Fast and Practical Indexing and Querying of Very Large Graphs

Silke Trißl
Humboldt-Universität zu Berlin
Unter den Linden 6
D-10099 Berlin, Germany
trissl@informatik.hu-berlin.de

Ulf Leser
Humboldt-Universität zu Berlin
Unter den Linden 6
D-10099 Berlin, Germany
leser@informatik.hu-berlin.de

ABSTRACT

Many applications work with 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 queries in graphs.

GRIPP requires only linear time and space. Using GRIPP, we can answer reachability queries on graphs with 5 million nodes on average in less than 5 milliseconds, which is unrivaled by previous methods. We evaluate the performance and scalability of our approach on real and synthetic random and scale-free graphs and compare our approach to existing indexing schemes. GRIPP is implemented as stored procedure inside a relational database management system and can therefore very easily be integrated into existing graph-oriented applications.

Categories and Subject Descriptors

H.2.8 [**Database management**]: Database Applications—*graph indexing and querying*

General Terms

Performance.

Keywords

Graph indexing, Reachability queries, Databases

1. INTRODUCTION

Managing, analyzing, and querying graph-like data is important in many areas such as geographic information systems [14], web site analysis [12], and querying XML documents with XPointers [22]. In addition, the semantic web builds on RDF, a graph-based data model, and on graph-based query languages such as RQL [19] or SparQL[1]. Thus, querying graphs will likely become even more important in the near future. In our own research we mostly work with data from the Life Science domain. The importance of graphs in this area is also increasing rapidly. 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 [5]. Nodes in these networks are molecules, reactions, or physical interactions. These nodes may be annotated with a vast amount of additional data stored in various databases. Edges represent interactions, such as the enzymatic conversion of molecules, the regulation of gene expressions, or the physical interaction of proteins. Large networks, e.g., metabolic [18] or protein-protein interaction networks [23], are built from single interactions. In [16] van Helden and colleagues identified several important queries on biological networks. For instance, the question "find all genes whose expressions is directly or indirectly influenced by a given molecule" can be mapped to a reachability query in a directed graph of genes and regulation events.

The size of the graphs or networks under consideration can be very large. Typical biological 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 [4], which are believed to contain more than 300,000 different proteins. Already today, networks of biomedical entities (genes, diseases, drugs, etc.) extracted from publication databases contain more than 10 million edges[2].

One important type of queries in graphs are *reachability queries*. Given two nodes v and w in a graph, we want to verify whether there exists a path from v to w. There are two obvious approaches to answer such queries. First, one can recursively traverse the graph at query time, starting from v and performing a depth-first or breadth-first search until w is reached or no more edges remain [9]. Given a graph G with n nodes and m edges this method requires $O(m)$ lookups. No index is needed, but performance is bad even on small graphs. Second, one can pre-compute the transitive closure (TC) of the graph. Using the TC as index reachability queries can be answered by a single lookup. But on the downside, the computation of the TC is $O(n^3)$ and its size $O(n^2)$ [9]. This renders its computation and storage infeasible for large graphs (see also Section 7).

Table 1 shows the worst case complexity of several approaches to reduce computation cost or storage space (see

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

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

Section 2.1 for details). Chen et al. [6] (Labeling+SSPI) only index a spanning tree and store additional edges in a separate index structure, called SSPI. The entire index requires $O(n+m)$ space, but parts must be traversed recursively at query time. The Dual Labeling approach by Wang et al. [26] can be queried in constant time. They also first compute a spanning tree and build a condensed transitive closure over the remaining t edges. Index generation requires $O(n+m+t^3)$ time and its size is $O(n+t^2)$. This is acceptable for sparse, tree-like graphs (with $t \ll n$), but for denser graphs ($t > 2n$) the method also requires an prohibitively large amount of space. Schenkel et al. [22] proposed HOPI, a method to compute the 2-Hop-Cover, which requires only $O(nm^{1/2})$ space, but as the TC $O(n^3)$ time to compute the index.

Table 1: Worst case complexities of different index and query strategies to answer reachability queries.

	Query time	Index time	Index size
Recursive	$O(n+m)$	-	-
Labeling+SSPI	$O(m-n)$	$O(n+m)$	$O(n+m)$
GRIPP	$O(m-n)$	$O(n+m)$	$O(n+m)$
Dual Labeling	$O(1)$	$O(n+m+t^3)$	$O(n+t^2)$
HOPI	$O(m^{1/2})$	$O(n^3)$	$O(nm^{1/2})$
TC	$O(1)$	$O(n^3)$	$O(n^2)$

In this paper we present the GRIPP index (GRaph Indexing based on Pre- and Postorder numbering) for indexing very large graphs. Its basic idea is an adaptation of the pre- and postorder numbering scheme – so far only applied to trees [10] and directed, acyclic graphs (DAGs) [1, 24] – to (cyclic, possibly unrooted) graphs. The GRIPP index can be computed in $O(n+m)$ time and requires only $O(n+m)$ space. Therefore, GRIPP can be used to index graphs far beyond the scope of the TC or Dual Labeling. Answering reachability queries with GRIPP requires in worst case $O(m-n)$ time (see Table 1), which is the same as for Labeling+SSPI. However, we will show that with GRIPP the actual time to answer a reachability query is almost constant over different sizes, shapes, and densities of graphs. We will support this claim both experimentally and analytically. GRIPP indexes graphs containing 50,000 nodes and 100,000 edges in ~ 120 sec and answers reachability queries on such graphs in ~ 3.5 ms using ~ 2 queries. Even for the largest graphs tested, consisting of 5 million nodes and 10 million edges, the query time increases only marginally.

GRIPP is designed as a persistent index stored in a relational database management system (RDBMS). All operations for indexing and querying are implemented as stored procedures, thus fully leveraging the main memory management capabilities of an RDBMS. Therefore, GRIPP has no particular requirements regarding the size of available main memory. Integrating our method into an existing, RDBMS-based application only requires the installation of stored procedures. After the index is created using a simple SQL function, applications use another SQL function to answer reachability queries. We therefore believe that GRIPP is a highly practical, non-intrusive method.

Our paper is organized as follows. In the next section we describe our model and discuss related work. In Section 3 we present the GRIPP index structure itself. In Section 4 we show how to evaluate reachability queries using GRIPP and propose pruning strategies. The effectiveness of GRIPP depends on the order in which the graph is traversed during index creation, which is discussed in Section 5. In Section 6 we describe several heuristics for an efficient implementation of GRIPP. 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 5,000,000 nodes and different graph densities. Section 8 concludes the paper.

2. BACKGROUND AND RELATED WORK

We adopt notation from Cormen et al. [9]. 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 graph has n nodes and m edges, the *size* of a graph is $|G| = n + m$. The *degree* of a node is the number of incoming and outgoing edges of the node. The *density* of a graph is the ratio between n and m. Given a graph G, a *path p* is a sequence of nodes that are connected by directed edges.

We assume that graphs are stored as a collection of nodes and edges in an RDBMS. The information on nodes includes a unique identifier. Edges are stored as binary relationship between two nodes, i.e., as adjacency list.

We analyze the problem of answering reachability queries on graphs. Let $G = (V, E)$ be a graph and let $v, w \in V$ be two nodes of G. w is *reachable* from v, iff there exists a path from v to w. Given two nodes v and w, the function *reach*(v, w) returns `true` if w is reachable from v, and `false` otherwise.

Two nodes $v, w \in G$ are in the same *strongly connected component* if *reach*(v, w) = *reach*(w, v) = *true*, otherwise not. Collapsing every strongly connected component into a representative node results in the *component graph*, which forms a directed acyclic graph (DAG).

2.1 Related Work

The simplest way to answer reachability questions on graphs is to traverse the graph at query time using depth- or breadth-first search [9]. SQL:2003 provides standard syntax to express recursive queries and some database management systems have implemented that standard. But in most RDBMS recursive queries cannot be expressed by SQL queries, but must be implemented using stored procedures (see Section 7 for their performance).

Another option is to pre-compute the transitive closure (TC). The TC of a graph is the set of node pairs (v, w) for which a path from v to w exists. Efficient algorithms for computing the TC in relational databases have been developed [2, 21]. But the size of the TC is $O(n^2)$ and its computation time $O(n^3)$, which makes it inapplicable to large graphs. For instance, computing the transitive closure with the method described in [21] on a graph of 50,000 nodes and 100,000 edges did not finish within 24 hours (see Section 7 for details).

To reduce storage space, Cohen and colleagues [8] developed the 2-Hop-Cover, which requires $O(nm^{1/2})$ space and can answer reachability queries with only two lookups. However, the problem of computing the optimal 2-Hop-Cover is NP-hard and requires the TC to be computed first [8]. Schenkel et al. [22] proposed graph partitioning as a method to get away from the necessary pre-computation of the entire TC, thus reducing storage requirements for the index creation process. This approach, called HOPI, works very well for forests with few connections between the different sub-trees. But for denser graphs, such as the metabolic network of KEGG, the partitioning is not effective as the size of the 2-Hop-Cover is only two times smaller than the transitive closure itself (R. Schenkel, personal communication, May 2006). Cheng et al. [7] proposed a complex, geometry based method that does not require the computation of the TC to compute the 2-Hop-Cover. In their approach they first identify strongly connected components and then label each component. Based on these labels they generate a reachability map that is used to compute the 2-Hop-Cover. In contrast to Cohen et al. they use an approximation for determining the densest subgraph, which is required during creation of the 2-Hop-Cover. This approximation reduces the computation time, but might increase the index size. But for their tested graphs the storage space is only slightly larger than compared to other approaches. However, their approach for computing the index is main memory based, which limits its scalability towards very large graphs.

2.1.1 Interval-Based Approaches

A different indexing strategy is to label nodes using the pre- and postorder numbering scheme. This indexing scheme was originally described for tree structured data [10]. In the pre- and postorder numbering scheme each node in the tree receives a pre- and postorder value. Both values are assigned 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 as soon as node v is encountered during the traversal. The postorder value v_{post} is assigned after all successor nodes of v have been traversed.

A table of all nodes with their assigned pre- and postorder values forms an index with which reachability queries can be answered with a single query. If w is reachable from v, w must have a higher preorder and lower postorder value than v, i.e., $w_{pre} > v_{pre} \land w_{post} < v_{post}$. However, the evaluation of this condition in an RDBMS is prohibitively slow due to the two non-equijoins [13]. An obvious optimization is to use only one counter for the pre- and postorder values. Therefore, all successor nodes w of v must lie within the borders given by the pre- and postorder values of v, i.e., $[v_{pre}, v_{post}]$. Thus, $reach(v, w) \Leftrightarrow v_{pre} < w_{pre} < v_{post}$.

Still, this method 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. To extend this strategy to directed, acyclic graphs (DAGs) we used an 'unfolding' technique [24], where each added 'non-tree' edge in the DAG introduces a new entry in the index structure. The target node of the additional edge as well as all its successors get additional pre- and postorder values incurring an exponential explosion in the index size as DAGs become very 'tree-unlike'. Our newly proposed index structure GRIPP also traverses nodes multiple times, but does not visit children of an already visited node again, which makes its space requirements only linear in the size of the graph (see also Section 3).

Agrawal et al. [1] described a different method to index DAGs. They propagate pre- and postorder values upwards. The source of an additional edge as well as all its ancestors receive the pre- and postorder value of the target as another pre- and postorder value pair. In contrast, in GRIPP only the target will get an additional pre- and postorder value. For GRIPP this comes at the cost that at query time we have to traverse the index recursively as explained in Section 4, while for the approach of Agrawal et al. the query time is linear in the number of intervals assigned to a node. To reduce the number of intervals of a node and therefore storage space Agrawal et al. merge pre- and postorder ranges of nodes. They present an algorithm to compute an optimal index structure, i.e., an index structure with least storage space. This algorithm determines an optimal order for the traversal of nodes during labeling. The authors state that computing the optimal index structure has the same time complexity as the computation of the transitive closure, which also makes it inapplicable to large DAGs. However, it would be worth studying whether the heuristics described for GRIPP in Sections 5 and 6 would also be applicable here.

2.1.2 Hybrid Approaches

Chen et al. [6] presented a hybrid index structure for DAGs, called Label+SSPI. This approach uses pre- and postorder labeling for a spanning tree and an additional data structure, called SSPI, for storing non-tree edges. This results in an index structure in the size of $O(n + m)$. For answering $reach(v, w)$ the spanning tree part is handled by an initial range query. If w is not found in the range of v the additional data structure is traversed recursively, which leads to $(m - n)$ queries in worst case (see Table 1).

He et al. [15] proposed a different indexing strategy, called HLSS, that first identifies strongly connected components and collapses these to one node to reduce the size of the graph. The remaining structure is a DAG. They label the nodes of a spanning tree with pre- and postorder values. To encode the reachability relationship over non-tree edges they compute the 2-Hop-Cover over these edges. The query time is not constant, but depends on the size of the 2-Hop-Cover label of a node.

Wang et al. [26] proposed an index structure, called Dual Labeling that allows to answer reachability queries in constant time. They also identify strongly connected components and collapse these to one node. They label the nodes of a spanning tree with pre- and postorder values. Instead of computing the 2-Hop-Cover they compute the transitive closure over the remaining edges (called TLC matrix). Using pre- and postorder values of nodes the TLC matrix can be further reduced in size. The authors state that in sparse, tree-like graphs the number of non-tree edges is small. Therefore the size of the TLC matrix is much smaller than the TC of the graph itself.

In Section 7 we will compare the approaches from Chen et al. and Wang et al. with our index structure GRIPP.

3. GRIPP INDEX STRUCTURE

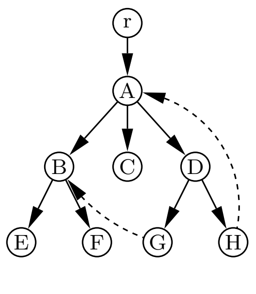
GRIPP extends the pre- and postorder labeling scheme to work on graphs. Every node in the graph receives at least one pair of pre- and postorder values. As nodes can have multiple parents one pair is not sufficient to encode the entire graph structure. Therefore, some nodes will get more than one pair of values.

For now, we assume that the graph has exactly one root node, i.e., one node without incoming edges. We also assume an arbitrary, yet fixed order among child nodes, e.g., given by the ID of the node. In Section 6 we explain how to deal with graphs with multiple or no 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. We always traverse child nodes according to their order. A node v with $n > 1$ incoming edges is reached n times 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 to v and proceed the depth-first traversal. After all child nodes have a value pair, v receives its postorder value. Of course, we reach v $n-1$ times again. Assume we reach v over edge e_j, $e_j \neq e_i$. We call e_j a *non-tree edge* and assign a pre- and postorder value to v, but do not traverse child nodes of v. We store the pre- and postorder values together with the node identifier as *node instances* in an *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 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.*

Figure 1(a) shows a graph and Figure 1(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 in G.



(a) Graph, G.

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

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

Figure 1: Graph G and its GRIPP index table $IND(G)$. Solid lines 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 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 and all edges of G.*

Intuitively, $O(G)$ consists of a spanning tree $T(G)$ of the graph and a non-tree part $N(G)$. $T(G)$ contains the tree instance of every node in the graph and is connected by tree edges E^T. $N(G)$ contains a node for every non-tree instance in $IND(G)$, which is connected by a non-tree edge to a node in the spanning tree $T(G)$. Therefore, every non-tree instance is a leaf node, while tree instances can be inner or leaf nodes. Note that the shape of $O(G)$ depends on the order with which G is traversed. In Section 6 we shall explain how we can determine an order that is well suited for our purpose. In Figure 2 the instances of $IND(G)$ shown in Figure 1(b) are plotted. Nodes A and B occur twice in $O(G)$ as they have two instances in $IND(G)$.

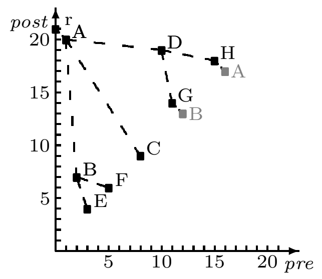


Figure 2: Pre-/ postorder plane for $IND(G)$ in Figure 1(b). Dotted lines indicate $O(G)$. Non-tree instances are displayed in gray.

The space requirement to store the GRIPP index table is $O(n+m)$, i.e., linear in the size of the graph. More precisely $IND(G)$ has as many entries as G has edges plus one entry for every root node (see also Section 6.2). To create the GRIPP index structure we perform a depth-first traversal, requiring $O(n + m)$ time.

4. QUERYING GRIPP

In the following chapter we show how to use the GRIPP index to efficiently answer reachability queries for a fixed pair of nodes. Recall that reachability queries in trees 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}. When we try to query 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, using the *hop technique*.

4.1 Hop technique

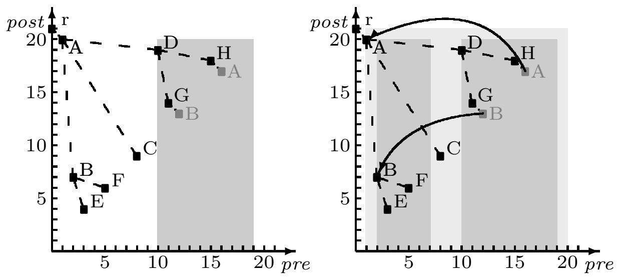
To evaluate *reach*(v, w) we use the index table $IND(G)$. Observe that v can have many instances in $IND(G)$. Every non-tree instance of v in $IND(G)$ is a leaf node in $O(G)$ and therefore has no successors in $O(G)$. Let v' be the tree instance of v. If v' is an inner node in $O(G)$ it has reachable nodes w' in $O(G)$ such that $v'_{pre} < w'_{pre} < v'_{post}$. Those can be retrieved with a single query. We call this set of instances *reachable instance set* of v. In Figure 3(a) the reachable instance set of node D is shown. It contains instances of nodes G, B, H, and A.

Definition 3 (Reachable instance set). *Let $v \in V$ be a node of graph G and $v' \in IND(G)$ its tree instance. The* reachable instance set *of v, written $RIS(v)$, is the set of all instances that are reachable from v' in $O(G)$, i.e., that have a preorder value in $[v'_{pre}, v'_{post}]$.*

To answer *reach*(v, w) we proceed as follows. We first find the tree instance v' of v and retrieve its reachable instance set. If $w \in RIS(v)$, we finish and return *true*, otherwise we have to extend the search. 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 that, we have to examine all non-tree instances of nodes in $RIS(v)$. We call those nodes *hop nodes*. In Figure 3(a) $RIS(D)$ contains non-tree instances of nodes B and A, i.e., both are hop nodes for D.

DEFINITION 4 (HOP NODE). *Let $v, h \in V$ and h' be a non-tree instance of h. If $h' \in RIS(v)$ then h is called a* hop node *for v.*



(a) $RIS(D)$ (b) $RIS(D)$ and $RIS(B)$ in dark gray; $RIS(A)$ in light gray.

Figure 3: The example shows $reach(D, r)$ evaluated on the GRIPP index structure from Figure 1(b). 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' in $O(G)$. But we need to check if w is in one of those. Therefore, we identify all hop nodes and recursively check their reachable instance sets by performing 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 node w in some reachable instance set or if there exists no further non-traversed hop node in a reachable instance set.

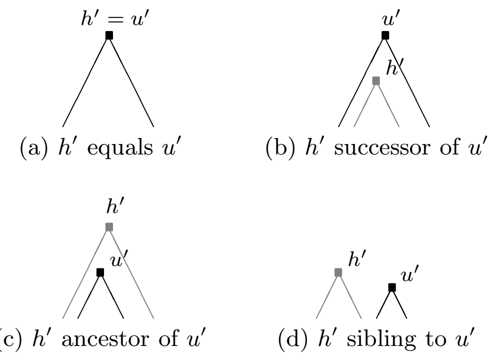
In $IND(G)$ there exist $m - n$ non-tree instances, each of which can be a hop node. Thus, querying GRIPP to answer $reach(v, w)$ requires in worst case $m - n$ queries. However, in the following we show pruning strategies that allow to query graphs on average in almost constant time as shown in Section 7.

4.2 Pruning strategies

Consider Figure 3(b) and $reach(D, r)$. We find non-tree instances of nodes B and A in $RIS(D)$. If we first use node 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, using B to retrieve $RIS(B)$ is not necessary; B can be pruned from the list of hop nodes.

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 have been used to retrieve a reachable instance set. 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' of h relative to the tree instances of nodes in U. There are four possible locations of h' in relation to the tree instance u' of a node $u \in U$ in $O(G)$. These are shown in Figure 4. h' either is (a) equal to, (b) a successor of, (c) an ancestor of, or (d) a sibling to u'. Given that we may consider all nodes in U for pruning, this results in four possible cases: (a) h' is equal to the tree instance of some node in U; (b) h' is successor of the tree instance of some node in U; (c) h' is ancestor to tree instances of nodes in U and neither (a) nor (b) is true; (d) h' is sibling to 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 $RIS(h)$. For the remaining three cases we can apply pruning strategies.



(a) h' equals u' (b) h' successor of u' (c) h' ancestor of u' (d) h' sibling to u'

Figure 4: Possible locations of h' of hop node h relative to u', $u \in U$.

In the first case, we can skip h entirely because 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, we can also skip h. In this case (see Figure 4(b)) there exists $u \in U$ such that h' is successor of u', i.e., $h' \in RIS(u)$ in $O(G)$. Thus, the entire reachable instance set of hop node h is contained in $RIS(u)$.

In the third case we have to be more careful. Consider Figure 3(b) and the query $reach(D, r)$. Assume, we have retrieved $RIS(D)$ and $RIS(B)$ and expand the search using A as hop node. $RIS(A)$ contains the tree instance of B and D and therefore also contains $RIS(B)$ and $RIS(D)$ as well. Thus, when we consider $RIS(A)$ we can *skip* the pre- and postorder range of $RIS(B)$ and $RIS(D)$.

4.2.1 Skip Strategy

We first assume that only one u' exists that is a successor of h'. Thus, the reachable instance set of u is contained in $RIS(h)$. This situation is displayed in Figure 4(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 stored the pre- and postorder value, 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'.

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, because we search U only a few times during a reachability query (see also Section 7) we believe the cost to merge ranges does not account for the gain of merging. Therefore, if multiple u exist in $RIS(h)$ each of their ranges is considered separately for skipping.

4.2.2 *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 check every hop node in $RIS(h)$ and start a recursive search. It would be advantageous if we knew in advance that in $RIS(h)$ no hop node exists that will extend the search, because in that case we do not have to query for the tree instances of hop nodes. We now show cases where this property can be precomputed.

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 4(a), 4(b)). We can precompute a list of nodes S for which all hop nodes have this property. We call these nodes *stop nodes* as their reachable instance sets will not extend the search.

Definition 5 (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 $RIS(s)$.*

Intuitively, a stop node s is a node in G for which for every non-tree instance in $RIS(s)$ exists a corresponding tree instance in the same set. This means, that all nodes reachable from s in G are reachable from s' 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 in $RIS(s)$.

When we reach the tree instance of a stop node s during the search we immediately know that we do not need to extend the search using hop nodes of $RIS(s)$. The GRIPP index structure in Figure 1 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.

5. THE IMPACT OF TRAVERSAL ORDER

The GRIPP index structure is created using an arbitrary yet fixed order of nodes. The chosen order does not influence the size of the index, as the space requirements to store the GRIPP index table is linear in the size of the graph. However, it has a strong influence on the performance of reachability queries. In the following, we describe an order which works extremely well on many types of graphs. In Section 6.1 we will show simple heuristics to approximate this order with minimal effort. Using this order, querying the GRIPP index requires on average significantly less than $m-n$ recursive calls; in fact, as our experiments in Section 7 show, the number of calls remains almost constant over all tested types of graphs.

Our idea is based on the following observations. In every graph one can identify strongly connected components C_1 ... C_k in linear time. Each component can be collapsed into a representative node (see Figure 5). The reachability information for nodes within one component are identical (this obvious optimization is used by many graph indexing strategies, such as [1] or [26]). Therefore, we can divide the problem of finding a good traversal order in two separate parts. First, find a good traversal order for nodes within one strongly connected component and second, find a good traversal order for the components in the component graph.

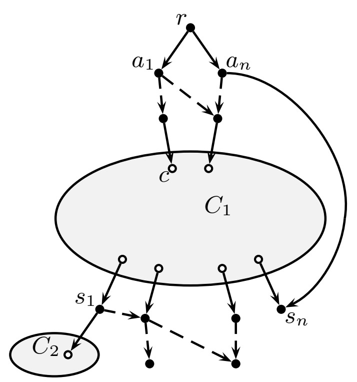


Figure 5: Structure of a graph. Solid lines indicate edges, dotted lines paths. The gray area contains all nodes and edges in the strongly connected components.

We first consider the traversal order for nodes within a strongly connected component C. Assume that during index creation we reach node $c \in C$. We add the tree instance of c to $IND(G)$. If no other node of C has been traversed before, we traverse all remaining nodes of C – all are reachable from c since C is a strongly connected component. Thus, every node in C will have a tree instance in $RIS(c)$ and we can answer $reach(v, w)$ for $v = c$ and $w \in C$ with a single lookup.

If $v \neq c$, but $v \in C$ the situation is different. Suppose $RIS(v)$ contains a non-tree instance of c and suppose we use c as first hop node. We then can answer $reach(v, w)$ (with $w \in C$) with two recursive calls, i.e., one to retrieve $RIS(v)$ and one for $RIS(c)$. To achieve this for every $v \in C$, we have to find a traversal order such that for every node $v \in C$, $RIS(v)$ contains a non-tree instance of c. We therefore must solve the following problem: Find a node $c \in C$ such that we can divide C in partitions $P_1, \ldots, P_n$ with n equals the indegree of c. For every P_i, $1 \leq i \leq n$ compute a Hamilton path starting at node v and ending at node c, with $v = c$ or v child node of a node in P_j, $j \neq i$. If we create GRIPP along those Hamilton paths we can ensure that for every node $v \in C$, $RIS(v)$ contains at least one non-tree instance of c.

Now suppose that we have not traversed any successor nodes of c in G when we traverse c, i.e., we have not traversed any nodes of C or any nodes in successor components of C. We traverse nodes in C along Hamilton paths and also traverse all nodes in successor components of C. This means all reachable nodes of c in G have a tree instance in $RIS(c)$. In addition, every non-tree instance in $RIS(c)$ must also have its corresponding tree instance in $RIS(c)$, i.e., c is a stop node. In Figure 6 the tree instance of c, c' is shown as double circled node in the gray area. Given $v \in C$ we can answer $reach(v, w)$ for any node $w \in G$ with at most two recursive calls, one initial call to test $RIS(v)$, finding a non-tree instance of c (or possibly already an instance of w), and a second call using c as hop node to test $RIS(c)$. As c is a stop node we do not have to use any further hop nodes, regardless if $RIS(c)$ contains an instance of w or not.

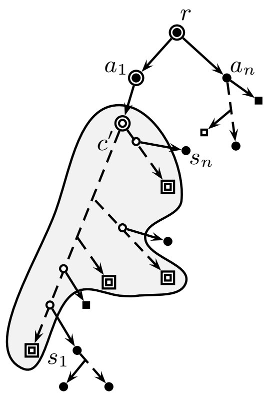


Figure 6: Optimal GRIPP index structure. Circles indicate tree instances, squares non-tree instances. The double circled node is the stop node, the double squared nodes are its non-tree instances. In gray is the area of instances of the giant strong component.

Therefore, we have to ensure that component C is traversed before any of its successor components in the component graph. Clearly, this is not possible for any C, but the problem is alleviated by the following observation. Erdös and Rényi [11] proved that directed random graphs with more edges that nodes contain one giant strongly connected component C. The size of C depends on the graph density. The experimental results given in Section 7 show that this is also true for our generated scale-free graphs. Therefore, graphs of a certain density usually appear as shown in Figure 5, with one component being very large (giant) and all other components being small. In this setting, it is only important to traverse the giant component before any of its successor components. The remaining successor components are traversed in descending order of the size of their successor sets, i.e., of the number of reachable nodes. Recall, for nodes in a component C that has been traversed before any of its successor components we can answer $reach(v, w)$, with $v \in C$ and $w \in G$, with two recursive calls.

We can also estimate the number of recursive calls to answer $reach(v, w)$ for every node $v \notin C$. If $RIS(v)$ contains no non-tree instance we can immediately return *false* using one call. Otherwise, we have to query GRIPP recursively, but we will at most use $m' - n'$ recursive calls with m' number of edges and n' number of nodes in the component graph.

In some cases this number can even be reduced. Consider the case where v is sibling to nodes in C and $RIS(v)$ only contains non-tree instances of nodes in C and possibly of nodes in successor components of C. Suppose we first use a node from C as hop node. We then need at most three recursive calls to answer $reach(v, w)$. One call to retrieve $RIS(v)$, finding the non-tree instance h' of a node $h \in C$ and using h as hop node, one call to retrieve $RIS(h)$, which contains a non-tree instance of c, and one call to test $RIS(c)$. If we can ensure this order of hop nodes we can also answer reachability queries for such cases with a constant number of calls.

Concluding, a good traversal order can be obtained as follows. First identify all strongly connected components and build the component graph. Using Tarjan's algorithm this takes $O(n+m)$ time. Second, determine the traversal order of components in the component graph by computing the size of the successor sets of all k components, which requires $O(k^3)$ time. Third, compute a good order for nodes within every component C by first identifying a node c and then computing Hamilton paths as described above. As finding Hamilton paths in graphs is NP-complete [9], this is not feasible for practical application. In the next section we present a simple heuristic for determining a traversal order, which, as we will show experimentally in Section 7, requires an almost constant number of calls to answer $reach(v, w)$ over different sizes, shapes and densities of graphs.

6. IMPLEMENTATION

In this section we present a suitable heuristic to compute a GRIPP index structure that works well on many types of graphs. We also present details on our implementation of the GRIPP indexing and search algorithm.

6.1 Giant Component and Node Order

During the creation of the GRIPP index for large graphs we want to avoid to compute the strongly connected components, as this also requires time. We found the following heuristic to work very well. To ensure that we traverse nodes of the giant strongly connected component before any other nodes we want to traverse a node from the giant strongly connected component as first node during index creation. Therefore we create a virtual root node (see Section 6.2) and attach an additional edge between the virtual root node and the node with the highest degree. This node can be found very quickly and, as nodes with a high degree tend to have many successor nodes and can be reached by many nodes, this node is very likely a member of the giant strongly connected component. Choosing this node has the additional advantage that it also has many incoming edges and therefore will get many non-tree instances in $IND(G)$. This means that it is likely to find a non-tree instance of that node in the reachable instance set of other nodes, and recall that this node is a stop node.

In the next step of the index creation we traverse child nodes of that node. We try to traverse the child node with the largest reachable instance set first as this node covers a large part of the remaining graph. We use the heuristic that a node with a high degree is likely to have a larger reachable instance set than a node with a lower degree. Therefore, we prefer child nodes with a high degree, i.e., we traverse child nodes according to their degree.

In Section 7 we show that using these heuristics we reach an almost constant query time over different sizes and shapes of graphs.

6.2 Virtual root node

We only explained the creation of the GRIPP index structure for graphs with a single root node. However, all kinds of graphs can be treated in the following way, essentially ignoring how many nodes have no incoming edges. We first add a virtual root node r to the graph. We add an edge between r and the node with the highest degree among all nodes. We then traverse and label the nodes as explained in Section 3 starting from r and using child nodes in the order of their degree. In general, some nodes will not be reached during this traversal, i.e., 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.

6.3 Stop node list

To create the list of stop nodes we have to check the reachable instance set of every node. As this is too time consuming for large graphs, we test only selected nodes. We are especially interested in nodes whose reachable instance set covers a large amount of instances. Therefore, we only consider child nodes of the virtual root node as stop node candidates. Additionally, we require that the size of the reachable instance set of a stop node candidate exceeds a certain threshold t. Furthermore, we only test a 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' of every hop node in $RIS(s)$ has a preorder value that is lower than the preorder value of the tree instance s' of s. If that is the case, h' is sibling to s' in $O(G)$ and s is not a stop node; otherwise, s is a stop node.

6.4 Query algorithm

The GRIPP index as well as all temporary information (stop nodes, visited hop nodes, etc.) are stored in relational tables. The instance type of a node is stored as special attribute in the index table. We created b-tree indexes on relevant attributes, including a combined index on the attributes preorder, node, and instance type. To answer $reach(v, w)$ Algorithm 1 starts by testing $w \in RIS(v)$ with a query over the index. It then adds v to the list U of used nodes. If v is a stop node, the algorithm stops.

Otherwise, we perform a depth-first search considering non-tree instances in $RIS(v)$ in ascending order of their preorder rank as hop nodes, unless $RIS(v)$ contains a non-tree instance of a stop node, which is preferentially used. In the next step we select all hop nodes from $RIS(v)$ which are not already covered by another reachable instance set. For every hop node h we determine the location of its tree instance h' and test if $RIS(h)$ is completely or partly covered from nodes in U. If not, we proceed, 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.

6.5 Practical Applicability

The GRIPP indexing and query algorithm is implemented as stored procedure. Therefore, there is virtually no limit in the size of the graphs, as all operations are performed as SQL queries leveraging the memory management of the underlying RDBMS. As an additional advantage, GRIPP may be integrated very easily into all applications that store graph-like data in a RDBMS. All that needs to be done is the installation of stored procedures. Views can be used to create the expected table structure for the indexing function. The index is stored in a separate table. Then, reachability of two nodes can be tested by a simple call of a user-defined SQL function.

Algorithm 1: Function to answer $reach(v, w)$ using the GRIPP index.

```
used_hops ← ∅
used_stops ← ∅
FUNCTION reachability(v, w) RETURNS boolean
    if w ∈ RIS(v) then
        return true
    else
        used_hops ← used_hops ∪ (v)
        if v ∈ STOP_NODES then
            used_stops ← used_stops ∪ (v)
            return false
        else
            while non_tree_inst ← nextStop(RIS(v)) do
                tree_inst ← getTreeInst(non_tree_inst)
                if reachability(tree_inst, w) then return
                true
            end
            if isInRIS(v, used_stops) then
                return false
            end
            H_1 ... H_n ← getUsedHopsInRIS(v)
            // skip ranges
            non_tree_instances ← getNonTreeInst(RIS(v) \
            RIS(H_1) \ ... \ RIS(H_n))
            foreach non_tree_inst ∈ non_tree_instances do
                tree_inst ← getTreeInst(non_tree_inst)
                if !hasChildren(tree_inst) then
                    continue
                end
                // if new hop has been used as hop
                if tree_inst ∈ used_hops then
                    continue
                end
                // if new hop is in a RIS of a used hop
                if isInRIS(tree_inst, used_hops) then
                    continue
                end
                // otherwise call recursively
                if reachability(tree_inst, w) then return
                true
                if isInRIS(v, used_stops) then
                    return false
                end
            end
            return false
        end
    end
end
```

7. EXPERIMENTAL RESULTS

To evaluate our approach we use synthetic as well as real-world data. We compare GRIPP in detail to the Dual Labeling approach from Wang et al. [26] and the Labeling+SSPI approach from Chen et al. [6]. Both algorithms can only index directed, acyclic graphs (DAG). Therefore we first identify strongly connected components of G and collapse each component into a representative node. This step takes $O(n+m)$ using Tarjan's algorithm [9]. The resulting component graph is a DAG. To compare our approach we created and queried the GRIPP index for the component graph as well as for the graph itself. For a more detailed comparison of GRIPP with TC and recursive functions see [25].

For synthetic data we created random as well as scale-free graphs in the size of 1,000 to 5,000,000 nodes and 0 to 2,000% more edges than nodes using the methods described in [3]. The degree distribution in scale-free graphs follows a power law with an exponent $\gamma = 2.7$. As real-world data we

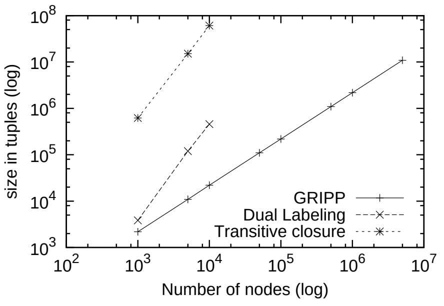
Table 2: Average time and size for different indexing methods on synthetic scale-free graphs with 100 % more edges than nodes.

(a) Average time (sec).

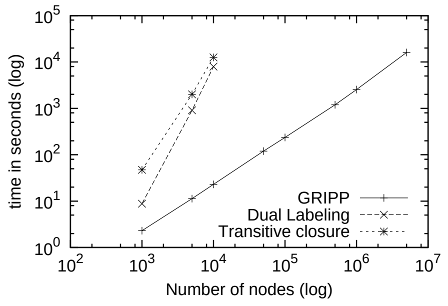
	Component Graph			Dual Labeling	GRIPP		Label + SSPI
No. nodes	No. nodes	No. edges	Time		GRIPP index	Stop nodes	
1,000	422.2	588.8	2.9	8.8	0.8	0.2	1.1
5,000	2,184.6	3,111.8	16.1	906.8	4.0	2.8	6.9
10,000	4,324.6	6,152.0	34.1	7,937.6	8.0	6.0	16.5
50,000	21,816.0	31,110.4	278.1	> 86,400.0	41.3	33.6	208.9

(b) Average number of tuples.

	Component Graph		Dual Labeling		GRIPP		Label + SSPI	
No. nodes	No. nodes	No. edges	Node labels	TLC values	GRIPP index	Stop nodes	Node labels	SSPI
1,000	422.2	588.8	423.2	3,431.8	768.2	1.0	423.2	533.2
5,000	2,184.6	3,111.8	2,185.6	117,699.8	3,975.0	1.0	2,185.6	2,838.4
10,000	4,324.6	6,152.0	4,325.6	452,693.8	7,969.6	1.0	4,325.6	5,583.4
50,000	21,816.0	31,110.4	-	-	39,905.0	1.0	21,817.0	28,267.0



(a) Average size (tuples)



(b) Average time (sec)

Figure 7: Average time and size for the GRIPP index table, Dual Labeling on the component graph, and the transitive closure for synthetic scale-free graphs with 100 % more edges than nodes.

used data of metabolic networks provided by KEGG [18], aMAZE [20], and Reactome [17]. Nodes represent enzymes, chemical compounds or reactions, while edges represent the participation of an enzyme or compound in a reaction. The degree distribution in metabolic networks follows a power law with exponent $\gamma = 3.0$, i.e., they are also scale-free. Properties of these data set can be seen in Table 4.

We implemented GRIPP as well as all competitive methods (based the original code kindly provided by their authors) as stored procedures in a commercial object-relational database system. Tests were performed on a DELL dual Xeon machine with 4 GB RAM. Queries were run without rebooting the database. The indexing times are averaged over five different graphs for every number of nodes and edges. The query times for $reach(v, w)$ are averaged over 5,000 randomly selected node pairs for every number of nodes and edges.

For the index creation and querying we also compared GRIPP to computing the transitive closure for the entire graph. Clearly, querying the transitive closure is the fastest method, but we cannot compute the transitive closure for graphs containing more than 10,000 nodes and 20,000 edges in feasible time and the resulting structure would contain over 60 million tuples. We also compared GRIPP to recursive query strategies, which need no index creation at all. We used our own implementation of a recursive search and the recursive SQL command available in the RDBMS. Our own implementation of a recursive traversal is always outperformed by GRIPP and all competing methods. Even in graphs having the small world characteristic, i.e., where each node can be reached from each node within ~ 6 steps, a breadth-first strategy requires in the order of d^6 calls, where d is the average out-degree of nodes. The built-in recursive SQL command 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 all paths are enumerated in the graph beginning from the start node.

7.1 Index Creation

Table 2(a) shows the average time required to index scale-free graphs with 1,000 to 50,000 nodes and 100 % more edges than nodes. The component graph has on average 43% of the nodes and 31% of the edges of the original graph, i.e., the component graph is much smaller than the original graph. All used graphs contain one giant strongly connected component. For instance, scale-free graphs with 50,000 nodes and 100,000 edges have one giant strongly connected component that contains on average 28,184 nodes, i.e., more than half the nodes of the entire graph. The remaining components usually contain only one node.

For graphs of 50,000 or more nodes we could not compute the Dual Labeling within 24 hours using our database-based re-implementation. We also tried the C++-based main-memory implementation of Dual Labeling provided by the authors of this algorithm. Compared to our re-implementation, their program is much faster for small graphs, but the program breaks for graphs with 50,000 or more nodes. In contrast, computing the GRIPP index table on the component graph for the same component graphs took less than 50 seconds. Computing the GRIPP index on the entire graph requires about 120 seconds. Our results support the analysis that the time complexity of Dual Labeling is $O(n + m + t^3)$, e.g., computing the index for 50,000 nodes and 100,000 edges might take almost two weeks. In contrast, computing the GRIPP index as well as the Label+SSPI index is linear in the number of edges. Therefore, both indexes can be computed for even larger graphs. We show this in the following for GRIPP.

Table 2(b) shows the average size of the index structures. Dual Labeling generates by far the largest index, mainly due to the TLC values. The TLC values basically represent a condensed transitive closure over the remaining edges. But the index structure is two orders of magnitude smaller than the transitive closure over the entire graph. For instance, for scale-free graphs with 10,000 nodes and 20,000 edges Dual Labeling requires on average 460,000 tuples, while the transitive closure requires on average over 60 million tuples.

GRIPP and Label+SSPI require space linear in the size of the graph. The GRIPP index is slightly smaller than Label+SSPI, because GRIPP creates one tuple for every edge plus one tuple for every child to the virtual root node. Label+SSPI creates one tuple for every node in the component graph (Node labels) and stores for every node that has more than one parent node all parent nodes in the SSPI index. In addition one tuple is created for every node that has a parent node with an entry in the SSPI index, i.e., in worst case this index has the size of m. This worst case is almost reached for the indexed graphs.

The figures for random graphs (data not shown) for all three methods are almost identical to the figures for scale-free graphs.

To test the scalability of GRIPP we created the index for graphs with 1,000 to 5,000,000 nodes and 100 % more edges than nodes. We did not compute the component graph, but applied the GRIPP indexing algorithm directly to the graph. Figure 7 shows the computation time and size of the GRIPP index, Dual Labeling, and the transitive closure for synthetic scale-free graphs. The data support our claim that GRIPP can be computed in linear time and space. In worst case, i.e., for a graph with $n - 1$ nodes without incoming edges and m edges GRIPP has the size of $n - 1 + m$. Figures for random graphs are comparable (data not shown).

We also indexed graphs with 100,000 nodes and increasing graph density (data not shown). The data show 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.

Concluding, GRIPP and Label+SSPI are highly scalable in terms of index creation, while Dual Labeling can not be applied to large graphs. In the next section we evaluate the query performance.

7.2 Query times

We compare querying GRIPP with querying the other two indexing methods. For the comparison we randomly selected 5,000 node pairs for every number of nodes and edges and computed $reach(v, w)$.

Table 3(a) shows the average number of recursive calls for the different query strategies on scale-free graphs with 1,000 to 50,000 nodes and 100 % more edges than nodes. Dual Labeling requires only one call to answer $reach(v, w)$ using the index structure. The number of recursive calls for the Label+SSPI strategy depends on the size of the graph. For graphs of 50,000 nodes and 100,000 edges it requires on average 994 recursive calls, ranging from 1 call for a pair of nodes in the same component to 11,504 calls in worst case. This explains the high standard deviation.

Table 3: Average number of calls and average query time to answer $reach(v, w)$ for the three different query strategies on scale-free graphs with 100% more edges than nodes.

(a) Average number of calls.

No. nodes	Dual Labeling	GRIPP DAG	Label+SSPI
1,000	1.0 ± 0.00	1.8 ± 0.74	22.0 ± 52.30
5,000	1.0 ± 0.00	1.9 ± 0.82	92.1 ± 238.31
10,000	1.0 ± 0.00	1.8 ± 0.77	194.7 ± 497.68
50,000	-	1.9 ± 0.77	944.3 ± 2,419.83

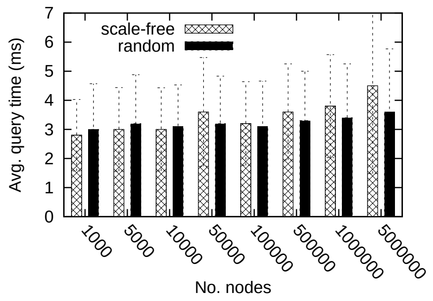
(b) Average query time (ms).

No. nodes	Dual Labeling	GRIPP DAG	Label+SSPI
1,000	0.8 ± 0.33	1.6 ± 1.45	5.9 ± 13.39
5,000	0.8 ± 0.32	2.0 ± 2.15	22.7 ± 59.39
10,000	0.8 ± 0.32	2.1 ± 2.56	48.8 ± 127.67
50,000	-	4.4 ± 6.74	253.0 ± 637.68

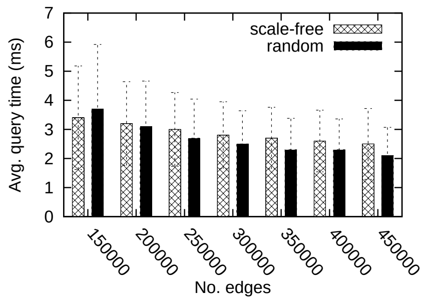
When querying the component graph (DAG) as well as the graph itself using GRIPP the number of recursive calls remains almost constant over different sizes of graphs, supporting our analysis from Section 5 and selection of heuristics. The maximum number of recursive calls is between 7 and 8 for different sizes of scale-free graphs. The number of recursive calls for GRIPP on DAGs is smaller than on graphs. The reason is that we do not require a recursive call for nodes in the same component, i.e., we can immediately answer $reach(v, w)$ if both nodes are in the same component.

The query times shown in Table 3(b) for the different strategies correspond well with the number of recursive calls. Dual Labeling requires on average 0.8 ms regardless the size of the graph. For GRIPP on the component graph the average query times range from 1.6 to 4.4 ms while for Label+SSPI the query times range from 5.9 to 253.0 ms. The time difference between GRIPP and Label+SSPI strategy grows as the number of nodes and edges increases. The same is true for random graphs (data not shown).

Figure 8(a) shows the average time necessary to answer $reach(v, w)$ using GRIPP on scale-free and random graphs. The query times increase slightly with increasing number of nodes. The reason is that reachable instance sets become larger. As these are accessed through b-tree indexes the increase is sublinear. The number of recursive calls remains with 2.3 almost constant over the different sizes of graphs with constant density (data not shown), supporting our analysis from Section 5. The maximum number of recursive calls ranges from 6 calls for the graph with 1,000 nodes to 10 calls for the graph with 5,000,000 nodes.



(a) Graphs with 100 % more edges than nodes.



(b) Graphs with 100,000 nodes.

Figure 8: Average query times (ms) and standard deviation for synthetic random and scale-free graphs.

Table 4: Indexing and querying real-world graphs using GRIPP.

Database	**Graph size**			**GRIPP index**		**Stop nodes**		**Querying GRIPP**	
	No. nodes	No. edges	Density	Time (sec)	No. Tuples	Time (sec)	No. Tuples	Avg. query time (ms)	Avg. No. recursive calls
Reactome	3,677	14,447	3.93	21.1	14,906	0.6	22	4.63 ± 4.016	2.56 ± 1.124
aMAZE	11,876	35,846	3.02	35.4	37,568	0.1	1	3.43 ± 1.597	2.25 ± 0.967
KEGG	14,269	35,170	2.46	37.2	36,527	0.1	1	3.34 ± 1.430	2.36 ± 0.913

Figure 8(b) shows the average query time for graphs with 100,000 nodes and increasing density. We observed that with increasing density the number of recursive calls for GRIPP even decreases. For instance, on scale-free graphs with 100,000 nodes and 150,000 edges GRIPP requires on average 2.3 recursive calls to answer $reach(v, w)$. In contrast, for scale-free graphs with 100,000 nodes and 450,000 edges GRIPP requires on average only 1.8 and the time drops from 3.4 ms to 2.5 ms. This trend continues as the density increases (tested for graphs with 100,000 nodes and up to 2,000,000 edges). There are two reasons for this. First, with increasing graph density the size of the giant strongly connected component also increases, i.e., more nodes are reachable from the first traversed node. Therefore, when reaching that node, a large fraction of the graph is already covered and less recursive calls are necessary. The second reason is that more and more nodes receive non-tree instances in GRIPP. This means with increasing density the chance increases that $RIS(v)$ contains an instance of w.

With further increasing graph density, Dual Labeling and Label+SSPI will also perform better as the size of the component graph decreases. For Dual Labeling this means that generating the index will become faster, and for Label+SSPI indexing as well as querying will be faster.

7.3 Real world graphs

To evaluate GRIPP on real-world graphs we used the metabolic networks provided by Reactome [17], aMAZE [20], and KEGG [18]. Table 4 shows the properties of the graph, i.e., number of nodes and edges and density. The table also shows the time required to compute the GRIPP index and the stop node list. The times correspond well with the times for generated graphs of comparable size.

The table also shows the the average number of calls and average time to answer $reach(v, w)$. The average number of calls as well as the average query time is slightly higher than for synthetic scale-free graphs of comparable size. This indicates that, although the networks are also scale-free, they still have a different structure than synthetic graphs.

8. CONCLUSION

We presented the GRIPP index structure for reachability queries on directed graphs. Since creating GRIPP requires only linear time and space, it can be used to index graphs with five million and more nodes. We showed analytically and experimentally that using GRIPP we can answer reachability queries on many types of graphs in almost constant time using an almost constant number of calls. As GRIPP is entirely based on SQL it can easily be integrated into existing graph applications.

No graph index structure suits all possible graph applications equally well. We tested GRIPP on synthetic random and scale-free graphs and on real biological datasets of various sizes and shapes and obtained very favorable results. GRIPP works particular well on large graphs that contain one large strongly connected component, which is a typical feature of graphs having a density above a certain threshold. For very small graphs whose indexes can be computed and held in main memory, GRIPP is outperformed by methods based on transitive closure or variations of it, such as Dual Labeling. The later is also superior for very sparse graphs, as long as they have below ∼ 10.000 nodes. For denser graphs the component graphs typically shrinks enormously (as almost all nodes fall into one component), which favors all methods that first compute the component graph (including GRIPP-DAG). However, GRIPP is by far the fastest method for indexing typical and large biological networks. This observation very likely carries over to other types of graphs such as social networks or Web graphs, as these share many characteristics with biological networks [3]. Finally, GRIPP is highly advantageous for any application which stores and analyzes graphs in a RDBMS since its integration is very easy.

In the future we plan to include GRIPP as indexing component into a comprehensive graph query language. We will study extensions of GRIPP to support distance (length of the shortest path between two nodes) and path length queries (all paths between two nodes of a certain length).

Finally, for this purpose GRIPP needs to be adapted to set-oriented query semantics. A typical query would have to compute, given a node v and a set of nodes W, all nodes in W reachable from v. We are confident that there are better ways of using the GRIPP index structure for such queries than calling the reachable function $|W|$ times.

Acknowledgments

This work is supported by BMBF grant no. 0312705B.

9. 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 Press.

[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. Barabási and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, Oct 1999.

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

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

[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 Press.

[7] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast Computation of Reachability Labeling for Large Graphs. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT)*, volume 3896 of *Lecture Notes in Computer Science*, pages 961–979, 2006. Springer.

[8] 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.

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

[10] 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 Press.

[11] P. Erdös and A. Rényi On the Evolution of Random Graphs. *Publ. Math. Inst. Hungar. Acad. Sci.*, 5:17 – 61, 1960.

[12] 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.

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

[14] 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.

[15] 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 Press.

[16] J. van Helden, A. Naim, R. Mancuso, M. Eldridge, *et al.* Representing and analysing molecular and cellular function using the computer. *Journal of Biological Chemistry*, 381(9-10):921–935, Sep-Oct 2000.

[17] G. Joshi-Tope, M. Gillespie, I. Vastrik, P. D'Eustachio, *et al.* Reactome: a knowledgebase of biological pathways. *Nucleic Acids Research*, 33:D428–D432, Jan 2005.

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

[19] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A declarative query language for RDF, 2002. In *Proceedings of the 11th Intl. World Wide Web Conference (WWW)*, 2002.

[20] C. Lemer, E. Antezana, F. Couche, F. Fays, *et al.* The aMAZE LightBench: a web interface to a relational database of cellular processes. *Nucleic Acids Research*, 32:D443–D448, Jan 2004.

[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] 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.

[23] U. Stelzl, U. Worm, M. Lalowski, C. Haenig, *et al.* A human protein-protein interaction network: a resource for annotating the proteome. *Cell*, 122(6):957–968, Sep 2005.

[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] S. Trißl and U. Leser. GRIPP - Indexing and Querying Graphs based on Pre- and Postorder Numbering. Technical Report No. 207, Humboldt-Universität zu Berlin, Institut für Informatik, 2006.

[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.