Compact Reachability Labeling for Graph-Structured Data

Hao He† Haixun Wang‡ Jun Yang† Philip S. Yu‡

†Duke University
Durham, NC 27708
{haohe, junyang}@cs.duke.edu

‡IBM T. J. Watson Research
Hawthorne, NY 10532
{haixun, psyu}@us.ibm.com

ABSTRACT

Testing reachability between nodes in a graph is a well-known problem with many important applications, including knowledge representation, program analysis, and more recently, biological and ontology databases inferencing as well as XML query processing. Various approaches have been proposed to encode graph reachability information using node labeling schemes, but most existing schemes only work well for specific types of graphs. In this paper, we propose a novel approach, HLSS(*H*ierarchical *L*abeling of *S*ub-*S*tructures), which identifies different types of substructures within a graph and encodes them using techniques suitable to the characteristics of each of them. We implement HLSS with an efficient two-phase algorithm, where the first phase identifies and encodes strongly connected components as well as tree substructures, and the second phase encodes the remaining reachability relationships by compressing dense rectangular submatrices in the transitive closure matrix. For the important subproblem of finding densest submatrices, we demonstrate the hardness of the problem and propose several practical algorithms. Experiments show that HLSS handles different types of graphs well, while existing approaches fall prey to graphs with substructures they are not designed to handle.

Categories and Subject Descriptors: H.3.1 [Information Storage and Retrieval]: Content Analysis and Indexing

General Terms: Algorithms, Design, Performance

Keywords: graph reachability labeling

1. INTRODUCTION

Consider a directed graph $G = (V, E)$. *Graph reachability* is the following decision problem: Given two nodes u and v in G, is there a path from u to v? If the answer is yes, we say that u can *reach* v, or $u \xrightarrow{*} v$.

Graph reachability has been a well-known problem with many traditional applications, e.g., testing concept subsumption in knowledge representation systems; and reasoning about inheritance in compiler design for object-oriented programming languages. Recently, the interest in graph reachability work has been rekindled by new applications of graph-structured databases. There are several well-known projects in bioinformatics, including the Biopathways Graph Data Manager [11] and the BioCyc project [18]. They model data such as protein interactions, metabolic pathways and gene regulatory networks as directed graphs. Nodes in them represent entities such as compounds, promoters and proteins whereas edges specify how entities are related. In these projects, researchers are interested in reachability questions such as whether a reactant u might indirectly activate or inhibit protein v through some chain of reactions. In the Semantic Web, two key technologies RDF and OWL are designed to capture graph data. Reasoning and subsumption query on them are both reachability queries. In addition, although XML is generally modelled as a tree, there exist many XML applications where cross-reference edges (through IDREF/ID) are treated as first-class citizens [16, 15, 7], making the data graph-structured. In this case, the ancestor/descendant axis "$u//v$" of XML query is an instance of graph reachability query. Finally, the reachability query is also a basic building block of other types of graph queries such as subgraph isomorphism. Efficient support for reachability testing is crucial since it might be invoked heavily for large data and complex queries.

The well-known single-source shortest path algorithm can be used to answer reachability queries. However, the algorithm has a high complexity of $O(|E|)$, making it infeasible for efficient query processing. On the other extreme, we can precompute and store the transitive closure of the graph. Reachability queries can then be answered with constant-time matrix lookups. However, the space requirement is $O(|V|^2)$, making it infeasible for large graphs.

If we only consider reachability in trees (or forests), *interval labeling* is a nice solution that takes linear space and supports reachability queries in constant time. This approach can be dated back to the 1980's [10], and has been extensively applied to XML query processing in recent years (e.g., [20, 3, 4]). It labels each node u in the tree by an interval $[start(u), end(u)]$. The labels can be assigned with a depth-first traversal of the tree, using a counter that is incremented whenever the traversal enters or leaves a node; $start(u)$ and $end(u)$ are assigned the value of the counter when the traversal enters or leaves u, respectively. It is not difficult to see that interval labeling has the following property: Given two tree nodes u and v, $u \xrightarrow{*} v$ iff $[start(u), end(u)] \supseteq [start(v), end(v)]$. Thus, reachability can be verified in constant time. Unfortunately, this approach is not directly applicable to graphs.

Labeling a general graph to support efficient reachability queries is a hard problem. Cohen et al. [9] has shown that there exist graphs for which any reachability labeling scheme would require $O(|V| \times |E|^{1/2})$. Still, a variety of labeling schemes have been proposed, and we survey them in Section 2. Briefly, the two most relevant and popular schemes are the *interval-based approach* by Agrawal et al. [1] and the 2-*hop approach* by Cohen et al. [9]. The interval-based approach extends the basic interval labeling to work on DAGs, and is effective on graphs that mostly resemble trees or forests. However, the performance degrades when the graph contains many non-tree edges. The 2-hop approach identifies subgraphs where one set of nodes connect to another set of nodes via a "hop" node; between these two sets, reachability relationships can be encoded compactly. This approach is thus optimized for graphs that contain many good "hop" nodes, i.e., nodes that connect two large sets of other nodes. However, the approach is less efficient for graphs with other types of substructures, e.g., long, branchless paths or one-way bipartite graphs.

In summary, there is currently no single approach that works well for all types of graphs. Our goal is to develop a labeling scheme that is robust for a larger variety of graphs. We note that each existing approach to reachability labeling exploits certain substructural features in graphs. We seek to combine the strengths of these different approaches to achieve generality of our labeling scheme. More specifically, this paper makes the following contributions:

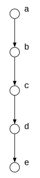
- We propose a hierarchical approach to reachability labeling called *HLSS* (*H*ierarchical *L*abeling of *S*ub-*S*tructures). A graph often contains different types of substructures whose reachability is easier to encode with different labeling techniques. We argue that we should extract such substructures and apply efficient techniques suitable to each of them.
- To implement HLSS, we propose a two-phase labeling algorithm. The first phase identifies and encodes strongly connected components as well as tree substructures. The second phase encodes the remaining reachability relationships by compressing dense rectangular submatrices in the transitive closure matrix. The hierarchical approach handles different types of graphs well, while existing approaches fall prey to graphs with substructures they are not designed to handle.
- For the important subproblem of finding densest submatrices, we demonstrate the hardness of the problem and propose three algorithms, including a 2-approximation algorithm, an algorithm that enhances the 2-hop approach, and an algorithm based on singular-value decomposition. A notable feature of these algorithms is that they allow false positives to encourage larger submatrices, which can be encoded more efficiently; meanwhile, they consider the cost of filtering out false positives to balance this benefit.

2. BACKGROUND AND RELATED WORK

Before discussing related work, we recast the problem of graph reachability labeling as the problem of finding a compact representation for a transitive closure matrix. From this viewpoint, we will analyze the two most popular existing approaches, interval-based and 2-hop, and then discuss other related work.

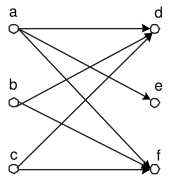
We can represent a directed graph $G = (V, E)$ by its $|V| \times |V|$ binary adjacency matrix $\mathbf{A}$, where $a_{ij} = 1$ iff there is an edge from the i-th node to the j-th node. The transitive closure of G can be represented by a $|V| \times |V|$ binary matrix $\mathbf{T} = \mathbf{A}^{|V|}$, the $|V|$-th power of $\mathbf{A}$. If we compute and store $\mathbf{T}$, we can answer reachability queries in constant time by looking up appropriate entries in $\mathbf{T}$. However, it is too costly to store $\mathbf{T}$ for large graphs. The goal of reachability labeling is indeed to represent a transitive closure matrix in a succinct way that supports reachability queries efficiently.

Interval-Based Approach As described in Section 1, the interval approach labels nodes by intervals, whose containment relationships encode ancestor-descendant relationships among nodes in a tree. In the transitive closure matrix, each directed path in the graph corresponds to a reordered submatrix with 1's in the upper triangle and 0's in the lower triangle (Figure 1). This submatrix can be en-



$$\begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 1 & 1 & 1 & 1 & 1 \\ b & 0 & 1 & 1 & 1 & 1 \\ c & 0 & 0 & 1 & 1 & 1 \\ d & 0 & 0 & 0 & 1 & 1 \\ e & 0 & 0 & 0 & 0 & 1 \end{array}$$

Figure 1: A path and its corresponding submatrix in T.



$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline a & 1 & 0 & 0 & 1 & 1 & 1 \\ b & 0 & 1 & 0 & 1 & 0 & 1 \\ c & 0 & 0 & 1 & 1 & 0 & 1 \\ d & 0 & 0 & 0 & 1 & 0 & 0 \\ e & 0 & 0 & 0 & 0 & 1 & 0 \\ f & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

Figure 2: A bipartite graph and its transitive closure matrix T.

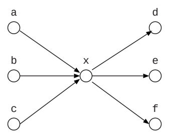
coded succinctly by labeling the nodes involved with nested intervals. Thus, the interval approach is effective in compressing those transitive closure matrices that contain many such upper triangular submatrices. This approach works especially well for graphs with long paths: The longer the path, the better the compression ratio.

Although originally proposed for trees [10], the interval-based approach was extended by Agrawal et al. [1] to DAGs. Each node u is assigned a set of non-overlapping intervals $L(u)$; $u \overset{*}{\rightarrow} v$ iff every interval in $L(v)$ is contained in some interval in $L(u)$. Labeling is done by first finding a spanning forest and assigning interval labels for nodes in the forest. Next, to capture reachability relationships through non-spanning-forest edges, we add additional intervals to labels in reverse topological order of the DAG; specifically, if (u, v) is an edge not in the spanning forest, then all intervals in $L(v)$ are added to $L(u)$ (as well as labels of all nodes that can reach u). Consider the graph in Figure 2. Using the spanning tree rooted at a, we label a, d, e, and f with $[1, 8]$, $[2, 3]$, $[4, 5]$, and $[6, 7]$, respectively; we label b and c with $[9, 10]$ and $[11, 12]$ as they belong to separate trees in the spanning forest. Both b and c also receive intervals from d and f, resulting in $L(b) = \{[9, 10], [2, 3], [6, 7]\}$ and $L(c) = \{[11, 12], [2, 3], [6, 7]\}$.

With more complicated graphs, the size of a label can become linear in the graph size. For example, if d and f have many non-spanning-forest descendants, their labels will become larger, which will in turn cause ancestors of b and c to have larger labels. Since reachability queries involves checking containment for all intervals in a label, large labels can seriously impact query performance.

2-**Hop Approach** The 2-hop approach was recently proposed as an alternative to the interval-based approach by Cohen et al. [9]. For each node u, let $C_{in}(u)$ denote the set of nodes that can reach u, and $C_{out}(u)$ the set of nodes that can be reached by u. The key observation of this approach is that every node in $C_{in}(u)$ can reach every node in $C_{out}(u)$. For example, in Figure 3, $C_{in}(x) = \{a, b, c, x\}$ and $C_{out}(x) = \{x, d, e, f\}$. The 2-hop approach assigns each node two sets of nodes as its *in-label* and *out-label*, such that $u \overset{*}{\rightarrow} v$ iff the out-label of u intersects with the in-label of v. Thus, reachability relationships from nodes in $C_{in}(x)$ to nodes in $C_{out}(x)$ can be encoded succinctly by adding x to the out-label of every node in $C_{in}(x)$, and the in-label of every node in $C_{out}(x)$.

From the viewpoint of compressing the transitive closure matrix, the 2-hop approach seeks to compress the submatrix induced by x consisting of all ones; its columns correspond to the nodes in $C_{in}(x)$ and its rows correspond to the nodes in $C_{out}(x)$, as illustrated in Figure 3. Thus, this approach works especially well on graphs with many well-connected hop nodes. Its effectiveness de-



$$\begin{array}{c|cccc} & x & d & e & f \\ \hline a & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 \\ c & 1 & 1 & 1 & 1 \\ x & 1 & 1 & 1 & 1 \end{array}$$

Figure 3: Two sets of nodes connected through a single node, and the corresponding submatrix in T.

pends on the area-to-circumference ratio of submatrices identified for compression: The larger the area compared with circumference, the better the compression ratio. The 2-hop algorithm repeatedly and greedily encodes the submatrix induced by x that maximizes $\frac{|C_{in}(u)|\times|C_{out}(u)|-k}{|C_{in}(u)|+|C_{out}(u)|}$, where k is the number of ones in the submatrix that have been previously encoded.

On the other hand, it is obvious from the matrix compression viewpoint that the 2-hop approach may miss many submatrices that are high-quality candidates for compression, because the approach only considers submatrices induced by hop nodes. For example, in Figure 2, the submatrix spanned by rows $\{a, b, c\}$ and columns $\{d, f\}$ consists of all ones and would be a good choice to compress, but it is not induced by a hop node.

Other Existing Approaches Graph reachability encoding has a wide range and long history. In objected-oriented programming languages, a series of schemes based on bit vector encoding were developed for labeling class inheritance hierarchies. The bottom-up labeling scheme by Aït-Kaci et al. [2] (called *modulation*) uses $O(n)$ bits per label, where n is the number of nodes. The top-down labeling scheme by Caseau [5] (called *compact hierarchical encoding*) exploits possible reuses of bit positions to achieve more compact encoding, although in the worst it still requires $O(n)$ bits per label. In network optimization, Katz et al. [14] proposed a scheme for labeling flow and connectivity in weighted flow graphs; however, their scheme assumes undirected graphs, for which reachability is much simpler to encode than for directed graphs.

Recently, graph reachability labeling problem has enjoyed renewed interest because of its application in XML and the Semantic Web. Vagena et al. [19] investigated techniques for evaluating twig queries over graph-structured XML data. They used the 2-hop approach to label directed graphs, and noted that planar DAGs could be handled more efficiently from [13]. For a non-planar DAG, they proposed converting it into a planar DAG by adding dummy nodes at crossing points; however, this approach may introduce many false positives that may be expensive to filter out subsequently.

Christophides et al. [8] applied efficient labeling schemes to the problem of encoding subsumption hierarchies for the Semantic Web. They compared three main families of labeling schemes: Besides the interval-based and bit vector encoding approaches discussed earlier, they also considered prefix-based approach exemplified by Dewey encoding [6]. To extend Dewey encoding to a DAG, they proposed propagating each node's Dewey prefix (obtained from a spanning forest) to all nodes that it can reach in the graph. Each result label consists of a set of Dewey prefixes, which can then be compressed by eliminating those that are strict prefixes of others in the set. However, since the length of each Dewey prefix is proportional to the depth of the spanning forest, this approach does not work well for graphs with tall spanning forests; recursive propagation of Dewey prefixes further increases the size of the labels.

3. OVERVIEW OF HLSS

As we have seen, existing approaches to labeling graph reachability each have their respective strengths and weaknesses, and are most effective on specific types of graphs. The interval-based approach works best on tree-like graphs, while the 2-hop approach works best on graphs with many well-connected hop nodes. By combining the power of these approaches and other optimizations that we have developed, we propose HLSS, a hierarchical labeling scheme that can work well for graphs with different characteristics.

HLSS assigns labels in two phases, each focusing on exploiting different characteristics of the input graph G. The first phase, *tree-reachability reduction*, begins with a preprocessing step that identifies each strongly connected component, collapses the component into one representative node, and uses this node to label others in the component. Then, we identify tree structures in G and assign interval labels to nodes based on these tree structures. Containment of interval labels implies reachability through tree paths. This phase also computes a *remainder graph* G_r that captures any remaining reachability information not encoded by interval labels. Specifically, a node can reach another node through *portals* in G_r. We label nodes by their portals to facilitate reachability checking.

The second phase, *remainder graph-reachability encoding*, aims at compressing the reachability information in the remainder graph G_r produced by the first phase. We do so by assigning additional labels to portals so that reachability among them can be checked efficiently by comparing their labels. We will present several techniques for assigning such labels, including an enhanced version of the 2-hop approach as well as techniques inspired by data mining, linear algebra, and graph algorithms.

To summarize, these two phases will create a label of a 4-level hierarchy for each node u:

- A *strongly connected component label*, $l_s(u)$, which is a representative node in the strongly connected component containing u, if any. It is assigned by the tree-reachability reduction phase.
- A pair of numeric *interval labels*, $l_i^{\llcorner}(u)$ and $l_i^{\lrcorner}(u)$, which form the interval $[l_i^{\llcorner}(u), l_i^{\lrcorner}(u)]$. They are assigned by the tree-reachability reduction phase.
- A pair of *portal labels*, $l_p^{in}(u)$ and $l_p^{out}(u)$, which are two portals of u in G_r if they exist. They are also assigned by the tree-reachability reduction phase.
- A pair of *remainder labels*, $L_r^{in}(u)$ and $L_r^{out}(u)$, each of which consists of a set of symbols in general. They are assigned by the remainder graph-reachability encoding phase.

Next, we describe the two phases in Sections 4 and 5. In Section 6, we show how to answer reachability queries using these labels. Due to the lack of space, label maintenance due to graph updates is discussed in the technical report [12] and omitted here.

4. TREE-REACHABILITY REDUCTION

our goal in this first phase is to preprocess the graph G to collapse strongly connected components, extract tree structures within the result graph, capture their implied reachability relationships using interval-based labeling, and produce a remainder graph G_r containing the remaining reachability relationships for the next step.

We first identify the strongly connected components (SCC) in G. If two nodes u and v belong to the same SCC, they are indistinguishable from each other in terms of reachability. For any node w, if $w \xrightarrow{*} u$, then $w \xrightarrow{*} v$; similarly, if $u \xrightarrow{*} w$, then $v \xrightarrow{*} w$. Hence, we collapse each SCC of G into one single representative node that retains all edges coming in and out of this SCC. Subsequently, we will only deal with this representative node; nodes strongly connected to it receive it as their SCC label (l_s). All SCCs could be

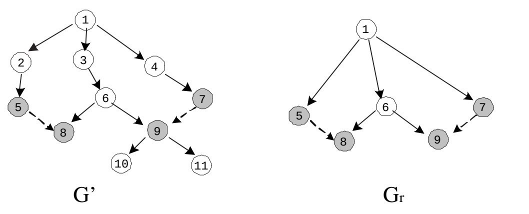


Figure 4: An example illustrating Definition 1.

found in $O(|V|+|E|)$ time by Tarjan's algorithm. By replacing all SCCs with their representative nodes, we obtain a result graph G' with no cycles. The idea of reducing a cyclic graph to its SCCs is considered in [17] as well for computing the 2-hop cover.

Next, we identify a spanning forest $\mathcal{T}$ of G' and assign interval labels $l_i^{\llcorner}(u)$ and $l_i^{\lrcorner}(u)$ to each node u to capture all reachability relationships in $\mathcal{T}$. The identification of $\mathcal{T}$ and assignment of interval labels can be done easily in linear time of $|V_{G'}|$ and omitted here.

Finally, we must capture all remaining reachability relationships in a remainder graph G_r, which is usually much smaller than G'. Nodes of G_r are a subset of nodes in G', which we call *portals*. We label each node u of G' with two portals: an *in-portal* $l_p^{in}(u)$ and an *out-portal* $l_p^{out}(v)$. To support efficient reachability checking, portal labels and G_r must have the following property:

(P1) For any two nodes $u, v \in G'$, $u \xrightarrow{*} v$ iff

- $[l_i^{\llcorner}(u), l_i^{\lrcorner}(u)] \supseteq [l_i^{\llcorner}(v), l_i^{\lrcorner}(v)]$, or
- $l_p^{out}(u) \xrightarrow{*} l_p^{in}(v)$ in G_r.

In other words, unless u can reach v via tree edges in the spanning forest, the only way for u to reach v is by going through the out-portal of u and the in-portal of v. To this end, we define $l_p^{out}(u)$, $l_p^{in}(u)$, and G_r as follows:

DEFINITION 1 (PORTALS AND REMAINDER GRAPH). *Given a spanning forest $\mathcal{T}$ of G', a node $u \in G'$ is* exposed *if there exists an edge (u, v) (or (v, u)) in G' such that u is not v's ancestor (or descendant, respectively) in $\mathcal{T}$.*

- *The* in-portal *of u, $l_p^{in}(u)$, is u's lowest exposed ancestor in $\mathcal{T}$, if any.*
- *The* out-portal *of u, $l_p^{out}(u)$, is the lowest common ancestor of all u's exposed descendants in $\mathcal{T}$, if any.*

The remainder graph *G_r of G' consists of nodes that are in-portals or out-portals of some nodes in G'. There is an edge between two nodes u and v in G_r iff $u \xrightarrow{*} v$ in G'.*

The definition of portals assumes that ancestor and descendant relationships are reflexive, i.e., a node is an ancestor and descendant of itself. Note that an out-portal is not necessarily exposed; a non-exposed node can be an out-portal if it is the lowest common ancestor of some exposed nodes. Figure 4 shows a sample graph G', where solid edges belong to the spanning forest $\mathcal{T}$. Based on the definition, gray nodes are exposed nodes. Node 6 is not exposed, but it is the out-portal of node 3 since it is the least common ancestor of all node 3's exposed descendants (nodes 8 and 9).

The following theorem ensures that the reduction of G' into G_r given by Definition 1 preserves all remaining reachability information. The proof of this theorem is provided in Appendix A of [12].

THEOREM 1. *The portal labels and remainder graph defined by Definition 1 have property (P1).*

Algorithm 4.1, REDUCE, assigns portal labels and constructs the remainder graph in linear time. Basically, REDUCE performs a depth-first traversal on each tree in $\mathcal{T}$. During the traversal, REDUCE maintains a stack of all exposed ancestors to make in-portal

Algorithm 4.1 Assign portal labels and construct remainder graph.

```
REDUCE($G'$, $\mathcal{T}$)
{$\mathcal{T}$: spanning forest of $G'$ for interval labels}
 1: $(V_r, E_r) \leftarrow (\varnothing, \varnothing)$;
 2: for each root node $t$ in $\mathcal{T}$ do
 3:     REDUCEHELPER($t, \varnothing$);
 4: return $(V_r, E_r)$;

REDUCEHELPER($u, S$)
{$S$: a stack with $u$'s exposed ancestors in $\mathcal{T}$}
 1: if $u$ is exposed then
 2:     PUSH($S, u$);
 3:     $l_p^{out}(u) \leftarrow u$;
 4: $l_p^{in}(u) \leftarrow$ TOP($S$) if $S \neq \varnothing$;
 5: for each child $v$ of $u$ in $\mathcal{T}$ do
 6:     REDUCEHELPER($v, S$);
 7:     if $l_p^{out}(v)$ has been assigned then { incrementally update $l_p^{out}(u)$ to
        be the lowest common ancestor}
 8:         if $l_p^{out}(u)$ has not been assigned then
 9:             $l_p^{out}(u) \leftarrow l_p^{out}(v)$;
10:         else
11:             $l_p^{out}(u) \leftarrow u$;
12: if $u =$ TOP($S$) then
13:     POP($S$);
14: if $u = l_p^{out}(u)$ then { add $u$ and its incoming edges to $G_r$}
15:     $V_r \leftarrow V_r \cup \{u\}$;
16:     $E_r \leftarrow E_r \cup \{($TOP$(S), u)\}$ if $S \neq \varnothing$;
17:     for each edge $(w, u) \in G'$ where $w$ is not $u$'s ancestor in $\mathcal{T}$ do
18:         $V_r \leftarrow V_r \cup \{w\}$; $E_r \leftarrow E_r \cup \{(w, u)\}$;
19: return;
```

label assignments. Out-portal labels are computed bottom-up. Portals and their incoming edges are added to the remainder graph as they are identified. In fact, we can augment REDUCE so that it can also find the spanning forest $\mathcal{T}$ and assign interval labels in the same pass over G'. For clarity, however, we only present here how to assign portal labels and construct the remainder graph given $\mathcal{T}$.

Finally, we note in the following theorem that the size of the remainder graph is linear in the number of "non-tree" edges, i.e., those that are not in $\mathcal{T}$ or implied by $\mathcal{T}$. The proof of Theorem 2 is provided in [12]. Therefore, in practice, the remainder graph can be much smaller than the original graph. In particular, if the input graph is a tree or forest, the remainder graph would be empty and all portal labels would be unassigned, and our scheme basically degenerates into the interval-based approach.

THEOREM 2. *Let E_{nt} denote the set of edges in G' that are not from a node to its descendant in $\mathcal{T}$. The remainder graph G_r has fewer than $4|E_{nt}|$ nodes and $5|E_{nt}|$ edges.*

5. REMAINDER GRAPH-REACHABILITY ENCODING

After extracting reachability relationships in the spanning forest, we now turn to the problem of encoding remaining reachability relationships in the remainder graph G_r. As outlined in Section 3, our goal in this phase is to assign the remainder labels $L_r^{in}(u)$ and $L_r^{out}(u)$ for each node u in G_r to help checking reachability among nodes in G_r. Let $\mathbf{T}_r$ denote the transitive closure matrix of G_r. The general idea is to compress the content of $\mathbf{T}_r$ into remainder labels, in a way that allows any $\mathbf{T}_r$ entry to be recovered efficiently.

While many compression algorithms can be applied to $\mathbf{T}_r$ (e.g., Blocked Huffman coding or LZW), most of them do not support efficient recovery of individual entries. Our approach is to identify a dense submatrix of $\mathbf{T}_r$ with mostly ones, and encode it compactly in remainder labels of the nodes associated with rows and columns of the submatrix. This process is repeated until unencoded ones in

Algorithm 5.1

```
FINDDSM_2APPROX($\mathbf{A}$)
 1: $(R, C) \leftarrow$ (rows of $\mathbf{A}$, columns of $\mathbf{A}$); $d_{\max} \leftarrow -1$;
 2: while $R \neq \varnothing$ and $C \neq \varnothing$ do
 3:    if $density(\mathbf{A}(R, C)) > d_{\max}$ then
 4:       $d_{\max} \leftarrow density(\mathbf{A}(R, C))$;
 5:       $(R_{\max}, C_{\max}) \leftarrow (R, C)$;
 6:    $r \leftarrow \arg\min_{r \in R} n_1(\mathbf{A}(\{r\}, C))$;
 7:    $c \leftarrow \arg\min_{c \in C} n_1(\mathbf{A}(R, \{c\}))$;
 8:    if $n_1(\mathbf{A}(\{r\}, C)) < n_1(\mathbf{A}(R, \{c\}))$ or
       $(n_1(\mathbf{A}(\{r\}, C)) = n_1(\mathbf{A}(R, \{c\}))$ and $|R| > |C|)$ then
 9:       $R \leftarrow R - \{r\}$;
10:    else
11:       $C \leftarrow C - \{c\}$;
12: return $(R_{\max}, C_{\max})$;
```

$\mathbf{T}_r$ are sparse enough to be stored efficiently in a sparse matrix. In Section 5.1, we describe in detail how to encode a dense submatrix with remainder labels, and how to quantify the quality of encoding in term of *encoding density* of the submatrix. In Section 5.2, we first show the hardness of finding the submatrix with the highest encoding density, and then present several algorithms that attempt to find such submatrices. The overall algorithm for encoding the entire $\mathbf{T}_r$ matrix will be given in Section 5.3.

5.1 Encoding a Dense Submatrix

Suppose R and C are two non-empty sets of nodes in the remainder graph G_r. Let $\mathbf{T}_r(R, C)$ denote the *submatrix of* $\mathbf{T}_r$ *spanned by* R *and* C, i.e., the submatrix whose rows and columns correspond to nodes in R and C, respectively. This submatrix captures reachability relationships from R nodes to C nodes. To encode these reachability relationships, we pick a unique symbol s; then add s to $L_r^{out}(u)$ for each node $u \in R$ and to $L_r^{in}(v)$ for each node $v \in C$. In addition, for each entry (u, v) of T_r with value 0, we add the pair (u, v) to a *zero-exception set* $\mathcal{E}_0$. Clearly, $u \xrightarrow{*} v$ if

$$L_r^{out}(u) \cap L_r^{in}(v) \neq \varnothing \text{ and } (u, v) \notin \mathcal{E}_0.$$

Intuitively, adding a common symbol to $|R| + |C|$ remainder labels has the effect of remembering $\mathbf{T}_r(R, C)$ as a submatrix of all ones. Any zero in it needs to be remembered in $\mathcal{E}_0$ as an exception. Thus, we no longer need to store entries of $\mathbf{T}_r(R, C)$ in $\mathbf{T}_r$.

The amount of space used in remainder labels to encode $\mathbf{T}_r(R, C)$ is $(|R| + |C|) \times size(s)$, where $size(s)$ denotes the size of symbol s in bits; in addition, the amount of space used in the zero-exception set is $n_0(\mathbf{T}_r(R, C)) \times size(e_0)$, where $n_x(\cdot)$ counts the number of entries with value x in a matrix, and $size(e_0)$ denotes the size of an entry in $\mathcal{E}_0$. We can quantify the quality of encoding by the *encoding density of the submatrix* $\mathbf{T}_r(R, C)$, defined as the ratio between the number of ones in the submatrix and the amount of space used in encoding the submatrix:

$$density(\mathbf{T}_r(R, C)) = \frac{n_1(\mathbf{T}_r(R, C))}{(|R| + |C|) \times size(s) + n_0(\mathbf{T}_r(R, C)) \times size(e_0)}. \quad (1)$$

The higher the encoding density of a submatrix—or in short, the *denser* the submatrix—the better it is to apply our encoding. The overall remainder graph-reachability encoding algorithm, to be presented in Section 5.3, greedily identifies a dense (if not the densest) submatrix of $\mathbf{T}_r$, encodes it, marks its entries as encoded (using a value other than one or zero), and repeats the process. Thus, in the general case that parts of $\mathbf{T}_r$ have already been encoded, Equation (1) defines the encoding density to be the ratio between the number of unencoded ones and the amount of additional space used in encoding (zeros covered by previously encoded parts are already remembered in $\mathcal{E}_0$ and thus do not require additional space).

The encoding density in the 2-hop is essentially a restricted case of our definition. As discussed in Section 2, the 2-hop approach only considers submatrices induced by single nodes. The submatrix of $\mathbf{T}_r$ induced by node u is $\mathbf{T}_r(C_{in}(u), C_{out}(u))$, where, recall from Section 2, $C_{in}(u)$ is the set of nodes that can reach u, and $C_{out}(u)$ is the set of nodes that u can reach. Note that all entries in this submatrix are ones, so $n_1(\mathbf{T}_r(C_{in}(u), C_{out}(u))) = |C_{in}(u)| \times |C_{out}(u)|$ and $n_0(\mathbf{T}_r(C_{in}(u), C_{out}(u))) = 0$. Hence, our definition of encoding density reduces (up to a constant factor) to the one used by the 2-hop: $\frac{|C_{in}(u)| \times |C_{out}(u)| - k}{|C_{in}(u)| + |C_{out}(u)|}$, where k is the number of ones in the submatrix that have been encoded.

5.2 Finding a Dense Submatrix

A critical step of our remainder graph-reachability encoding algorithm involves identifying a submatrix of $\mathbf{T}_r$ to encode, preferably the densest one. Before we present the algorithms for finding such submatrices, we give a formal definition of the general problem and show its hardness.

DEFINITION 2 (DENSEST SUBMATRIX PROBLEM). *The* densest submatrix problem *(*DSM*) is defined as follows: Given a binary matrix* $\mathbf{A}$ *and non-negative parameters* $size(s)$ *and* $size(e_0)$*, find a subset* R *of rows and a subset* C *of columns from* $\mathbf{A}$ *that maximize* $density(\mathbf{A}(R, C))$*, the encoding density of the submatrix spanned by* R *and* C.

THEOREM 3. *Under the plausible assumption that* 3*-SAT does not have a sub-exponential time algorithm, DSM is hard to approximate within a factor of* $2^{(\log n)^{\delta} - 1}$ *for some* $\delta > 0$*. Here,* n *denotes the total number of rows and columns in the input matrix to the DSM problem.*

The proof of Theorem 3 is provided in Appendix C of [12]. This result implies that, for the general DSM problem, it may be fruitless to go after the optimal solution. Thus, we turn to heuristics or algorithms that consider a restricted solution space or special instances of the DSM problem for which efficient approximation is possible. The rest of this section covers two such algorithms. We also propose another SVD-based algorithm in [12], but it has higher complexity and therefore is omitted here.

5.2.1 A 2-Approximation Algorithm

The first algorithm, FINDDSM_2APPROX (Algorithm 5.1), is greedy. To obtain a dense submatrix of $\mathbf{A}$, it simply keeps removing the row or column with the least number of ones from $\mathbf{A}$, one at a time. This process produces a sequence of submatrices as intermediate results. The densest submatrix among them is chosen.

FINDDSM_2APPROX can be made an efficient $O(n^2)$ algorithm, where n denotes the total number of rows and columns in $\mathbf{A}$. The main loop runs at most n times. Without any optimization, each iteration of the loop would take $O(n^2)$ time, most of which is spent on counting ones. However, for each row (or column) to be removed, we can remember the count of its ones, and update this count whenever a column (or row, respectively) is removed. This maintenance only takes $O(n)$ time per iteration, and the cost of finding the row or column with the least number of ones is reduced to $O(n)$ per iteration. Thus, we have reduced the overall complexity of FINDDSM_2APPROX to $O(n^2)$. For simplicity of presentation, we do not show this optimization in Algorithm 5.1.

Additional good news is that, for the instance of DSM with $size(e_0) = 0$, FINDDSM_2APPROX turns out to be a 2-approximation algorithm, i.e., it returns a submatrix whose encoding density is in a factor of two of the densest one. Note that this instance of DSM is not at all unreasonable: Setting $size(e_0) = 0$ in Equa-

Algorithm 5.2

```
FINDDSM_EXT2HOP(A)
1:  $d_{\max} \leftarrow -1$;
2:  for $u \in G_r$ do {first use 2-hop to find a dense submatrix}
3:      if $density(\mathbf{A}(C_{in}(u), C_{out}(u))) > d_{\max}$ then
4:          $d_{\max} \leftarrow density(\mathbf{A}(C_{in}(u), C_{out}(u)))$;
5:          $(R, C) \leftarrow (C_{in}(u), C_{out}(u))$;
6:  $(R_{all}, C_{all}) \leftarrow$ (rows of $\mathbf{A}$, columns of $\mathbf{A}$);
7:  while $(R, C) \neq (R_{all}, C_{all})$ do {further extend submatrix}
8:      $r \leftarrow \arg\max_{r \in R_{all} - R} n_1(\mathbf{A}(\{r\}, C))$ if $R \neq R_{all}$;
9:      $c \leftarrow \arg\max_{c \in C_{all} - C} n_1(\mathbf{A}(R, \{c\}))$ if $C \neq C_{all}$;
10:     if $C = C_{all}$ or $R \neq R_{all}$ and
        $density(\mathbf{A}(R \cup \{r\}, C)) > density(\mathbf{A}(R, C \cup \{c\}))$ then
11:         $(R', C') \leftarrow (R \cup \{r\}, C)$;
12:     else
13:         $(R', C') \leftarrow (R, C \cup \{c\})$;
14:     if $density(\mathbf{A}(R', C')) \geq density(\mathbf{A}(R, C))$ then
15:         $(R, C) \leftarrow (R', C')$;
16:     else
17:         break;
18: return $(R, C)$;
```

tion (1) does not imply that the cost of zero-exception list is ignored, because the numerator, $n_1(\mathbf{A}(R, C))$, still favors submatrices with more ones. Theorem 4 is proven in Appendix D of [12].

THEOREM 4. FINDDSM_2APPROX *is a* 2*-approximation algorithm for finding the densest submatrix* $\mathbf{A}(R, C)$ *with encoding density defined as* $density_2(\mathbf{A}(R, C)) = \frac{n_1(\mathbf{A}(R,C))}{|R|+|C|}$.

5.2.2 An Enhanced 2-Hop Algorithm

Recall that the 2-hop approach considers only submatrices induced by single nodes, and picks the densest submatrix among them. The approach does not consider any submatrix containing zeros, or any submatrix corresponding to a bipartite subgraph such as the one illustrated in Figure 2. Our second algorithm, FINDDSM _EXT2HOP (Algorithm 5.2), removes these limitations by further extending the submatrix found by the 2-hop approach with additional rows and columns as long as they increase density. The resulting submatrix may contain zeros, and its ones may correspond to paths that do not go through a common node. In sum, FINDDSM_EXT2HOP considers a larger solution space than the 2-hop approach, while using the 2-hop solution to seed the search.

In FINDDSM_EXT2HOP, the 2-hop loop requires only $O(n^2)$ time, because density calculation is simplified by the fact that all submatrices considered by the 2-hop approach contain no zeros. The second loop for extending the result submatrix runs at most n times, and each iteration can be optimized to run in $O(n)$ time using two optimizations that we have applied earlier to the previous algorithm. First, for each remaining row (or column), we can record the count of ones that it can add to the current submatrix, and update this count whenever a column (or row, respectively) is added; this optimization brings down the cost of finding the row and column with the most ones to $O(n)$. Second, we can record the numbers of ones and zeros in the submatrix and update the two counts in each iteration; this optimization brings down the cost of density calculation to $O(n)$. Overall, the complexity of FINDDSM_EXT2HOP becomes $O(n^2)$.

5.3 Overall Algorithm for Remainder Graph

We now present ENCODE (Algorithm 5.3), the overall algorithm for encoding remainder graph reachability. The input is the remainder graph G_r. The algorithm first computes the transitive closure matrix $\mathbf{T}_r$ for G_r. In each iteration of its main loop, the algorithm greedily identifies a dense submatrix of $\mathbf{T}_r$ using one of the two algorithms in Section 5.2. The content of this submatrix is encoded in remainder labels and the zero-exception set $\mathcal{E}_0$, using the procedure described in Section 5.1. Next, the algorithm marks the entries of the submatrix as already encoded, by setting their values to 0.001. As the loop continues, unencoded ones become fewer and sparser, submatrix densities become less, and dense submatrices become smaller. When eventually dimensions of candidate submatrices shrink to 1×1, we terminate the loop and remember all unencoded ones in a *one-exception set* $\mathcal{E}_1$. Both $\mathcal{E}_0$ and $\mathcal{E}_1$ can be implemented using hash tables.

Algorithm 5.3 Assign remainder labels.

```
ENCODE($G_r$)
1:  $\mathbf{T}_r \leftarrow$ the transitive closure matrix of $G_r$;
2:  for each entry $\mathbf{T}_r(u, v)$ of $\mathbf{T}_r$ with value one do
3:      if $[l_i^{\llcorner}(u), l_i^{\lrcorner}(u)] \supseteq [l_i^{\llcorner}(v), l_i^{\lrcorner}(v)]$ then
4:          $\mathbf{T}_r(u, v) \leftarrow 0.001$; {ignore ones implied by interval labels}
5:  loop
6:      $(R, C) \leftarrow$ FINDDSM($\mathbf{T}_r$); {any algorithm in Section 5.2}
7:      if $|R| \times |C| \leq 1$ or $density(\mathbf{T}_r(R, C)) = 0$ then
8:          break;
        {update remainder labels}
9:      $s \leftarrow$ a new unique symbol;
10:     for each $u \in R$ do
11:         $L_r^{out}(u) \leftarrow L_r^{out}(u) \cup \{s\}$;
12:     for each $v \in C$ do
13:         $L_r^{in}(v) \leftarrow L_r^{in}(v) \cup \{s\}$;
        {remember unencoded zeros, and mark all entries as encoded}
14:     for each entry $\mathbf{T}_r(u, v)$ in $\mathbf{T}_r(R, C)$ do
15:         if $\mathbf{T}_r(u, v) = 0$ then
16:             $\mathcal{E}_0 \leftarrow \mathcal{E}_0 \cup \{(u, v)\}$;
17:         $\mathbf{T}_r(u, v) \leftarrow 0.001$;
    {remember all unencoded ones}
18: for each entry $\mathbf{T}_r(u, v)$ of $\mathbf{T}_r$ with value one do
19:     $\mathcal{E}_1 \leftarrow \mathcal{E}_1 \cup \{(u, v)\}$;
20: return $(\mathcal{E}_0, \mathcal{E}_1)$;
```

Let m denote the number of nodes in G_r (i.e., the portals). The running time of ENCODE is $O(m^3)$. Computation of the transitive closure matrix can be done easily in $O(m^3)$, using a simplified version of the Floyd-Warshall algorithm. We can control the main loop so that it executes at most $O(m)$ times. First, note that FINDDSM_EXT2HOP can be run $O(m)$ times before triggering the break condition of the main loop, because each run uses up one hop node. For FINDDSM_2APPROX, we can run them $O(m)$ times and then simply switch to running FINDDSM_EXT2HOP subsequently, which results in at most $O(m)$ iterations of the main loop overall. The cost of each iteration is dominated by finding a dense submatrix, which takes $O(m^2)$ time as discussed in Section 5.2.

6. QUERY ALGORITHM

Given two nodes u and v, testing whether $u \xrightarrow{*} v$ is straightforward. If either one has a strongly connected component label, the node in the label is checked instead. Then check their interval labels. If the answer is not affirmative, we look up their portal labels and check reachability between u's out-portal and v's in-portal, which involves testing whether their remainder labels intersect, and whether the pair belong to zero- and one-exception sets (implemented as hash tables). All steps take constant time except testing intersection of two remainder labels, which takes time linear to the lengths of these labels. The algorithm is provided below.

7. EXPERIMENTS

In this section, we demonstrate the effectiveness of HLSS and compare it with other approaches using experiments on a variety of graphs. We have implemented HLSS as well as two other labeling schemes for comparison: the interval-based approach [1] and the 2-hop approach [9]. One primary metric for measuring the ef-

Algorithm 6.1

```
IsReachable$(u, v)$
 1: $u \leftarrow l_s(u)$ if $l_s(u)$ exists; $v \leftarrow l_s(v)$ if $l_s(v)$ exists;
 2: if $[l_i^{\llcorner}(u), l_i^{\lrcorner}(u)] \supseteq [l_i^{\llcorner}(v), l_i^{\lrcorner}(v)]$ then
 3:     return true;
 4: $(u_p, v_p) \leftarrow (l_p^{out}(u), l_p^{in}(v))$;
 5: if $[l_i^{\llcorner}(u_p), l_i^{\lrcorner}(u_p)] \supseteq [l_i^{\llcorner}(v_p), l_i^{\lrcorner}(v_p)]$ then
 6:     return true;
 7: if $(u_p, v_p) \in \mathcal{E}_1$ then
 8:     return true;
 9: if $(u_p, v_p) \in \mathcal{E}_0$ then
10:     return false;
11: if $L_r^{out}(u_p) \cap L_r^{in}(v_p) \neq \varnothing$ then
12:     return true;
13: return false;
```

fectiveness of a labeling scheme is its *compression factor*, which is defined as the total number of entries in the transitive closure matrix (i.e., the number of bits needed to store this matrix in dense format) divided by the number of bits required by the labeling scheme. For HLSS, we count the space used by labels as well as the zero- and one-exception sets. The larger the compression factor, the greater the space saving. For all three schemes we are comparing, the time to answer a reachability query is linear in the size of labels. Therefore, compression factor is a rough indicator of average query performance. To get an more accurate picture of overall query performance, we also measure the distribution of individual label sizes.

Random Graphs with Varying Densities For the first set of experiments, we generate "random graphs" as follows. We fix the number of nodes at 1000 and create each edge by choosing its source and sink uniformly at random. We vary the number of edges from 1000 to 3000; as we will see, this range is large enough to capture the trends in results. Figure 5 shows the compression factors of three labeling schemes as we vary the *graph density*, i.e., the ratio between the number of edges and the number of nodes.

At low graph densities, compression factors tend to fluctuate because reachability relationships heavily depends on the randomly generated topologies; the difference among the three scheme is small. However, a clear trend emerges as we increase the graph density. Even though the graphs remain sparse (the densest one has on average only 3 edges per node), as the number of edges increases, the fraction of nodes in SCCs increases quickly. All labeling schemes benefit from this increase. For the 2-hop approach, there are more and better hop nodes. For the interval-based approach (with our extension for handling SCCs), the original graph is reduced to a much smaller DAG. As shown in Figure 5, the compression factors of both schemes quickly approach a common limit. This limit can be obtained analytically based on the observation that, for both schemes, each label must have at least two components: in-label and out-label for the 2-hop approach, and two interval endpoints for the interval-based approach. This limit works out to be 50 for a graph of 1000 nodes (detailed calculation omitted), which is confirmed by Figure 5. HLSS, on the other hand, uses only a single representative node to label nodes in the same SCC. Therefore, HLSS compression factor can grow beyond this limit.

Synthetic Graphs with Varying Parameters As shown in the previous set of experiments, the dominating factor in the performance of labeling schemes on a random graph with moderate to high density turns out to be how well they handle SCCs. To expose other factors affecting performance, we need finer control over the properties of synthetic graphs. To this end, we generate graphs with three adjustable parameters: p_s, the percentage of nodes in SCCs; p_u, the percentage of unexposed nodes (recall Definition 1) in the remaining DAG (after collapsing SCCs); and d_o, the average out-degree of exposed nodes in the remaining DAG. For lack of space, we omit the details of how to construct a graph with these parameters. We have experimented graphs with various numbers of nodes and found relative performance of the three schemes to be consistent. So we only present results for graphs with 2000 nodes.

Figure 6 shows the compression factors of the three schemes as we vary p_s (percentage of nodes in SCCs) from 0 to 0.6. We fix $p_u = 0.4$ and $d_o = 3$. From the figure, we see that HLSS outperforms the other two approaches by a big margin because of its efficiency in handling SCCs. The interval-based approach performs worst because the remaining DAG is complex enough (with an average out-degree of 3) that the label of a node will be propagated to a large number of nodes that can reach it. As p_s increases, all three schemes become more effective, with HLSS widening its lead for reasons discussed earlier under the results for random graphs.

Next, Figure 7 compares the three schemes as we vary d_o (the average out-degree of exposed nodes in the remaining DAG) from 1 to 100. We fix $p_s = 0.2$ and $p_u = 0.4$. When $d_o = 1$, the interval-based approach works best as expected, because the DAG is close to a tree, for which the interval-based approach is optimized. HLSS is also quite effective for $d_o = 1$, but the 2-hop approach is not nearly as good as the others. As d_o increases (from 2 to 10), however, the interval-based approach see the most dramatic drop in compression factor, because the DAG quickly increases in complexity and many intervals end up being propagated to many nodes. The 2-hop approach handles complex DAGs much more gracefully, eventually outperforming the interval-based approach; HLSS handles complex DAGs just as gracefully as the 2-hop approach, and maintains the lead over the 2-hop approach. As d_o continues to grow beyond 10, the DAG becomes increasingly connected, and its transitive closure matrix grows denser. Hence, all compression factors begin to increase again. The interval-base approach manages to pick up the largest gain because of increasing opportunities in applying the following optimization: If two intervals touch each other, they can be merged into one bigger interval. Overall, HLSS is superior to the other approaches almost all the time except the case of $d_o = 1$, where the interval approach really shines and HLSS ranks as a close second.

Finally, Figure 8 shows the results of varying p_u (the percentage of unexposed nodes in the remaining DAG) from 0.3 to 0.9, while fixing p_s at 0.2. Here, since adjusting p_u changes the number of exposed nodes, keeping d_o constant would make topologies of the resulting DAGs very different for different p_u. To focus on studying the effect of p_u, we fix an alternative parameter, *edge density*, to 0.01; edge density is defined as the ratio between the number of edges in the generated DAG and the maximum number of edges possible for a DAG of this size. When p_u is small, the graph has little resemblance to a tree, because only a small number of nodes are connected only through tree edges. Therefore, the interval-based approach suffers and performs much worse than the other two. As p_u increases, the graph looks more like a tree, and the interval-based approach performs much better. Again, HLSS is almost always the best except the extreme case where HLSS is a little behind the interval-based approach for a graph with mostly tree edges.

In conclusion, we see that HLSS consistently delivers performance better than or comparable to the the other two approaches, while they each have their respective strengths and weaknesses for different types of graphs. The interval-based approach is able to outperform HLSS by a small margin for tree-like graphs. However, the compression factor is so high for such graphs that the actual reduction in space is quite small (e.g., $\frac{1}{85}$ vs. $\frac{1}{75}$ for $d_o = 1$ in Figure 7). But HLSS provides much more substantial space savings for graphs with low compression factors ($\frac{1}{16}$ vs. $\frac{1}{4}$ for $d_o = 10$).

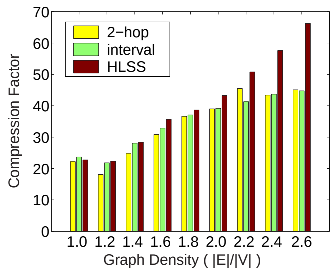


Figure 5: Random graphs.

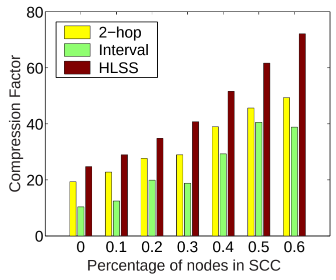


Figure 6: Varying p_s ($p_u = 0.4$, $d_o = 3$).

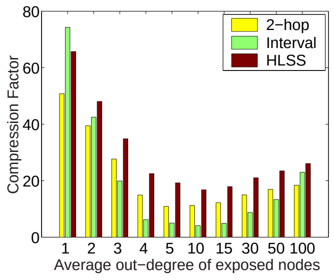


Figure 7: Varying d_o ($p_u = 0.4$, $p_s = 0.2$).

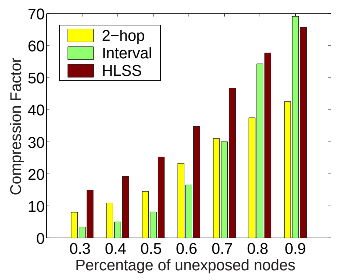
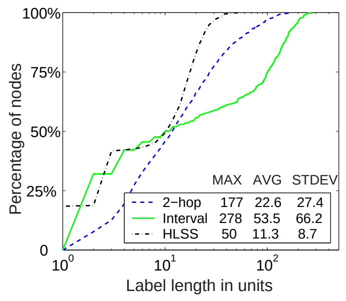


Figure 8: Varying p_u ($p_s = 0.2$, edge density 0.01).



	MAX	AVG	STDEV
2−hop	177	22.6	27.4
Interval	278	53.5	66.2
HLSS	50	11.3	8.7

Figure 9: Label size distributions.

Label Size Distribution Because of space constraint, we present the distributions of individual label sizes only for one graph with 2000 nodes, $p_s = 0.2$, $p_u = 0.3$, and $d_o = 5$. In Figure 9, the logarithmic horizontal axis shows the label size in terms of number of *units*, where each unit represents the amount of space required to store a node identifier, a symbol, or an interval endpoint. The vertical axis shows the percentage of labels whose size is less than or equal to the given size. The legend also shows the maximum label size, the average label size, and the standard deviation for each scheme. The interval-based approach has the widest range of possible label sizes; it has a large percentage of short labels (around 40% have length no more than 4 units), but unfortunately also a significant percentage of long labels (around 25% have length greater than 100 units), which would be quite expensive when queried. In contrast, the 2-hop approach has mostly medium-sized labels. Interestingly, HLSS inherits the good features from other two approaches and does much better than both of them. There are roughly $20\% \approx p_s$ of the labels with unit size; they correspond to node in SCCs. Close to $24\% \approx (1 - p_s)p_u$ of the labels have length 3; they correspond to non-portal nodes (recall Section 4). In this region of the graph, HLSS behaves much like the interval-based approach. The remaining part of the HLSS curve resembles the curve of the 2-hop approach, in that fewer and fewer labels have longer labels. The maximum label length is 50, significantly less than the other two. Overall, HLSS demonstrates the most favorable label size distribution among the three approaches.

8. CONCLUSION

In this paper, we have revisited the important problem of labeling graph reachability. Many labeling schemes have been proposed in the past, but most of them are optimized to exploit particular types of substructures in graphs and do not work well on other substructures. We have proposed a hierarchical approach that combines the strengths of existing approaches by labeling different types of substructures differently. Our two-phase algorithm is an efficient realization of this hierarchical approach; experiments have shown that it handles different types of graphs well, while existing approaches suffer from substructures that they do not handle well.

9. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proc. of the 1989 ACM SIGMOD Intl. Conf. on Management of Data*, 1989.

[2] H. Ait-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Trans. Program. Lang. Syst.*, 11(1):115–146, 1989.

[3] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. of the 2002 Intl. Conf. on Data Engineering*, pages 141–154, 2002.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proc. of the 2002 ACM SIGMOD*, 2002.

[5] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *Proc. of the 1993 Conf. on Object-oriented Programming Systems, Languages, and Applications*, 1993.

[6] Online Computer Library Center. Dewey decimal classification. http://www.oclc.org/dewey/.

[7] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003.

[8] V. Christophides, D. Plexousakis, and et al. On labeling schemes for the semantic web. In *Proc. of the 12th Intl. Conf. on WWW*, 2003.

[9] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proc. of the 13th ACM-SIAM SODA*, 2002.

[10] P. F. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th ACM symposium on Theory of computing*, 1982.

[11] Frank Olken et al. The Biopathways Graph Data Manager project. http://pueblo.lbl.gov/˜olken/graphdm/ graphdm.htm.

[12] H. He, H. Wang, J. Yang, and Philip S. Yu. Compact reachability labeling for graph-structured data. Technical report, Duke University, Nov. 2004. http://www.cs.duke.edu/˜haohe/research/report.pdf.

[13] T. Kameda. On the vector representation of the reachability in planar directed graphs. *Information Processing Letters*, 3(3), January 1975.

[14] M. Katz, N. A. Katz, and et al. Labeling schemes for flow and connectivity. In *Proc. of the 13th ACM-SIAM SODA*, 2002.

[15] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, June 2002.

[16] N. Polyzotis and M. N. Garofalakis. Structure and value synopses for XML data graphs. In *Proc. of the 28th VLDB*, pages 466–477, 2002.

[17] A. Sayed and R. Unland. Hid: An efficient path index for complex xml collections with arbitrary links. In *DNIS*, pages 78–91, 2005.

[18] SRI. The BioCyc project. http://biocyc.org/.

[19] Z. Vagena, M. Moro, and V. Tsotras. Twig query processing over graph-structured XML data. In *Proc. of the 7th Intl. Workshop on Web and Databases*, 2004.

[20] C. Zhang, J. Naughton, and et al. On supporting containment queries in relational database management systems. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, 2001.