Dual Labeling: Answering Graph Reachability Queries in Constant Time

Haixun Wang[1] Hao He[2] Jun Yang[2] Philip S. Yu[1] Jeffrey Xu Yu[3]
[1]IBM T. J. Watson Research Center, {haixun,psyu}@us.ibm.com
[2]Duke University, {haohe,junyang}@cs.duke.edu
[3]The Chinese University of Hong Kong, yu@se.cuhk.edu.hk

Abstract

Graph reachability is fundamental to a wide range of applications, including XML indexing, geographic navigation, Internet routing, ontology queries based on RDF/OWL, etc. Many applications involve huge graphs and require fast answering of reachability queries. Several reachability labeling methods have been proposed for this purpose. They assign labels to the vertices, such that the reachability between any two vertices may be decided using their labels only. For sparse graphs, 2-hop based reachability labeling schemes answer reachability queries efficiently using relatively small label space. However, the labeling process itself is often too time consuming to be practical for large graphs. In this paper, we propose a novel labeling scheme for sparse graphs. Our scheme ensures that graph reachability queries can be answered in constant time. Furthermore, for sparse graphs, the complexity of the labeling process is almost linear, which makes our algorithm applicable to massive datasets. Analytical and experimental results show that our approach is much more efficient than state-of-the-art approaches. Furthermore, our labeling method also provides an alternative scheme to tradeoff query time for label space, which further benefits applications that use tree-like graphs.

1 Introduction

Given two vertices **u** and **v** in a directed graph, we want to know if there is a path from **u** to **v**. The problem is known as graph reachability, and has been well-explored in several fields of computation [3, 8, 2, 19, 17]. In many applications (e.g., XML query processing), graph reachability is one of the most basic operations, which means fast processing is mandatory. A naïve approach to this problem is to precompute the reachability between every pair of vertices – in other words, to compute and store the transitive closure of the graph, so that we can answer reachability queries in constant time. However, this requires $O(n^2)$ storage, which makes it impractical for massive graphs.

Recently, Cohen et al. [8] showed that, for sparse graphs, a sophisticated graph labeling method called 2-hop can answer reachability queries efficiently (although not in constant time) using much less storage. This result is important because most real life massive graphs are sparse. However, 2-hop labeling itself may incur a tremendous amount of computation cost. In fact, according to Schenkel et al. [20], it takes a 64-processor, 80-Gb memory Sun server more than 45 hours to label the well-known DBLP dataset [14] using Cohen's algorithm [8]. Clearly, in practice, such labeling methods cannot be used for massive graphs. To resolve this, we introduce a novel labeling algorithm for sparse graphs. Our method labels a sparse graph in almost linear time, and answers reachability queries in constant time.

1.1 Applications

Answering reachability queries efficiently is important to applications in many areas ranging from XML query processing to genome biology. Recently, the interest in graph reachability is rekindled by research on XML data processing. XML documents are often represented by tree structures. Nevertheless, an XML document may contain IDREF/ID reference links that turn itself into a directed graph. One of the most important research topics on XML is efficient evaluation of structural queries. For example, consider a simple path expression `//fiction//author` that queries authors who write fictions. A typical way of processing this query is to obtain (possibly through some index on elements) all `fiction` and `author` elements, and then test if an `author` element is reachable from any `fiction` element in the XML graph. Reachability testing thus becomes a fundamental operation in XML query processing. It is clear to see that efficient support for reachability testing is crucial to XML query processing because it is invoked heavily for answering complex queries on large datasets.

Reachability labeling schemes such as 2-hop focus on sparse graphs as graphs in most real life applications are sparse [8]. For instance, XML documents are mostly tree structures plus a few reference links. The XMark dataset [1], which is designed to model e-commerce applications and has since become a popular benchmark for evaluating the performance of XML query processing, is sparse — the density (edge/vertex ratio) of an XMark document is about 1.15. In biology, graph and network models of all kinds have been used on research topics such as gene-regulatory networks or metabolic networks. Graph reachability models such relationships as whether two genes interact with each other or whether two proteins participate in a common pathway. Many such graphs are sparse. For example, HumanCyc [18], a bioinformatics database that describes human metabolic pathways and human genome, has 40,051 vertices and 43,879 edges, yielding an edge/vertex ratio of 1.096. Eco_O157Cyc, a dataset in the EcoCyc database [13] that consists of annotated genome sequences of Escherichia coli, has 13,800 vertices and 17,308 edges, yielding an edge/vertex ratio of 1.25.

1.2 Challenges

Given an n-vertex, m-edge directed graph, we have two naïve approaches to handle reachability queries. One is to use the single source shortest path algorithm, that is, for any two vertices, we use the shortest path algorithm to determine if they are connected. This approach may take $O(m)$ query time, but requires no extra data structure besides the graph itself for answering reachability queries. The other extreme is to compute and store the transitive closure of the graph. It answers reachability query in constant time but need $O(n^2)$ space to store the transitive closure of an n-vertex graph. Many applications involve massive graphs, yet require fast answering of reachability queries. This makes the naïve approaches infeasible.

Several approaches have been proposed to encode graph reachability information using vertex labeling schemes [3, 8, 19, 20]. A labeling scheme assigns labels to vertices in the graph, and it answers reachability queries by comparing the labels of the vertices. Interval-based labeling is best for tree structures. For graphs, however, reachability queries may take $O(m)$ time using the interval-based approach. Cohen et al. [8] proposed the 2-hop labeling scheme so that for sparse graphs reachability queries can be answered efficiently using relatively less storage. However, it has been shown that there exist graphs for which any reachability labeling is of size $O(nm^{1/2})$, which yields to $O(n^2)$ in the worst case. Correspondingly, each 2-hop label has average length $O(m^{1/2})$, which means answering reachability queries requires $O(m^{1/2})$ comparisons.

Furthermore, labeling can be a time costly process. For instance, finding optimal 2-hop labels is NP-hard. Using approximation algorithms, Cohen et al. [8] reduced the complexity to $O(n^4)$, and later, the HOPI algorithm proposed by Schenkel et al. [19, 20] reduced it to $O(n^3)$. But it is still impractical for massive graphs.

1.3 Our Approach and Contributions

We propose a novel method called dual labeling to handle reachability queries for massive, sparse graphs. The goal is to optimize both query time and labeling time. Our method consists of two schemes, Dual-I and Dual-II. The Dual-I labeling scheme has constant query time, and for sparse graphs, the labeling complexities of both Dual-I and Dual-II are almost linear. The Dual-II scheme has higher query complexity but uses less space in practice. Table 1 compares our dual labeling approaches with existing approaches.

	Query time	Index time	Index size
Shortest Path	$O(m)$	0	0
Transitive Closure	$O(1)$	$O(n^3)$	$O(n^2)$
Interval	$O(n)$	$O(n)$	$O(n^2)$
2-Hop	$O(m^{1/2})$	$O(n^4)$	$O(nm^{1/2})$
HOPI	$O(m^{1/2})$	$O(n^3)$	$O(nm^{1/2})$
Dual-I	$O(1)$	$O(n+m+t^3)$	$O(n+t^2)$
Dual-II	$O(\log t)$	$O(n+m+t^3)$	$O(n+t^2)$

Table 1: Complexity comparison

In our approach, we consider a graph as having two components: a tree (spanning tree) plus a set of t non-tree edges. For sparse, tree-like graphs, we assume $t \ll n$. As we have previously mentioned, many real life graphs are sparse.

The two components together contain the complete reachability information of the original graph. The dual labeling scheme seamlessly integrates i) interval-based labels, which encode reachability in the spanning tree, and ii) non-tree labels, which encode additional reachability in the rest of the graph. At query time, we first consult the interval-based labels to see if two nodes are connected by tree edges, if not, we consult non-tree labels, and check if they are connected by paths that involve non-tree edges. For Dual-I, both operations have constant time complexity. For Dual-II, the second operation takes $O(\log t)$ time. Since $t \ll n$ for sparse graphs, $O(\log t)$ is often negligible. Furthermore, the two set of labels can be assigned by depth-first traversal of the graph, which is of linear complexity. The preprocessing step may take $O(t^3)$ time in the worst case. However, as we will demonstrate in our experiments, this cost is almost negligible for sparse graphs. To check reachability encoded by non-tree labels, the Dual-I approach relies on an additional data structure of size t^2. Since the spanning tree of a connected graph has $n-1$ edges, the number of non-tree edges t is at most $m-n+1$. This means for XMark datasets [1] whose edge/vertex ratio is approximately 1.15, the extra storage is about $0.022n^2$, and for the HumanCyc dataset [18], the extra storage is about $0.009n^2$, both of which is much smaller than the n^2 space used by the transitive closure matrix. Still, we introduce a tradeoff between time and space. By paying a negligible cost of $O(\log t)$ in query time, the Dual-II scheme manages to use much less space in query processing[1].

From our discussion above, it is clear that t, the number of non-tree edges, is an important performance factor in our approach. In this paper, we show that we can reduce t with-

[1]Although in the worst case the space requirement for Dual-II is still $O(n+t^2)$, in practice the space requirement is much less.

out losing reachability information in the original graph if we choose the spanning tree carefully. As a matter of fact, if we find spanning trees in the minimal equivalent graph of the original graph, we can minimize t, thus further improving query and indexing performance.

The rest of the paper is organized as follows. In Section 2, we survey some related work in the field of graph reachability. Section 3 presents the dual labeling scheme for encoding graph reachability. Section 4 introduce the time space tradeoff to further reduce label size. In Section 5 we discuss an optional processing step that finds the minimal equivalent graph of the input graph. Section 6 evaluate our approach on several datasets, and we conclude in Section 7.

2 Related Work

Graph reachability has applications in a wide range of areas. For example, in objected-oriented programming, graph reachability is important in managing class inheritance hierarchies. Aït-Kaci et al. [4] proposed a bottom-up bit vector labeling scheme (called *modulation*) that uses $O(n)$ bits per label, where n is the number of vertices. Later, Caseau [5] proposed a top-down labeling scheme (called *compact hierarchical encoding*) to exploit the reuse potential of the bits in order to achieve more compact encoding. In the area of semantic web, Christophides et al. [7] applied efficient labeling schemes to the problem of encoding subsumption hierarchies. In the area of network flow optimization, Katz et al. [12] proposed a method for labeling flow and connectivity in weighted flow graphs. However, their scheme assumes undirected graphs, where reachability is a much simpler problem.

Recently, reachability labeling has enjoyed much attention due to its application in XML query processing. The interval-based labeling scheme is one of the most widely used labeling scheme for tree structures. It assigns an interval to each node in a tree structure, and the ancestor-descendant relationships between two nodes can be determined by checking set containment relationships between their interval labels. For tree structures, the interval-based labeling approach answers reachability queries in constant time, and the labeling process is of linear complexity. Agrawal et al. [3] extends the interval-based approach to DAGs. In his approach, each node u is assigned a set of non-overlapping intervals $L(u)$. A node v is reachable from u iff every interval in $L(v)$ is contained by some interval in $L(u)$. Although labels can still be assigned efficiently, for large, complicated graphs, the size of $L(u)$ can be linear in the graph size. Because reachability queries require checking containment relationship for all intervals in a label, long labels can seriously impact query performance.

The 2-hop approach [8] was proposed to handle graph reachability queries. The 2-hop approach assigns to each node u two labels, $C_{in}(u)$ and $C_{out}(u)$, where $C_{in}(u)$ contains a set of nodes that can reach u, and $C_{out}(u)$ contains a set of nodes reachable from u. Then, a node v is reachable from node u if $C_{out}(u) \cap C_{in}(v) \neq \emptyset$. For the 2-hop approach, the overall label size can be as large as $O(nm^{1/2})$, which in the worst case approaches the $O(n^2)$ space requirement of the naïve approach that stores the transitive closure for reachability queries. For the 2-hop approach, reachability queries may take $O(m^{1/2})$ time because the average size of each label is $O(m^{1/2})$. An important issue with regard to the 2-hop approach is the complexity of its labeling process. Finding optimum 2-hop labeling is equivalent to solving the weighted set covering problem, which is NP-hard. Cohen et al. used an approximation algorithm that greedily finds the largest uncovered submatrix in the transitive closure matrix in each step. Still, the process is extremely time consuming for large datasets. Recently, much work has focused on improving the labeling efficiency of the 2-hop method. The HOPI algorithm [19, 20], for example, reduces 2-hop's labeling complexity from $O(n^4)$ to $O(n^3)$. However, it still cannot be applied to applications that involve massive graphs.

3. Dual Labeling

In this section, we present our dual labeling approach. The input is a directed graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, and we assume it is acyclic. If not, we find strongly connected components of G and collapse each component into a representative node. Clearly, all of the nodes in a strongly connected component is equivalent to its representative node as far as reachability is concerned. This step takes $O(n + m)$ time using Tarjan's algorithm [16]. Then, we find a spanning tree in the graph, and assign interval-based labels and non-tree labels to each node in the graph. We call this labeling scheme the *Dual-I* labeling scheme. We show that the complexity of assigning dual labels is close to linear for sparse graphs, and the Dual-I labeling scheme answers reachability queries in $O(1)$ time.

3.1. Non-Tree Edges and the Transitive Link Table

Our first step is to find a spanning tree in the graph so that we can assign interval-based labels to the nodes. We keep track of the non-tree edges so that the reachability information is complete. Note that the choice of the spanning tree has an impact on number of non-tree edges we must keep track of. We postpone the discussion of finding an optimum spanning tree to Section 5.

We use an example to demonstrate the problem at hand. In graph G shown in Figure 1, there are two nodes, x and y, whose in-degrees are greater than 1. This prevents us from directly applying interval-based labeling to G.

We find a spanning tree T in G. Let the solid lines in Figure 2 represent the edges of the spanning tree, then the dotted lines are *non-tree edges*. We assign an interval-based label $[start, end)$ to each node u, where $start$ and $end - 1$ are u's preorder and postorder number respectively (with regard to the spanning tree). Then, if the preorder number of node v is inside the range of $[start, end)$, then v is u's descendant in the spanning tree.

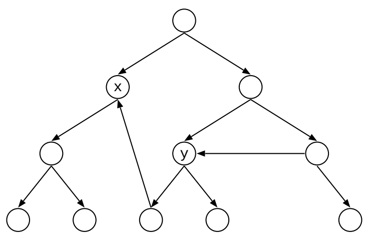

Figure 1: An input graph

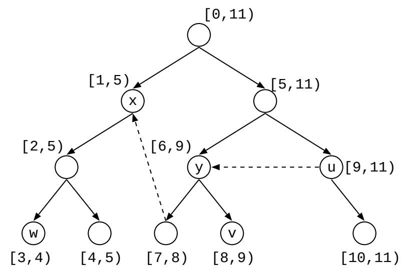

Figure 2: A spanning tree with interval-based labels

The reachability information contained in graph G and T are not the same. Thus, in addition to the tree, we must also keep track of the non-tree edges. If there is a non-tree edge from a node labeled $[a, b)$ to a node labeled $[c, d)$, then we record the edge in a *link table*. We denote this link by:

$$a \to [c, d)$$

Note that if $c \in [a, b)$, which means node $[c, d)$ is reachable from node $[a, b)$ via tree edges already, then the non-tree edge is superfluous, and there is no need to keep track of it. In Section 5, we show how input graphs can be preprocessed to remove superfluous edges so that the number of non-tree edges we need to store in the link table is minimal.

Combining interval-based labels and the link table, we have complete reachability information of the graph, as the following lemma indicates.

Lemma 1. *Assume two nodes u and v are labeled $[a, b)$ and $[c, d)$ respectively. There is a path from u to v iff $c \in [a, b)$ or the link table contains a series of m non-tree edges*

$$i_1 \to [j_1, k_1), \ldots, i_m \to [j_m, k_m) \qquad (1)$$

such that $i_1 \in [a, b)$, $c \in [j_m, k_m)$, and $i_{m'} \in [j_{m'-1}, k_{m'-1})$ for all $1 < m' \leq m$.

Proof. Interval-based labeling guarantees $c \in [a, b)$ is the necessary and sufficient condition of the existence of a tree path between node $[a, b)$ and $[c, d)$. If a path from u to v contains m non-tree links, then we can express the path in the form of Eq 1. On the other hand, if we are given a series of non-tree links as Eq 1, then because $i_1 \in [a, b)$, $c \in [j_m, k_m)$, and $i_{m'} \in [j_{m'-1}, k_{m'-1})$ for all $1 < m' \leq m$, we know there is a path between $[a, b)$ and $[c, d)$. □

As an example, in Figure 2, the path from u to v involves the non-tree edge $9 \to [6, 9)$, and the path from u to w involves two non-tree edges $9 \to [6, 9)$ and $7 \to [1, 5)$.

Applying Lemma 1 naïvely for answering reachability queries would involve traversing and exploring the non-tree edges in an iterative fashion, which is extremely costly. To "shortcut" this graph search, we can compute the transitive closure of the link table. That is, given two links $i_1 \to [j_1, k_1)$ and $i_2 \to [j_2, k_2)$ in the link table, if $i_2 \in [j_1, k_1)$, we add a new link $i_1 \to [j_2, k_2)$ to the table. We repeat this process until no new links can be added. We call the resulting table the *transitive link table* and denote it T.

Consider the example graph shown in Figure 2. From its link table which contains two non-tree edges $9 \to [6, 9)$ and $7 \to [1, 5)$, we generate a new link $9 \to [1, 5)$. Therefore, the transitive link table consists of the following entries:

$$9 \to [6, 9)$$
$$7 \to [1, 5)$$
$$9 \to [1, 5)$$

Property 1 (Size of the Transitive Link Table). *Assume the original link table has t entries. The transitive link table can have up to but no more than $\frac{t(t+1)}{2}$ entries.*

Proof. We denote each entry in the link table as $L(i) \to R(i)$, where $i = 1, \cdots, t$. A derived link has the form $L(i) \to R(j)$, $i \neq j$. Thus, potentially we can add $t(t-1)$ entries. Let $L(i) \to R(j)$ be a derived link. It must be derived from a series of links $L(i) \to R(i), \cdots, L(j) \to R(j)$. Then, the node represented by $L(j)$ is reachable from the node represented by $R(i)$. Because the graph does not have cycles, the potential entry $L(j) \to R(i)$ cannot be derived. This means at most half of the entries are eligible to be added into the transitive link table. □

The following theorem follows directly from Lemma 1 and the definition of the transitive link table.

Theorem 1. *Assume nodes u and v are labeled $[a, b)$ and $[c, d)$ respectively. There is a path from u to v if and only if $c \in [a, b)$ or there exists an entry $i \to [j, k)$ in the transitive link table such that $i \in [a, b)$, and $c \in [j, k)$.*

Thus, to check reachability between two nodes, we search the transitive link table. A linear search has time complexity $O(t^2)$. In the remainder of this section, we present methods to reduce the search complexity to $O(1)$.

3.2. Transitive Link Counting

Following the above discussion, given two nodes u and v with labels $[a_1, b_1)$ and $[a_2, b_2)$, we want to find out if there exists an entry $i \to [j, k)$ in the transitive link table such that $i \in [a_1, b_1)$ and $a_2 \in [j, k)$. Figure 3 serves to further illustrate the problem and show the intuition behind our solution given below.

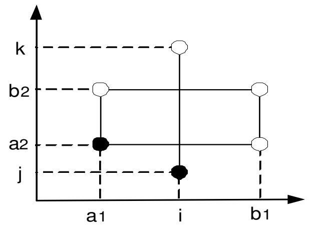

Figure 3: Intuition

Each link $i \rightarrow [j,k)$ in the transitive link table can be represented as a vertical line segment with i as the x coordinate and $[j,k)$ as the range of the y coordinate. The two nodes of interest, with labels $[a_1,b_1)$ and $[a_2,b_2)$, are represented as a query rectangle. Thus, the question of *whether there exists a link $i \rightarrow [j,k)$ such that $i \in [a_1,b_1)$ and $a_2 \in [j,k)$* is tantamount to the question of *whether there exists a vertical line segment that intersects (stabs through) the lower edge of the query rectangle*.

We note here that this is an instance of the *range-temporal aggregation* problem [21], for which a number of existing data structures with logarithmic query time are directly applicable. We will elaborate more on this point when we discuss space-time tradeoffs in Section 4. For now, however, our goal is $O(1)$ query time. Fortunately, our instance of the problem has many special properties that we can exploit for efficiency. Namely, the links in the transitive link table are not arbitrary vertical line segments, and the query rectangles are not arbitrary either; the endpoints of these objects all have coordinates corresponding to numbers used in interval labeling of a tree. We will see how to exploit these properties for efficient query processing.

As a first cut, we define the *TLC (transitive link count) function* $N(\cdot,\cdot)$ over the two-dimensional space as follows.

Definition 1 (Transitive Link Count). *The TLC function $N(x,y)$ computes the number of links $i \rightarrow [j,k)$ in the transitive link table that satisfy $i \geq x$ and $y \in [j,k)$.*

In Figure 3, the geometric interpretation of $N(a_1,a_2)$ is the number of vertical line segments intersecting the horizontal ray $x \geq a_1, y = a_2$. Similarly, $N(b_1,a_2)$ is the number of vertical line segments intersecting the horizontal ray $x \geq b_1, y = a_2$. Hence, the number of vertical line segments intersecting the lower edge of the query rectangle can be computed by $N(a_1,a_2) - N(b_1,a_2)$.

As a concrete example, based on the transitive closure table for the graph in Figure 2 in Section 3.1, we have $N(9,3) = 1$ because there is a link $9 \rightarrow [1,5)$ that satisfies the condition of Definition 1, and we have $N(11,3) = 0$ because no link satisfies the condition.

The following theorem shows that, with the TLC function $N(\cdot,\cdot)$, reachability queries can be answered directly.

Theorem 2. *Assume two nodes u and v are labeled $[a_1,b_1)$ and $[a_2,b_2)$ respectively, and u is not an ancestor of v in the spanning tree (i.e., $a_2 \notin [a_1,b_1)$). Node v is reachable from node u via some non-tree links if and only if*

$$N(a_1,a_2) - N(b_1,a_2) > 0 \tag{2}$$

Proof. According to Theorem 1, v is reachable from u via one or more non-tree edges if and only if there is a link $i \rightarrow [j,k)$ in the transitive link table such that $i \in [a_1,b_1)$ and $a_2 \in [j,k)$. According to Definition 1, there are $N(a_1,a_2)$ links satisfying $i \geq a_1$ and $a_2 \in [j,k)$; among them, $N(b_1,a_2)$ links have $i \geq b_1$. Thus, there is at least one link that satisfies $i \in [a_1,b_1)$ and $a_2 \in [j,k)$ as long as $N(a_1,a_2) - N(b_1,a_2) > 0$. □

As an example, consider the reachability between node u and node w in Figure 2. Here, the two nodes are labeled $[9,11)$ and $[3,4)$ respectively. We have $N(9,3) - N(11,3) = 1 - 0 > 0$, thus we know w is reachable from u via some non-tree links.

Based on the above discussion, we know that if we compute and store $N(x,y)$ for any pair of x and y, then the reachability query can be answered in constant time. The cost of storing one particular $N(x,y)$ value is low, as the following property shows.

Property 2 (Size of a TLC Value). *Any value of $N(\cdot,\cdot)$ can be stored in $2\log t$ bits.*

Proof. According to Definition 1, $N(\cdot,\cdot)$ is the number of links in the transitive link table that satisfy a certain condition. Since there are no more than $t(t+1)/2$ transitive links, the range of $N(\cdot,\cdot)$ is $[0, t(t+1)/2]$, thus it requires no more than $2\log t$ bits to store each value. □

Unfortunately, if we store $N(\cdot,\cdot)$ for each and every input pair that might be used for querying, the storage requirement would be prohibitive. The reason is that interval labels use $\Theta(n)$ distinct numbers, meaning that the number of possible input pairs for $N(\cdot,\cdot)$ is $O(n^2)$, which is unacceptable for large graphs. Next, we discuss ways to avoid storing the TLC function for all possible inputs.

3.3. Space Reduction by Gridding and Snapping

To reduce the storage requirement of the TLC function, we first observe that its value can change only at x coordinates where there is a vertical line segment, or at y coordinates where a vertical line segment begins or ends.

Intuitively, we can think of the two-dimensional space as covered by a grid of cells. Figure 4 shows the grid for the example graph in Figure 2. From Definition 1, it should be clear that for each grid cell, the value of the TLC function remains constant throughout the interior of the cell as well as its lower and right boundaries. Therefore, we can simply store the value at the lower-right corner point as the representative for the entire cell (cells to the far right do not need any representatives because the TLC value in them is always 0). To look up the value of $N(x,y)$, we simply "snap"

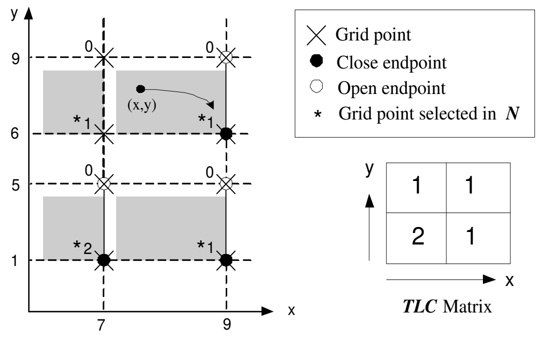

Figure 4: The TLC grid and $N(x, y)$ values

the point (x, y) to its representative grid point and retrieve the stored TLC value.

We can further reduce the storage requirement by more intelligent "snapping" that exploits the fact that all line segments come from interval labeling of a tree. Suppose we are checking reachability from $[a_1, b_1)$ to $[a_2, b_2)$ through non-tree edges, which can be determined by computing $N(a_1, a_2) - N(b_1, a_2)$. The following lemma shows that we can instead compute $N(a_1, a_0) - N(b_1, a_0)$, where a_0 is the $start$ label of the lowest (tree) ancestor of $[a_2, b_2)$ that has a non-tree incoming edge. Intuitively, the only way for $[a_1, b_1)$ to reach $[a_2, b_2)$ is through this node. Using this lemma, we only need to compute $N(\cdot, \cdot)$ for y coordinates that correspond to the lower ends of some vertical line segments. Therefore, we only need to store the TLC values at the following grid points (at most t^2 of them):

$$\{i \mid i \to [j, k) \in T\} \times \{j \mid i \to [j, k) \in T\}.$$

Lemma 2. *Consider any two nodes labeled $[a_1, b_1)$ and $[a_2, b_2)$ where $[a_2, b_2) \not\subseteq [a_1, b_1)$. Let $[a_0, b_0)$ be the label of the lowest (tree) ancestor of $[a_2, b_2)$ (or itself) with a non-tree incoming edge in the link table. If such a node exists, then $N(a_1, a_2) - N(b_1, a_2) = N(a_1, a_0) - N(b_1, a_0)$. If no such node exists, then $N(a_1, a_2) - N(b_1, a_2) = 0$.*

Proof. In the case where no such node exists, it is obviously impossible for $[a_1, b_1)$ to reach $[a_2, b_2)$, so $N(a_1, a_2) - N(b_1, a_2) = 0$. We now focus on the case when $[a_0, b_0)$ exists. $N(a_1, a_2) - N(b_1, a_2)$ counts the number of vertical line segments intersecting $x \in [a_1, b_1), y = a_2$. Thus, it suffices to prove that any vertical line segment intersecting $y = a_0$ must intersect $y = a_2$, and vice versa. For any vertical line segment $i \to [j, k)$ intersecting $y = a_0$, $[j, k)$ is an interval label containing a_0; therefore, $[j, k)$ is an ancestor of $[a_0, b_0)$ and in turn must be an ancestor of $[a_2, b_2)$, which implies that $i \to [j, k)$ also intersects $y = a_2$. On the other hand, for any vertical line segment $i \to [j, k)$ intersecting $y = a_2$, $[j, k)$ contains a_2 and therefore is an ancestor of $[a_2, b_2)$. At the same time, the fact that $i \to [j, k) \in T$ implies that $[j, k)$ has a non-tree incoming edge. However, $[a_0, b_0)$ is the lowest ancestor of $[a_2, b_2)$ with a non-tree incoming edge. Therefore, $[j, k)$ must be an ancestor of $[a_0, b_0)$ or $[j, k) = [a_0, b_0)$; either way, $[j, k)$ intersects $y = a_0$. □

To store the TLC values at necessary grid points, we use a *TLC matrix* $\mathbf{N}$. Let $index_x(i)$ denote the position (starting from 0) of i within the set $\{i \mid i \to [j, k) \in T\}$ ordered by value, and similarly, let $index_y(j)$ denote the position of j within the ordered set $\{j \mid i \to [j, k) \in T\}$. We store the TLC value $N(i, j)$ at $\mathbf{N}[index_x(i), index_y(j)]$. Clearly, $\mathbf{N}$ is at most a $t \times t$ matrix. The algorithm for constructing the TLC matrix is given as Algorithm 1.

Algorithm 1 Build the TLC matrix

```
COMPUTETLCMATRIX($G$)
1:  for each non-tree edge $a \to [b, c]$ in $G$ do
2:      insert $a$ into the ordered list $X$
3:      insert $b$ into the ordered list $Y$
4:  $index_x(x)$ ($index_y(y)$) is the index of $x$ in $X$ ($y$ in $Y$)
5:  initialize an $|X| \times |Y|$ matrix $\mathbf{N}$
6:  initialize a counter list $C(y) = 0$ for each $y \in Y$
7:  $x_c = \max(x)$ in $X$
8:  for each $i \to [j, k) \in T$ where $T$ is decreasingly sorted
    by $i$ do
9:      if $i < x_c$ then
10:         for each $y \in Y$ do
11:             $\mathbf{N}[index_x(x_c), index_y(y)] = C(y)$
12:         $x_c = i$
13:     for each $y \in [j, k)$ do
14:         $C(y) = C(y) + 1$
15: for each $y \in Y$ do
16:     $\mathbf{N}[index_x(x_c), index_y(y)] = C(y)$
```

3.4. Non-Tree Labeling

We now show how to assign *non-tree labels* to nodes, which would enable us to answer reachability queries in constant time with the help from the TLC matrix $\mathbf{N}$.

Definition 2 (Non-Tree Labels). *Let u be a node with interval label $[a, b)$. The* non-tree labels *of u is a triple $\langle x, y, z \rangle$, where*

- *$x = index_x(a')$, where $a' = \min\{i \mid i \to [j, k) \in T \wedge i \geq a)\}$. If such an a' does not exist, let x be the special symbol "$-$."*
- *$y = index_x(b')$, where $b' = \min\{i \mid i \to [j, k) \in T \wedge i \geq b)\}$. If such a b' does not exist, let y be "$-$."*
- *$z = index_y(a^*)$, where a^* is the start interval label of the lowest (tree) ancestor of u with a non-tree incoming edge. If such an a^* does not exist, let z be "$-$."*

Figure 5 shows an example of the non-tree labels. For instance, the non-tree label of the root node is $\langle 0, -, - \rangle$, because (1) the root start label "snaps" to the first x-coordinate in the TLC grid; (2) the root end label lies beyond the last x-coordinate and therefore "snaps" to $-$; and (3) the root has no ancestor with non-tree incoming edge. Similarly, the non-tree labels of nodes u and v are $\langle 1, -, - \rangle$ and $\langle 1, 1, 1 \rangle$, respectively.

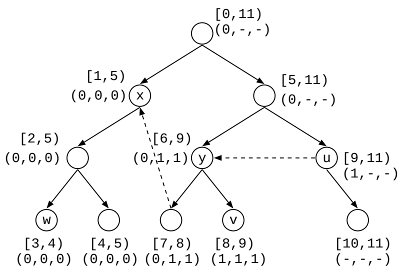

Figure 5: Graph with non-tree labeling

To assign non-tree labels, we use Algorithm 2. It basically traverses the spanning tree in a depth-first manner following the order of interval labels. The x component of the non-tree label is assigned when the traversal enters the node, and the y component is assigned when the traversal leaves the node. These labels are assigned in constant time by stepping through the ordered list of x-coordinates in the TLC grid in parallel. To assign the z component, a stack is used to keep track of the lowest ancestor with a non-tree incoming edge. Algorithm 2 has linear complexity. Besides, the process of creating the transitive link table (Section 3.1) in the worst case may take $O(t^3)$ steps, and Algorithm 1 $O(t^2)$ steps. Since $t \ll n$, our labeling algorithm is much more efficient than the popular 2-hop labeling [8], which has complexity $O(n^4)$, or the HOPI algorithm [20], which has complexity $O(n^3)$.

Now, we have the complete Dual-I labeling scheme. It is also the main result of this paper, which shows that we can answer reachability queries in constant time with interval labels, non-tree labels, and the help of the TLC matrix $\mathbf{N}$.

Theorem 3. *Suppose two nodes u and v are labeled $([a_1, b_1), \langle x_1, y_1, z_1 \rangle)$ and $([a_2, b_2), \langle x_2, y_2, z_2 \rangle)$ respectively. Node v is reachable from node u if and only if:*

- $a_2 \in [a_1, b_1)$, *or*
- $\mathbf{N}[x_1, z_2] - \mathbf{N}[y_1, z_2] > 0$. [2]

Proof. (Sketch) If v is reachable from u by tree edges, then we have $a_2 \in [a_1, b_1)$. Otherwise, the only way u can reach v is via non-tree edges. According to Theorem 2, v is reachable from u via some non-tree edge if $N(a_1, a_2) - N(b_1, a_2) > 0$. We use the same symbols a', b' and a^* as in Definition 2. If we can show

$$\begin{aligned} index_x(a_1) &= index_x(a'_1) = x_1, \\ index_x(b_1) &= index_x(b'_1) = y_1, \\ index_y(a_2) &= index_y(a^*_2) = z_2, \end{aligned}$$

then the theorem is proved.

According to Definition 2, there is no link $i \rightarrow [j, k]$ such that $i \in [a_1, a'_1)$, which means $index_x(a_1) = index_x(a'_1)$. Similarly, $index_x(b_1) = index_x(b'_1)$. Since a^*_2 is v's closest ancestor with an incoming link, there is no link $i \rightarrow [j, k)$ such that $j \in [a^*_2, a_2)$, which means $index_y(a_2) = index_y(a^*_2)$. Putting everything together, we have:

$$\mathbf{N}[x_1, z_2] - \mathbf{N}[y_1, z_2] = N(a_1, a_2) - N(b_1, a_2) > 0$$

□

[2] Let $\mathbf{N}[x, -] = \mathbf{N}[-, y] = 0, \forall x, y$.

Algorithm 2 Non-tree Labeling

```
ASSIGNNONTREELABEL($G$)
Stack ← {−}
$X$ and $Y$ are the same lists defined in Algorithm 1
append − to the end of $X$
$i = 0$
for each $root$ in $G$ sorted by $root$.start do
    LABEL($root$)

LABEL($n$)
$ix \leftarrow i$
if $n$ has an incoming link then
    Stack.push($index_y(n$.start$)$)
for each child $c$ of $n$ do
    LABEL($c$)
if $n$.end $> X(i)$ then
    $i = i + 1$
$iy \leftarrow i$
$n$'s non-tree label is $\langle ix, iy,$ Stack.top()$\rangle$
if $n$ has an incoming link then
    Stack.pop()
```

For example, in Figure 5, the non-tree labels of node u and w are $\langle 1, -, - \rangle$ and $\langle 0, 0, 0 \rangle$ respectively. Although u is not an ancestor of w in the spanning tree, w is reachable from u because $\mathbf{N}[1, 0] - \mathbf{N}[-, 0] = 1 > 0$.

4. Trading off Time for Space

The Dual-I labeling scheme introduced in the previous section supports constant query time by using non-tree labels (totaling $O(n)$ space) and a TLC matrix ($O(t^2)$ space) in addition to interval labels. In this section, we propose the Dual-II labeling scheme, which reduces the space requirement of the Dual-I scheme.

There are two potential opportunities for reducing the space requirement. (1) We can try to avoid using non-tree labels altogether. Although doing so does not lower the asymptotic space complexity overall (because interval labels already take $O(n)$ space), the space saving can be significant in practice. (2) We can replace the TLC matrix with an alternative data structure with lower space complexity. To realize these space saving opportunities, however, we need to make commensurate sacrifice in query efficiency. We discuss a number of approaches that explore this space-time tradeoff below.

In order to avoid storing any non-tree labels, we propose *TLC search tree* as an alternative to the TLC matrix, so that we can efficiently search for the value of $N(x, y)$ for any input pair without remembering which grid point (x, y) snaps to. The TLC search tree has two layers. The lower layer consists of a sequence of mini-trees, each indexing a row of TLC grid points (with the same y coordinate) by their x coordinates, as shown in Figure 6. Consecutive entries with identical TLC values do not need to be duplicated. The upper layer indexes the sequence of mini-trees by their y coordinates. To compute $N(x_0, y_0)$, we first search the upper layer for the mini-tree with the largest $y \leq y_0$; then, we search this mini-tree for the entry with the smallest $x \geq x_0$. The TLC value of result entry is equal to $N(x_0, y_0)$. This computation take $O(\log t)$ time overall, because there are at most $2t$ mini-trees and each mini-tree has at most t entries. Although in the worst case the TLC search tree may index $2t^2$ entries and require $O(t^2)$ space, in practice it may take less space than the TLC matrix because of the optimization that collapses consecutive entries with identical TLC values in each row.

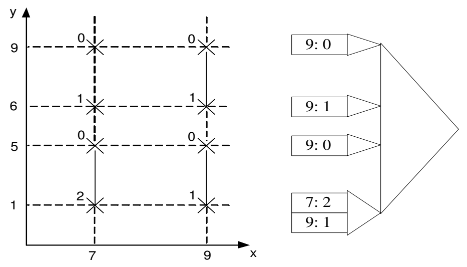

Figure 6: The search tree

Another approach that can further reduce storage in some situations is by casting the problem of checking reachability through the transitive link table as a *range-temporal aggregation problem* [21]. In Figure 6, we can regard the horizontal axis as the value dimension and the vertical axis as the temporal dimension. Each entry $i \longrightarrow [j, k)$ of the transitive link table, represented as a vertical segment, can be regarded as a fact with value i that is "alive" during time interval $[j, k)$. The query that checks whether there exists a vertical segment that intersects the horizontal segment $([a_1, b_1), a_2)$ can be reduced to a range-temporal `COUNT` query that counts the number of facts alive at time a_2 whose values are in $[a_1, b_1)$. The range-temporal `COUNT` problem has been studied extensively in both databases and algorithms communities. Suppose the transitive link table contains $|T|$ entries. In external memory with block size B, the *multiversion SB-tree* [21] can solve the problem in $O(\log_B |T|)$ time using $O(\frac{|T|}{B} \log_B t)$ space; the *CRB-tree* [10] further improves the space requirement to $O(\frac{|T|}{B})$. In internal memory, the *compressed range-tree* [6] can solve the problem in $O(\log_2 |T|)$ time with $O(|T|)$ space. Any of these data structures can replace the TLC matrix or the TLC search tree while also avoiding non-tree labels. In terms of query time, they are similar to the TLC search tree. In terms of space, they are least linear in $|T|$, which is $O(t^2)$. Therefore, these structures do not necessarily take less space than the TLC search tree (also $O(t^2)$ space), especially because they are more complicated and have bigger constant-factor space overhead not shown in asymptotic notation. However, in situations where there are many non-tree edges that cannot reach one another (i.e., $|T| \ll t^2$), we should use one of these data structures if logarithmic query time is acceptable.

5 Minimal Equivalent Graph

Recall that dual labeling starts with finding a spanning tree of the input graph (Section 3.1). The reachability information in the input graph is embodied by the edges in the tree plus some non-tree edges in the graph. The choice of the spanning tree has an impact on the number of necessary non-tree edges we must keep track of. In this section, we focus on minimizing the number of non-tree edges to reduce the extra space we need in dual labeling.

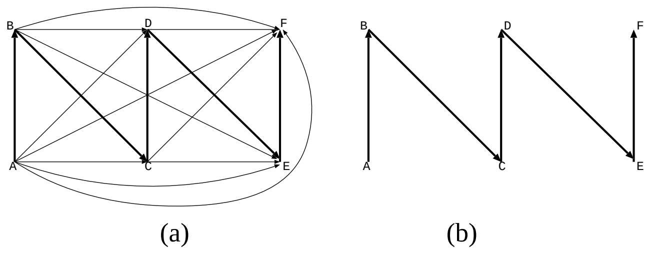

Figure 7: A graph and its minimal equivalent graph

Consider the graph shown in Figure 7(a). Many of its edges are superfluous as far as reachability is concerned. If the spanning tree consists of the bold-lined edges as shown in Figure 7(b), then no non-tree edge is needed, because the tree contains all reachability information. However, if we choose an arbitrary spanning tree, it is likely that we need to keep track of some non-tree edges. The question is how to choose a spanning tree such that the number of non-tree edges is minimal?

Agrawal et al. [3] finds the optimum tree-cover for a graph to minimize the number of intervals for interval-based labeling. Our focus is to minimize the number of non-tree edges. To do this, we find a minimal equivalent graph in the original graph, that is, we try to remove the maximum number of edges from the input graph without affecting its reachability. Then, we find a spanning tree in the reduced graph. Clearly, the remaining non-tree edges are minimal.

An algorithm for finding the minimal equivalent graph was given by Moyles and Thompson [15]. Their approach requires a maximum of up to $|V|^3 + \sum_{i=2}^{|V|-1} \binom{|V|-1}{|V|-i}$ steps of computation and a very large storage. Later, Hsu [11] gave an $O(n^3)$ algorithm for finding the minimal equivalent graph in a DAG by first computing its transitive closure matrix and then simplifying the matrix.

We propose an algorithm to find minimal equivalent graphs for sparse DAGs. We want to avoid computing the transitive closure matrix, because the $O(n^2)$ storage requirment is infeasible for large graphs. Our algorithm removes superfluous edges directly from the sparse graph. Although in the worst case (when each node is a parent of a given node), our algorithm may have to keep the ancestor sets of all nodes, in practice, it takes much less space and run time. The outline of our method is given in Algorithm 3.

Algorithm 3 Reduce a DAG to its minimal equivalent graph

```
MINIMALEQUIVALENTGRAPH($G$)
1: sort nodes in $G$ into topological order $v_1, \cdots, v_n$
2: $S_{v_1} = \emptyset$    /* ancestors of $v_1$ */
3: for each node $v$ in the topological order do
4:     assume $p_1, \cdots, p_k$ are $v$'s parent nodes
5:     for each $p_i$ do
6:         if $\exists j, p_i \in S_{p_j}$ then
7:             remove edge $p_i \to v$ from $G$
8:     $S_v = \{p_1, \cdots, p_k\} \cup S_{p_1} \cup \cdots \cup S_{p_k}$
```

First, we sort the nodes in G topologically into a linear order $v_1 \cdots v_i \cdots v_n$. Clearly, any node that can reach node v_i must appear before v_i in the linear order. The time complexity of the topological sort is $O(n + m)$.

Then, we visit the nodes in the topological order. For each node v, we remove edge $p_i \to v$ from G if p_i is an ancestor of any of v's parent. The ancestors of p_i can be computed as we visit the nodes in the topological order, and they can be discarded if all of p_i's child nodes have been visited.

We use the graph in Figure 7(a) as an example. The topological order is A, B, C, D, E, F. We visit the nodes in this order and remove superfluous edges. For instance, when we visit node C, we shall remove edge $A \to C$, because node A is in the ancesotr set of node B. Finally, the graph is reduced to Figure 7(b).

Now, we show that the remaining graph is indeed a minimal equivalent graph. That is, it is the smallest subgraph that contain all reachability information of the original graph.

Theorem 4. *The graph generated by Algorithm 3 is a minimal equivalent graph.*

Proof. First we show that the edges removed by Algorithm 3 will not affect the reachability of the graph. Assume an edge $p_i \to v$ is removed. According to the algorithm, there must exist a path that connects p_i to v through p_j, that is, $p_i \xrightarrow{*} p_j \to v$. But $p_i \to v$ cannot be part of the path $p_i \xrightarrow{*} p_j$ because otherwise $v \xrightarrow{*} p_j \to v$ becomes a cycle, which contradicts to the acyclic assumption of G. Second, we show that the remaining graph is minimal. Assume we remove one more edge $p_i \to v$. Now v is not reachable from p_i, because it is neither reachable from p_i directly, nor through any of v's parent nodes. □

6 Experiments

In this section, we evaluate our proposed Dual-I and Dual-II labeling schemes. We implemented both schemes in C++ using the Boost Graph Library [9] for graph support. We also implemented the interval-based and the 2-hop labeling methods for comparison. We evaluate them on two types of synthetic data – random graphs (Section 6.1) and single rooted DAGs (Section 6.2), as well as some real life graphs (Section 6.3). The experiments are carried out on a PC with a 3.0 GHz Intel Pentium 4 processor, 1GB memory and 40GB hard drive.

We measure the time and space complexity of the above labeling approaches. Time complexity has been our main concern. We measure *indexing time* for creating labels and *query time* for answering reachability queries. A query is generated by randomly picking a pair of nodes for reachability test. For every labeling scheme, we measure the time of answering $100,000$ same random reachability queries. We noticed that the overhead of $100,000$ iterations of retrieving two nodes accounts for a large proportion in the total running time. To resolve this, we measure the baseline time of an "no-op" iteration, that is, we retrieve two nodes but do nothing else in the loop. The real *query time* is defined as the difference between the total elapsed time and the baseline time. Besides time complexity, we also measure the *storage requirements* of the labeling schemes and compare with storing the transitive closure matrix.

6.1 Random Graphs

We evaluate the performance of the labeling schemes over random graphs generated by the Boost Graph library [9]. In Figure 8, we show the results on graphs having 2000 nodes, with the number of edges ranging from 2100 to 3900 (on the X axis). We have tested random graphs of various sizes and densities, and the results are consistent.

Random graphs are likely to contain cycles. Our preprocessing step merges strongly connected components into representative nodes, and the optional step of finding *minimal equivalent graphs* removes superfluous edges. The first bar chart in Figure 8 illustrates the reduction of nodes and edges after applying the two preprocessing steps. The ratio keeps decreasing when the number of edges increases, which means, the denser a random graph is, the more likely it contains cycles, and the more likely it contains superfluous edges. Note in the bar chart, since the number of edges

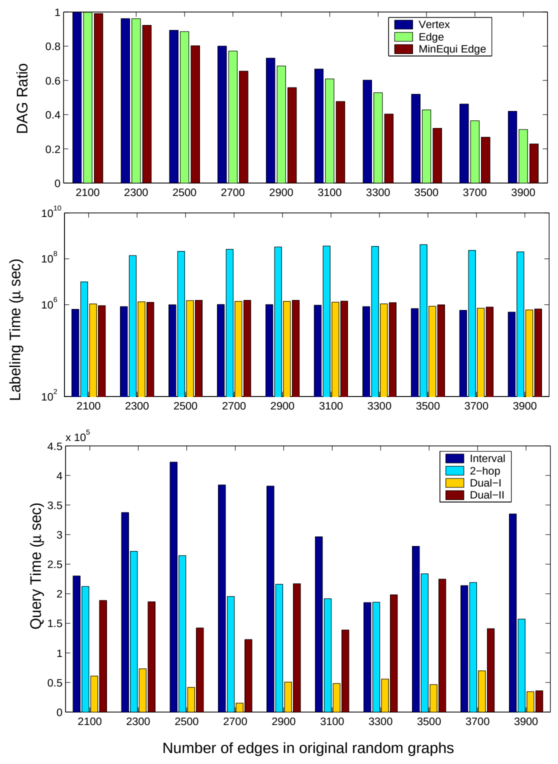

Figure 8: Results for random graphs with different edges ($|V| = 2000$, $|Q| = 100,000$)

in the original graphs is increasing (on the x axis), ratio decrease does not mean that the absolute number of edges is dropping. In fact, the number of edges after preprocessing reaches its peak value of 2214 when the original graph has 2500 edges.

The bar chart in the middle of Figure 8 reports *indexing time* of the random graph (after preprocessing). It shows that our Dual-I and Dual-II labeling schemes are comparable to interval-based labeling, which confirms our claim that the indexing complexity of the dual labeling schemes is almost linear. Compared to dual labeling, the 2-hop approach is 2 to 3 orders of magnitude more time consuming. Particularly, note that the indexing time of the 2-hop increases dramatically when the number of edges increases from 2100 to 2300. This is because when the number of edges is around 2100, the graph is very sparse, and most nodes are only connected to a few other nodes. In other words, the C_{in} and C_{out} labels for most nodes are very small, so the value of $S(in, out) \cap T$ can be quickly evaluated. When the number of edges increases to a certain level, long paths in the graph are formed, and C_{in} and C_{out} become large, which results in an significant increase of indexing time.

The bar chart in the bottom of Figure 8 reports *query time* over the random graphs. The numbers are averaged over multiple runs. We observe the following: i) Although the labeling time of the interval-based approach is comparable to our dual labeling approaches, it has the worst query performance. ii) The Dual-II labeling scheme is comparable to 2-hop in query performance. iii) The Dual-I labeling scheme consistently performs the best. In summary, the dual labeling approach optimizes both indexing (labeling) time and query time, and the experiment results shown here verified our theoretical analysis of the advantage of the dual labeling method.

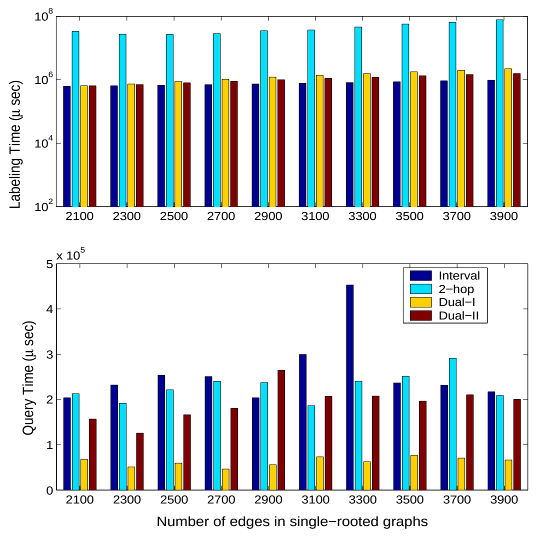

Figure 9: Results for single rooted DAGs with different edges ($|V| = 2000$, $|Q| = 100,000$)

6.2 Single Rooted DAGs

Except for 2-hop labeling, all other labeling schemes first convert input graphs into DAGs. We generate synthetic single rooted DAGs in order to study labeling performance more specifically. To generate DAGs, we first generate a spanning tree in a breadth-first manner. The shape of the spanning tree is adjusted by a maximum fanout parameter. Then we add non-tree edges $u \rightarrow v$ between randomly picked nodes u and v. To generate DAGs, we ensure u is on a higher level (the root is in the top level) than v. If u and v are on the same level, then u must have a smaller $start$ label (on the left).

Figure 9 shows the indexing time and query time for DAGs of 2000 nodes and varying number of edges. We found the results are similar to those obtained from random graphs: the dual labeling schemes outperforms other approaches in query time and is comparable to interval-based labeling in indexing time, which again confirms in practice, dual labeling schemes have almost linear indexing complexity. The 2-hop approach is 2 orders of magnitude more time consuming than the dual labeling approach. Note that when graphs are very sparse (e.g., with 2100 edges), the labeling process of 2-hop takes much longer when the graph is a DAG than when it is a random graph. The reason lies in the characteristics of the graph: a single rooted DAG is

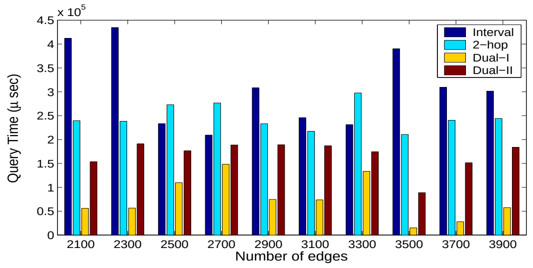

Figure 10: Query time of DAGs ($|V| = 2000$, $\max fanout = 9$)

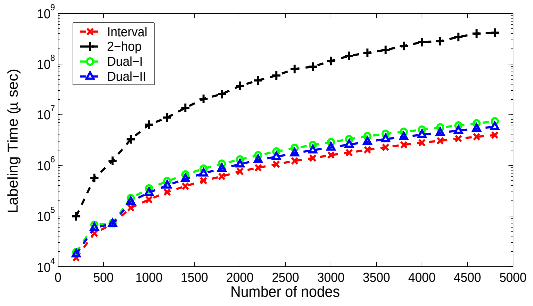

Figure 11: Indexing time for DAGs of fixed density, increasing size

different from a sparse random graph in that all nodes are connected in the DAG even if the number of edges is small, in which case, C_{in} and C_{out} of many nodes in the DAG are much larger than those in the random graph.

Next, we discuss the impact of the shape of the graph on query performance. The DAGs used for Figure 9 are generated based on spanning trees of maximum fanout 5, so the spanning tree is thin and tall. Figure 10 shows the query time for DAGs generated with maximum fanout 9. The conclusion is that query performance is not sensitive to the graph shape. Another graph characteristic is its density. We experimented with DAGs of fixed density ($m/n = 1.5$). We find query performance consistent with previous experiments. In Figure 11, we show the impact of DAG size on indexing time when the density is fixed. The interval-based approach is the fastest; the Dual-I and Dual-II schemes are a little slower but still comparable to the interval approach; the 2-hop is slower than other approaches by several orders of magnitude.

We now discuss the space requirements of the labeling schemes. Unlike the interval-based and the 2-hop labeling schemes, the Dual-I labeling scheme also relies on a *TLC* matrix, and the Dual-II labeling scheme relies on a search tree. We study both the space and the time complexity of the labeling schemes. Figure 12 and Figure 13 show the space requirement and the query time performance using the same synthetic graphs (number of nodes fixed at 2000 and number of edges varying from 2100 to 3000). The space requirement of the transitive closure matrix is shown as a horizontal line in Figure 12. It shows that the space requirement of the Dual-I scheme grows fast with the increase of graph density, and that the space requirement of the Dual-II scheme is comparable with the 2-hop and the interval-based approaches. It shows that Dual-I is good for sparse graphs only. However, as we see in Figure 13, the query performance of Dual-I is barely worse than that of the transitive closure matrix and is certainly much better than other labeling schemes. In other words, when a graph is quite sparse, the Dual-I scheme achieves query performance close to that of the transitive closure matrix without paying the latter's storage cost. In fact, as indicated by Figure 14, many real life graphs, such as VchoCyc, AgroCyc, Eco_O157 [13], and Xmark [1] are sparse enough to bring out the advantage of the Dual-I labeling scheme. Figure 14 shows the required space for graphs with number of nodes fixed at $10,000$. We did not show the curve for the 2-hop approach in Figure 14 because it is too time consuming to compute 2-hop labels for graphs with $10,000$ nodes.

6.3 Real Graphs

We also show results on some real graphs. The first four datasets in Table 2 are from EcoCyc [13]. The last dataset is an XML document generated by the XMark benchmark [1]. The first two columns are the numbers of nodes and edges in the original graph. The next two columns are those of the DAGs after preprocessing. The $|E_{MEG}|$ column shows the number of edges of the minimal equivalent graph. We report the indexing time and the query time. Because the 2-hop labeling is extremely slow to build for large graphs (the XMark graph takes 307 minutes for 2-hop labeling), we only evaluate the results of the other three approaches. The results are consistent with our previous experiments on synthetic graphs. Our Dual-I and Dual-II approaches are not much different from interval-based labeling in indexing time, and outperform interval-based labeling by at least one order of magnitude in query time.

7 Conclusion

Many applications involve massive, sparse graphs, yet require fast answering of graph reachability queries. State of the art reachability labeling schemes such as 2-hop have relatively efficient query performance, but has high complexity of indexing (labeling), which prevents them from being used on massive graphs. In this paper, we propose a novel graph reachability labeling scheme called the dual labeling scheme. It seamlessly integrates interval-based labels and non-tree labels, and achieves constant time query processing (the Dual-I scheme). Furthermore, the labeling complexity of our approach is close to linear for sparse graphs, which makes it applicable to massive datasets. Theoretical and experiment analysis demonstrated the effective of our approach.

References

[1] XMARK: The XML-benchmark project. http://monetdb.cwi.nl/ xml, 2002.

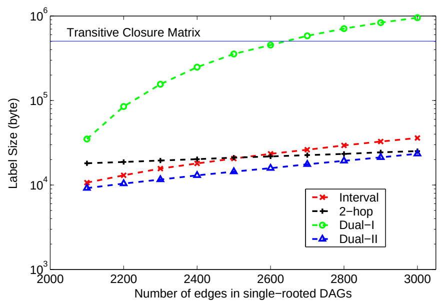

Figure 12: Label sizes of DAGs ($|V| = 2000$)

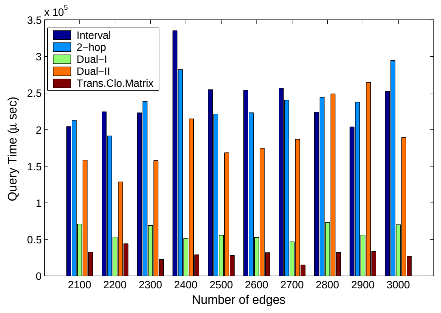

Figure 13: Query Time of DAGs ($|V| = 2000$)

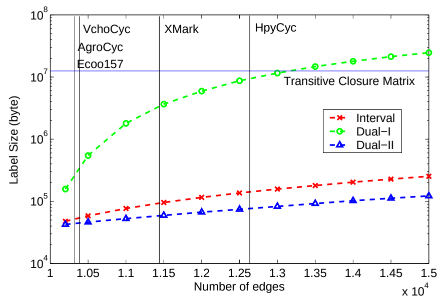

Figure 14: Label sizes of DAGs ($|V| = 10000$)

Graph Name	$\|V_G\|$	$\|E_G\|$	$\|V_{DAG}\|$	$\|E_{DAG}\|$	$\|E_{MEG}\|$	Indexing Time (*ms*)			Query Time (*ms*)		
						Interval	Dual-I	Dual-II	Interval	Dual-I	Dual-II
AgroCyc	13969	17694	12684	13408	13094	39242	40001	39916	1720	131	243
Ecoo157	13800	17308	12620	13350	13025	38529	39826	39037	1056	115	201
HpyCyc	5565	8474	4771	5859	5649	5926	6324	6253	806	108	337
VchoCyc	10694	14207	9491	10143	9860	21770	21943	22186	3253	81	165
XMark	6483	7654	6080	7028	6957	9065	9208	9767	1075	71	198

Table 2: Results of some real graphs

[2] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *Proc. ACM-SIAM Symposium on Discrete Algorithms(SODA)*, 2001.

[3] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, pages 253–262, 1989.

[4] 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.

[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] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.

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

[8] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the 13th annual ACM-SIAM Symposium on Discrete algorithms*, pages 937–946, 2002.

[9] Beman Dawes and David Abrahams. The boost c++ library. http://www.boost.org/.

[10] Sathish Govindarajan, Pankaj K. Agarwal, and Lars Arge. Crb-tree: An efficient indexing scheme for range-aggregate queries. In *ICDE*, 2003.

[11] Harry T. Hsu. An algorithm for finding a minimal equivalent graph of a digraph. *SIAM Journal of Computing*, 22(1):11–16, 1975.

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

[13] I.M. Keseler, J. Collado-Vides, S. Gama-Castro, J. Ingraham, S. Paley, I.T. Paulsen, M. Peralta-Gil, and P.D. Karp. Ecocyc: A comprehensive database resource for escherichia coli. *Nucleic Acids Research*, 33(D334-D337), 2005.

[14] Michael Ley. DBLP database web site. http://www.informatik.uni-trier.de/ ley/db, 2000.

[15] D. M. Moyles and G. L. Thompson. An algorithm for finding a minimal equivalent graph of a digraph. *SIAM Journal of Computing*, 16(3):455–460, 1969.

[16] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16:973–988, 1987.

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

[18] P. Romero, J. Wagg, M. L. Green, D. Kaiser, M. Krummenacker, and P. D Karp. Computational prediction of human metabolic pathways from the complete human genome. *Genome Biology*, 6(1):1–17, 2004.

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

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

[21] Donghui Zhang, Alexander Markowetz, Vassilis J. Tsotras, Dimitrios Gunopulos, and Bernhard Seeger. Efficient computation of temporal aggregates with range predicates. In *PODS*, 2001.