9 ± 110.9	113.5 ± 107.7	69.4 ± 98.7
	no		166.3 ± 149.9	93.3 ± 94.2
10,000	yes	957.1 ± 1,475.8	1,136.4 ± 1,123.6	2,374.4 ± 3,160.8
	no		1,657.7 ± 1,475.1	4,320.0 ± 4,585.8
50,000	yes	12,010.1 ± 18,966.1	5,797.8 ± 5,670.1	58,665.0 ± 86,981.6
	no		8,535.5 ± 7,692.7	114,993.9 ± 129,553.1
Random networks				
1,000	yes	104.5 ± 140.8	124.4 ± 114.8	72.4 ± 57.1
	no		173.6 ± 148.8	93.5 ± 77.5
10,000	yes	2,043.9 ± 2,813.6	1,214.9 ± 1,178.8	2,261.9 ± 2,864.4
	no		1,738.5 ± 1,517.5	3,920.7 ± 4,105.8
50,000	yes	31,377.5 ± 42,587.0	6,288.9 ± 5,896.7	54,998.2 ± 75.671.1
	no		-	-

Table 12: Comparison between breadth-first search with and without precomputing $reach(v, w)$. Average time in ms and standard deviation for synthetic graphs with 100 % more edges than nodes.

Table 12 shows the average query time for $dist(v, w)$ with and without applying $reach(v, w)$ over GRIPP first. The figures show that querying GRIPP for reachability first reduces the average query times for both methods of the breadth-first search.

8 Related Work

To efficiently answer reachability and distance queries, pre-computation of the transitive closure TC of a graph is a natural choice [27]. Efficient algorithms for computing the TC in relational databases have been developed [2], but the size of the TC is $O(|V|^2)$, making it inapplicable to large graphs.

To reduce storage space, Cohen and colleagues [7] developed the 2-Hop-Cover that requires in worst-case $O(|V| * |E|^{1/2})$ space and can answer reachability queries with only two lookups. However, computing the optimal 2-Hop-Cover is NP-hard and requires the TC to be computed first [7]. Schenkel et al. [23] proposed graph partitioning as a method to get away from the necessary pre-computation of the entire TC, thus reducing storage requirements during the index creation process. This approach works very well for forests with few connections between the different sub-trees. But for dense graphs, such as the metabolic network of KEGG, the partitioning is not very effective. Without partitioning the 2-Hop-Cover is about 5,600 times smaller than the transitive closure, while with partitioning this factor shrinks to about 500. Schenkel et al. also showed that the 2-Hop-Cover can be extended to answer distance queries. This comes with the tradeoff that the size of the 2-Hop-Cover is much larger. Using partitioning the 2-Hop-Cover for KEGG is only two times smaller than the transitive closure itself (R. Schenkel, personal communication, May 2006). Even without partitioning the cover is just 29.4 times smaller than the transitive closure – compared to 5,600 times for reachability. Clearly, these compression factors make the 2-Hop-Cover not applicable for large graphs to answer distance queries.

To index trees and DAGs a wealth of different numbering schemes have been proposed in the literature, especially to support XPath queries. Examples include pre- and postorder values [12], range-based labeling [5, 28], and Dewey numbers [22]. All these schemes only work on trees. Approaches that use numbering schemes on DAGs have been proposed. In previous work, we described an ’unfolding’ technique, where each node in a subtree with more than one parent node receives multiple pre- and postorder values [24]. Since this leads to a combinatorial explosion in the number of value pairs, it is only feasible for tree-like DAGs. Instead of labeling successor nodes multiple times, Agrawal et al. [1] proposed to propagate the intervals of child nodes ’upwards’. The graphs they used contained no more than 1,000 nodes. Chen et al. [6] presented a hybrid index structure for DAGs, using a region encoding for a spanning tree and an additional data structure for storing non-tree edges which is traversed recursively at query time. They applied their approach to DAGs with 200,000 nodes and 1.8 times more edges. It is not clear how their approach would perform on larger, cyclic, multi-rooted graphs. In none of these publications the problem of answering distance queries was discussed.

He, Wang, and colleges [14, 26] proposed two indexing strategies to answer reachability queries on graphs. For both approaches they first identify strongly connected components and collapses these to one node, therefore reducing the size of the graph. The remaining structure is a DAG. The first approach uses a combination of numbering schemes and 2-hop cover, while the second is merely based on a numbering scheme to encode the DAG. For experiments they used random graphs with 2,000 nodes and up to 4,000 edges. It is not clear, if their approach can be used to efficiently index dense graphs in the size of one million or more nodes. In addition both approaches will not support distance queries.

To answer distance queries on graphs Dijkstra’s algorithm and the A* algorithm are used [8]. Dijkstra’s algorithm works well on graphs with weighted edges. For graphs with unweighted edges – as is the case for biological networks – Dijkstra’s algorithm is basically a breadth-first search. The A* algorithm is an extension for Dijkstra’s algorithm and requires in addition to weighted edges also some information about the ’best’ edge to choose next. Therefore, both algorithms are not well suited to answer distance queries on graphs with unweighted edges.

9 Discussion and Conclusion

We presented the GRIPP index structure supporting reachability and distance queries on directed graphs. Since creating GRIPP requires only linear time and space, it can be used to index graphs with millions of nodes. And as the algorithms for indexing and querying GRIPP are implemented as stored procedures in a RDBMS GRIPP can be easily be integrated to index and query graphs in graph based applications.

With GRIPP, reachability queries on many types of graphs can be answered in almost constant time using an almost constant number of queries. For reachability queries we believe that GRIPP can be further improved using the idea of collapsing strongly connected components (SCC) into single nodes. SCC can be computed in linear time [8]. The effect of this optimization would strongly depend on the properties of the graph, i.e., the number and size of the SCCs, and would be the strongest for very dense graphs. However, given the current query times which are less than 5 ms even for very large graphs, this is not our primary next goal.

Distance queries in GRIPP require an order of magnitude less calls than recursive query strategies, but the time required is comparable or slower than recursive query strategies. But even for recursive strategies to answer distance queries GRIPP is important, as we can answer reachability first, i.e., reducing the time for distance queries where no path exists.

In the future, we plan to use GRIPP as an index structure for the pathway query language (PQL) [20]. PQL provides syntax to pose graph queries. We are interested in answering such queries efficiently, i.e., we plan to provide a cost based optimization for such queries. GRIPP is currently the most scaleable indexing method we are aware of. In addition the execution of reachability queries is very fast. For distance queries we have to further evaluate the conditions where GRIPP has advantages over recursive strategies. To cover the capabilities of PQL we plan to implement path length and path queries as well.

Acknowledgment. This work is supported by BMBF grant no. 0312705B (Berlin Center for Genome-Based Bioinformatics). Many thanks to Johannes Vogt who wrote the software to visualize the GRIPP index structure and the execution of queries on GRIPP.

A GRIPP_breadth – a different index structure

The index structure GRIPP_breadth is basically the same as GRIPP. In GRIPP_breadth we also assign every node in the graph G at least one pre- and postorder value. The difference is that for GRIPP_breadth we first perform a breadth-first search starting at the root node. During the search we store the distance between the root node and every node in the graph. In the next step we create $IND(G)$ during a depth-first traversal of G using the information from the breadth-first search. During this depth-first traversal we assign the pre- and postorder values and the depth information to a node.

For GRIPP we add a tree instance of node v to $IND(G)$ if we encounter v for the first time during the depth-first traversal. Every other time we reach v, i.e. $IND(G)$ already contains a tree instance of v, we add a non-tree instance of v to $IND(G)$. In contrast, in GRIPP_breadth we only add a tree instance for v to $IND(G)$ if (a) v has no tree instance in $IND(G)$ and (b) the depth of the instance of v in $O(G)$ equals the distance of v to the root node found during the breadth-first traversal. Every other time we add a non-tree instance of v to $IND(G)$.

A.1 Properties of this index structure

A.1.1 Time and Space Requirements

The space requirements to store the GRIPP_breadth index table are identical to the space requirements for GRIPP. Only during the index creation we temporarily have to store the information generated by the breadth-first search.

The time requirements for GRIPP_breadth are higher than for GRIPP, because (a) we first perform a breadth-first search and (b) during the traversal we have to evaluate if the depth in $O(G)$ of a node v is equal to the distance of v to the root node.

The index creation for large graphs is much slower than expected. This is due to the breadth-first search. During that search we only add nodes to the list that have not already been traversed. As this step requires a division operation, which is very costly in a RDBMS, the time increases dramatically with increasing number of nodes.

A.1.2 Properties of Nodes in O(G)

Node have exactly one tree instance For every node v in G there exists exactly one tree instance in $O(G)$. Proof omitted.

Preorder of tree instance In the GRIPP index structure the tree instance of a node v has a lower preorder rank than all non-tree instances of that node as we add a tree instance to $IND(G)$ the first time we see that node. This property does not hold for GRIPP_breadth. In GRIPP_breadth when we reach a node for the first time we will not generally add a tree instance to $IND(G)$. Instead we check if the depth of the instance in $IND(G)$ is equal to the distance to the root node. If the depth is higher than the distance to the root node we add a non-tree instance of v to $IND(G)$. We will add a tree instance of v at a later stage of the traversal. This also means that non-tree instances can have higher or lower preorder ranks than the tree instance of a node.

Shortest paths Given two nodes v and w in G and the tree instances v^T of v and w^T of w in $O(G)$ created by Algorithm 7. If v^T is ancestor to w^T in $O(G)$ we can immediately determine the distance between v and w in G by calculating $w^T_{depth} - v^T_{depth}$.

The reason for this is as follows. In GRIPP_breadth as in GRIPP $O(G)$ contains tree as well as non-tree instances. In $O(G)$ created by GRIPP_breadth the length of every path from the tree instance r^T of the root node r to a tree instance v^T of node v equals the distance of r to v in G. Remember, we only create a tree instance for v if the depth of v^T, i.e. the distance to r^T in $O(G)$ equals the distance of v to r in G. Every non-tree instance of v^N has the same or a greater distance to r^T in $O(G)$.

Knowing this we can also deduce the distance between two nodes v and w in G if v^T is ancestor to w^T. The distance then is $dist(v, w) = w^T_{depth} - v^T_{depth}$. But note, we can not ease the condition that the instance of w can also be a non-tree instance.

If v^T is no ancestor of w^T in $O(G)$ we can not immediately determine the distance between v and w in G. We have to execute a more complicated search as shown in Section 4.3.

A.2 Comparison GRIPP and GRIPP_breadth

In the order tree created by GRIPP the first child node c of the root node contains tree instances for all nodes v that are reachable from c in G. The higher connected the graph is, i.e. the more edges this graph contains, the more tree instances are successors of the tree instance of c in $O(G)$. The remaining child nodes to the root then contain only few tree instances and some non-tree instance. The general appearance of GRIPP is narrow, but deep.

In contrast, in the order tree created by GRIPP_breadth is broad and shallow. The differences can be seen in Figures 19 and 20 for a identical scale-free graph of 100 nodes and 200 edges.

A.2.1 Advantages of GRIPP_breadth

The advantage of GRIPP_breadth lies in the fact that every path between tree instances in $O(G)$ is shortest, i.e., we can immediately determine the distance between nodes if one tree instance is ancestor of the other tree instance in $O(G)$. This also means that during the execution of distance queries we can prune more often. But experiments show (data not shown) that the average time to execute distance queries for a pair of nodes only decreases by about 10 % compared to the execution time for GRIPP.

A.2.2 Disadvantages of GRIPP_breadth

GRIPP_breadth has several disadvantages, namely increased creation time compared to GRIPP and increased average query time for reachability queries. The increased creation time stems mainly from the breadth-first traversal as discussed earlier.

To understand the reason for the increased execution time have a look at Figures 19 and 20. In GRIPP the first traversed node during the index creation, which is also a stop node, has many reachable instances in the order tree. When querying for $reach(v, w)$ with

v in that set it is very likely that we find (a) an instance of w or (b) an instance of the stop node in $RIS(v)$, i.e., we can terminate the search very fast. In GRIPP_breadth this is not the case. Many nodes have some reachable instances in the order tree. This means, during a reachability search we might have to use many nodes as hop nodes. But this also means that on average reachability queries require more time on GIRPP_breadth than on GRIPP. Experiments show that the average time increases by over 100 % (data not shown).

As distance queries are not considerably faster on GRIPP_breadth and the index creation as well as reachability queries are much slower we will not investigate further in GRIPP_breadth.

A.3 Algorithm for GRIPP_breadth

Algorithm 7 shows the procedures and functions to compute the GRIPP_breadth index structure. We first compute the table `BREADTH_INFO` by applying a breadth-first search over the graph. We use the information in `BREADTH_INFO` during the depth-first traversal to compute the GRIPP_breadth index structure.

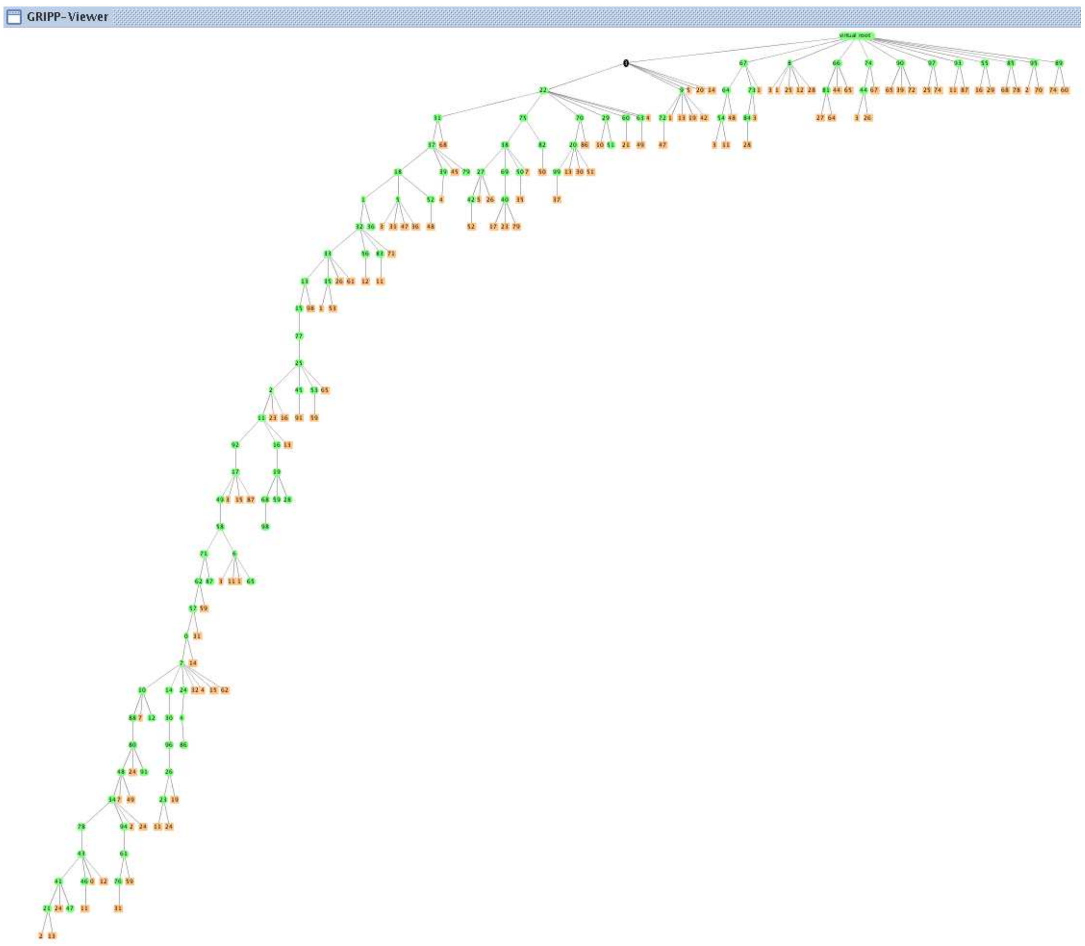


Figure 19: Order tree created by GRIPP for a graph of 100 nodes and 200 edges.

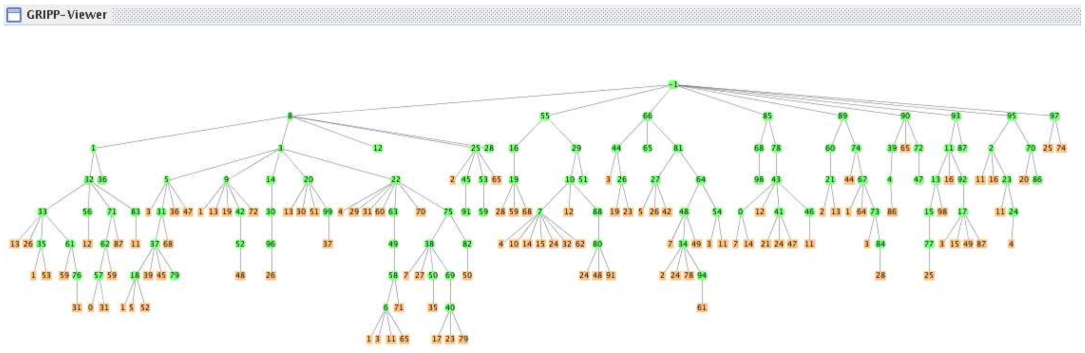


Figure 20: Order tree created by GRIPP_breadth for a graph of 100 nodes and 200 edges.

Algorithm 7: The GRIPP algorithm to compute $IND(G)$ according to GRIPP_breadth

```
pre_post ← 0
PROCEDURE compute_GRIPP(root_node)
    BREADTH_INFO ← breadth_first(root_node)
    pre_node ← pre_post
    pre_post ← pre_post + 1
    traverse(root_node, 0, BREADTH_INFO)
    GRIPP ← GRIPP ∧ (root_node, pre_node, pre_post, 0, T)
end

FUNCTION breadth_first(root_node)
    BREADTH_INFO ← (root_node, 0)
    push(node_stack, (root_node, 0))
    repeat
        (next_node, node_dist) ← pop(node_stack)
        forall child ∈ children(next_node) do
            if child ∉ BREADTH_INFO then
                push(node_stack, (child, node_dist +1))
                BREADTH_INFO ← BREADTH_INFO ∧ (child, node_dist +1)
            end
        end
    until node_stack = ∅
    return BREADTH_INFO
end

PROCEDURE traverse(next_node, cur_dist, BREADTH_INFO)
    seen ← seen ∪ next_node
    while child ← next(children(next_node)) do
        pre_node ← pre_post
        pre_post ← pre_post + 1
        if child ∉ seen AND cur_dist +1=getDepth(child, BREADTH_INFO) then
            node_inst ← T
            traverse(child, cur_dist +1)
        else
            node_inst ← N
        end
    end
    GRIPP ← GRIPP ∧ (child, pre_node, pre_post, cur_dist, node_inst)
end
```

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 253–262, 1989. ACM.

[2] R. Agrawal and H. V. Jagadish. Direct algorithms for computing the transitive closure of database relations. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB)*, pages 255–266, 1987. Morgan Kaufmann.

[3] A.-L. Barabąsi and Z. N. Oltvai. Network biology: understanding the cell's functional organization. *Nature Reviews Genetics*, 5(2):101–113, 2004.

[4] I. Borodina and J. Nielsen. From genomes to in silico cells via metabolic networks. *Current Opinion in Biotechnology*, 16(3):350–355, 2005.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 310–321, 2002. ACM.

[6] L. Chen, A. Gupta, and M. E. Kurul. Stack-based Algorithms for Pattern Matching on DAGs. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 493–504, 2005. ACM.

[7] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

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

[9] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th annual ACM Symposium on Theory of computing (STOC)*, pages 365–372, 1987. ACM.

[10] M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, 1997.

[11] T. Grust. Accelerating XPath location steps. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 109–120, 2002. ACM.

[12] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.*, 29:91–131, 2004.

[13] R. H. Güting. GraphDB: Modeling and Querying Graphs in Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 297–308, 1994. Morgan Kaufmann.

[14] H. He, H. Wang, J. Yang, and P. S. Yu. Compact reachability labeling for graph-structured data. In *Proceedings of the 2005 ACM International Conference on Information and Knowledge Management (CIKM)*, pages 594–601, 2005. ACM.

[15] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A. L. Barabsi. The large-scale organization of metabolic networks. *Nature*, 407(6804):651–654, 2000.

[16] G. Joshi-Tope, M. Gillespie, I. Vastrik, P. D'Eustachio, E. Schmidt, B. de Bono, B. Jassal, G. R. Gopinath, G. R. Wu, L. Matthews, S. Lewis, E. Birney, and L. Stein. Reactome: a knowledgebase of biological pathways. *Nucleic Acids Research*, 33(Database issue):D428–32, 2005.

[17] M. Kanehisa, S. Goto, S. Kavashima, Y. Okuno, and M. Hattori. The KEGG resource for deciphering the genome. *Nucleic Acids Research*, 32(Database issue):D277–D280, 2004. Database issue.

[18] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A declarative query language for RDF, 2002. In The 11th Intl. World Wide Web Conference (WWW2002).

[19] C. Lemer, E. Antezana, F. Couche, F. Fays, X. Santolaria, R. Janky, Y. Deville, J. Richelle, and S. J. Wodak. The aMAZE LightBench: a web interface to a relational database of cellular processes. *Nucleic Acids Research*, 32(Database issue):D443–448, Jan 2004.

[20] U. Leser. A query language for biological networks. *Bioinformatics*, 21(2):ii33-ii39, Sep 2005.

[21] H. Lu. New Strategies for Computing the Transitive Closure of a Database Relation. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB)*, pages 267–274, 1987. Morgan Kaufmann.

[22] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of xml twig pattern matching. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 193–204, 2005. ACM.

[23] R. Schenkel, A. Theobald, and G. Weikum. Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 360–371, 2005. IEEE Computer Society.

[24] S. Trißl and U. Leser. Querying Ontologies in Relational Database Systems. In *Proceedings of the Second International Workshop on Data Integration in the Life Sciences (DILS)*, volume 3615 of *Lecture Notes in Computer Science*, pages 63–79, 2005. Springer.

[25] C. von Mering, L. J. Jensen, B. Snel, S. D. Hooper, M. Krupp, M. Foglierini, N. Jouffre, M. A. Huynen, and P. Bork. STRING: known and predicted protein-protein associations, integrated and transferred across organisms. *Nucleic Acids Research*, 33(Database issue):D433–D437, Jan 2005.

[26] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, page 75, 2006. IEEE Computer Society.

[27] H. S. Warren. A modification of Warshall's algorithm for the transitive closure of binary relations. *Commun. ACM*, 18(4):218–220, 1975.

[28] F. Weigel, K. U. Schulz, and H. Meuss. The bird numbering scheme for XML and tree databases - deciding and reconstructing tree relations using efficient arithmetic operations. In *Proceedings of the Third International XML Database Symposium (XSym)*, volume 3671 of *Lecture Notes in Computer Science*, pages 49–67, 2005. Springer.