Core Labeling: A New Way to Compress Transitive Closure

Yangjun Chen

Dept. Applied Computer Science, University of Winnipeg
515 Portage Ave., Winnipeg, Manitoba, Canada R3B 2E9
y.chen@uwinnipeg.ca

ABSTRACT
A graph reachability query, as one of the primary tasks in numerous applications, is to find whether two given data objects, u and v, are related in any way in a large and complex dataset. Formally, the query is about to find if v is reachable from u in a directed graph which is large in size. In this paper, we focus ourselves on building a reachability labeling for large directed graphs, in order to process reachability queries efficiently. A new approach is proposed to compress transitive closure to support reachability checkings. The approach consists of two schemes, called Core-I labeling and Core-II labeling, respectively. For a graph \( G \) with \( n \) nodes and \( e \) edges, the labeling time of Core-I is bounded by \( O(n + e + t \cdot \min\{b, s\}) \), where \( b \) is the number of the leaf nodes of a spanning tree of \( G \), \( t \) is the number of the non-tree edges (edges that do not appear in the spanning tree) and \( s \) is the number of the start nodes of all non-tree edges in \( G \). The space overhead is bounded by \( O(n + s \cdot \min\{b, s\}) \) and the querying time is \( O(\log(\min\{b, s\})) \). Core-II needs \( O(n + e + \min\{b, s\} + d \cdot s \cdot \log(\min\{b, s\})) \) labeling time and \( O(n + d \cdot s) \) space, where \( d \) is the number of the end nodes of all non-tree edges in \( G \). But the query time is reduced to \( O(1) \). Experiments have been performed, showing that our method is promising.

1. INTRODUCTION
Given two nodes \( u \) and \( v \) in a directed graph \( G = (V, E) \), we want to know if there is a path from \( u \) to \( v \). The problem is known as graph reachability. In many applications, such as transportation network, internet traffic analyzing, semantic web, and computer vision, as well as metabolic network and XML query processing, graph reachability is one of the most basic operations, and therefore needs to be efficiently supported. Among them, some use sparse graphs, such as XML documents which are a labeled tree plus several IDREF/ID links, and metabolic networks which are an evolution tree plus some genes’ interactions.

A naive method is to precompute the reachability between every pair of nodes — in other words, to compute and store the transitive closure (TC for short) of a graph. Then, a reachability query can be answered in constant time. However, this requires \( O(n^2) \) space, which makes it impractical for massive graphs. Another method is to compute the shortest path from \( u \) to \( v \) over such a large graph on demand, which results in high query processing cost.

In this paper, we propose two schemes to compress transitive closure: Core-I and Core-II to speed up reachability queries for massive graphs. The main idea behind them is to recognize a subset of nodes of \( G \) and assign them labels in such a way that the reachability through non-tree edges can be determined by checking such labels only. For the Core-I labeling scheme, we need \( O(n + e + t \cdot \min\{b, s\}) \) labeling time, where \( b \) is the number of the leaf nodes of a spanning tree (or a spanning forest) of \( G \), \( t \) is the number of non-tree edges (edges that do not appear in the spanning tree) and \( s \) is the number of the start nodes of all non-tree edges in \( G \). The space overhead and the query time are bounded by \( O(n + s \cdot \min\{b, s\}) \) and \( O(\log(\min\{b, s\})) \), respectively. The Core-II labeling scheme has constant query time, but needs \( O(n + e + \min\{b, s\} + d \cdot s \cdot \log(\min\{b, s\})) \) labeling time, where \( d \) is the number of the end nodes of all non-tree edges in \( G \). The space overhead of Core-II is bounded by \( O(n + d \cdot s) \). Note that we always have \( d \leq \min\{n, t\}, s \leq \min\{n, t\} \). Especially, in the case of sparse graphs, we have \( t \leq n \) [23].

Although \( d, b \) and \( s \) are in general much smaller than \( n \), they are in the order \( O(n) \) for non-sparse graphs in the worst case.

The remainder of the paper is organized as follows. In Section 2, we review the related work. Section 3 is devoted to the description of our algorithms. Finally, a short conclusion is set forth in Section 4.

2. RELATED WORK
In the past two decades, many interesting labeling-based methods have been proposed to speed up reachability query evaluation, which can be roughly classified into two groups: strategies for sparse graphs and strategies for non-sparse graphs. In the following, some of them are reviewed.

- Strategies for sparse graphs
In [23], Wang et al. discussed an interesting approach, called Dual-I, for sparse graphs. It first finds a spanning tree, and then assigns to each node \( v \) a dual label: \((a_v, b_v)\) and \((x_v, y_v, z_v)\). In addition, a \( t \times t \) matrix \( N \) (called a TLC matrix) is maintained, where \( t \) is the number of non-tree edges. Another node \( u \) with \((a_u, b_u)\) and \((x_u, y_u, z_u)\) is reachable from \( v \) iff \( a_u \in \{a_v, b_v\} \), or \( N(x_v, z_u) > 0 \).

The size of all labels is bounded by \( O(n + e + t^2) \) and can be produced in \( O(n + e + t^2) \) time. The query time is \( O(1) \). As a variant of Dual-I, one can also store \( N \) as a tree (called a TLC search tree), which can reduce the space overhead from a practical viewpoint, but increases the query time to \( \log t \). This scheme is referred to as Dual-II.

The method proposed by Cohen et al. [6] labels a graph based on the so-called 2-hop covers. It is also designed specifically for sparse graphs. A hop is a pair \((h, v)\), where \( h \) is a path in \( G \) and \( v \) is
one of the endpoints of \( h \). A 2-hop cover is a collection of hops \( H \) such that if there are some paths from \( v \) to \( u \), there must exist \((h_1, v) \in H \) and \((h_2, u) \in H \) and one of the paths between \( v \) and \( u \) is the concatenation \( h_1 h_2 \). Using this method to label a graph, the worst space overhead is in the order of \( O(n \sqrt{e}) \). The main theoretical barrier of this method is that finding a 2-hop cover of minimum size is an NP-hard problem. So a heuristic method is suggested in [6], by which each node \( v \) is assigned two labels, \( C_{in}(v) \) and \( C_{out}(v) \), where \( C_{in}(v) \) contains a set of nodes that can reach \( v \), and \( C_{out}(v) \) contains a set of nodes reachable from \( v \). Then, a node \( u \) is reachable from node \( v \) if \( C_{in}(v) \cap C_{out}(v) \neq \emptyset \). Using this method, the overall label size is increased to \( O(n \sqrt{e} \log n) \). In addition, the reachability queries take \( O(\sqrt{e}) \) time because the average size of each label is above \( O(\sqrt{e}) \). The time for generating labels is \( O(n^2) \).

### 3.1 Tree labeling

As with Dual-I labeling, we will first find a spanning tree \( T \) (or a spanning forest) of \( G \). This can be done by running the algorithm given in [23]. Then, we label \( T \) as follows.

Each node \( v \) is assigned an interval \([\text{start}, \text{end}]\), where \( \text{start} \) is \( v \)'s preorder number and \( \text{end} - 1 \) is the largest preorder number among all the nodes in \( T[v] \). So another node \( u \) labeled \([\text{start}', \text{end}']\) is a descendant of \( v \) (with respect to \( T \)) iff \( \text{start}' \in (\text{start}, \text{end}) \) [23].

**Fig. 1. Illustration of interval-based labeling**

In the figure, the solid arrows represent the edges of the spanning tree while the dashed arrows are non-tree edges. Let \( v \) and \( u \) be two nodes in \( T \), labeled \([a, b]\) and \([a', b']\), respectively. If \( a \in [a', b'] \), we say, \([a, b]\) is subsumed by \([a', b']\). In this case, we must also have \( b \leq b' \). Therefore, if \( v \) and \( u \) are not on the same path in \( T \), we have either \( a' \geq b \) or \( a \geq b' \). In the former case, we say, \([a, b]\) is smaller than \([a', b']\), denoted \([a, b] \prec [a', b']\). In the latter case, \([a', b']\) is smaller than \([a, b]\).
### 3.2 Core of G

Now we define the core of G.

Let $T$ be a spanning tree of G. We denote by $E'$ the set of all the non-tree edges. Denote by $V$ the set of all the end points of the non-tree edges. Then, $V' = V_{\text{start}} \cup V_{\text{end}}$ where $V_{\text{start}}$ stands for a set containing all the start nodes of the non-tree edges and $V_{\text{end}}$ for all the end nodes of the non-tree edges.

**Definition 1.** An anti-subsuming subset $S \subseteq V_{\text{start}}$ is called an anti-subsuming set iff $|S| \geq 1$ and no two nodes in S are related by ancestor-descendant relationship with respect to $T$.

As an example, consider the spanning tree shown in Fig. 1.

With respect to this spanning tree, $V_{\text{start}} = \{d, f, g, h\}$. We have altogether 11 anti-subsuming subsets as shown in Fig. 2.

**Definition 2.** A node $v$ in a spanning tree $T$ of G is critical if $v \in V_{\text{start}}$ or there exists an anti-subsuming subset $S = \{v_1, v_2, ..., v_k\}$ for $k \geq 2$ such that $v$ is the lowest common ancestor of $v_1, v_2, ..., v_k$. We denote by $V_{\text{critical}}$ the set of all critical nodes.

Fig. 2. Anti-subsuming subsets

In the spanning tree shown in Fig. 1, node $e$ is the lowest common ancestor of $\{f, g\}$, and node $a$ is the lowest common ancestor of $\{d, f, g, h\}$. So $e$ and $a$ are two critical nodes. In addition, each $v \in V_{\text{start}}$ is a critical node. So all the critical nodes of G with respect to $T$ are $\{d, f, g, h, e, a\}$. We call a critical node trivial if it belongs to $V_{\text{start}}$; otherwise, non-trivial.

**Definition 3.** Let $G = (V, E)$ be a directed graph. Let $T$ be a spanning tree of G. The core of $G$ with respect to $T$ is a tree structure with the node set being $V_{\text{critical}}$, in which there is an edge from $u$ to $v$ if there is a path $P$ from $u$ to $v$ in $T$ and $P$ contains no other critical nodes. The core of G with respect to $T$ is denoted $G_{\text{core}} = (V_{\text{core}}, E_{\text{core}})$.

**Example 1.** Consider the graph G and the corresponding spanning tree $T$ shown in Fig. 1 once again. The core of G with respect to $T$ is shown in Fig. 3.

![Fig. 3. The core of G](image)

### 3.3 Graph labeling: Core-I

In this subsection, we show our first scheme: Core-I for graph labeling. The approach works in two steps. In the first step, we generate a data structure, called the core label (for G). It is in fact a set of interval sequences. In the second step, the core label is used to create non-tree labels for all the nodes in G.

#### 3.3.1 Core labeling

The core label for G is defined as below.

**Definition 4.** Let $V_{\text{core}} = \{v_1, ..., v_g\}$ be the node set of $G_{\text{core}}$. The core label for G is a set $\{L(v_1), ..., L(v_g)\}$, where each $L(v_l) (l = 1, ..., g)$ is an interval sequence associated with $v_l$ satisfying the following two properties:

1. Let $L(v_l) = [a_{l_1}, b_{l_1}), ..., [a_{l_r}, b_{l_r})$ for some r. Then, for any $i, j \in \{1, ..., r\}$, $b_{l_i} \leq a_{l_j}$ if $i < j$. That is, $[a_{l_i}, b_{l_i}) \prec [a_{l_j}, b_{l_j})$ for $i < j$. (In this sense, the intervals in $L(v_l)$ are considered to be sorted.)

2. Let $(a, b)$ be the interval associated with a descendant of $v_l$ with respect to G. There exists an interval $[a_{l_i}, b_{l_i}) (1 \leq i \leq r)$ in $L(v_l)$ such that $a \in [a_{l_i}, b_{l_i})$.

In order to generate the core label for G, we will first establish a graph, called a link graph, specified by the following definition.

**Definition 5.** Let $G = (V, E)$ be a directed graph. Let $T$ be a spanning tree of G. The link graph of G with respect to $T$ is a graph, denoted $G_{\text{link}}$, with the node set being $V'$ (the end points of all the non-tree edges) and the edge set $E' \cup E''$, where $E'$ is the set of all the non-tree edges, and for any $u, v \in V'$, $(u, v) \in E''$ iff $u \in V_{\text{end}}$ and $v \in V_{\text{start}}$.

**Example 3.** The link graph of G (shown in Fig. 1) with respect to the corresponding $T$ is shown in Fig. 6.

![Fig. 6. A link graph](image)

As the first step to generate the core label for G, we unite $G_{\text{core}}$ and $G_{\text{link}}$ to create a combined graph, denoted $G_{\text{com}} = G_{\text{core}} \cup G_{\text{link}}$, as shown in Fig. 7(a).

Now we notice that by labeling $T$, each node in $G_{\text{com}}$ will be initially associated with an interval as illustrated in Fig. 7(a). That is, if a node $v$ is labeled with $(a, b)$ in $T$, it will be initially labeled with the same interval $(a, b)$ in $G_{\text{com}}$.

If we can construct, for each node $v$, an interval sequence as shown in Fig. 7(b), the descendants of a node in $G_{\text{com}}$ can be represented in an economical way. Let $L = [a_1, b_1), ..., [a_k, b_k)$ be an interval sequence and each $[a_i, b_i)$ is an interval labeling a node $v_i (i = 1, ..., k)$ in $G_{\text{com}}$. Then, $L$ corresponds to the union of a set of subtrees $T[v_1], ..., T[v_k]$. For example, the interval sequence $[2, 4)[4, 5)[6, 9)[9, 12)$ associated with $h$ in Fig. 7(b) represents a union of 4 subtrees: $T[c], T[d], T[e]$ and $T[h]$, which contains all the descendants of $h$ in $G$. 

![Fig. 7. $G_{\text{com}} \cup G_{\text{link}}$ and a set of interval sequences](image)
For each \( v \) in \( G_{\text{com}} \), we will create such an interval sequence \( L(v) \). For this purpose, we sort the nodes of \( G_{\text{com}} \) topologically, i.e., \((v_i, v_j) \in G_{\text{com}} \) implies that \( v_i \) appears before \( v_j \) in the sequence of the nodes. The intervals to be generated for a node \( v \) are simply stored in a linked list \( A_v \) (see Fig. 8 for illustration). Initially, each \( A_v \) contains only one interval produced by labeling \( T \). It is trivially sorted. We scan the reverse topological sequence of the nodes one by one and at each step we do the following:

Let \( v \) be the node being considered. Let \( v_1, \ldots, v_k \) be the children of \( v \) (in \( G_{\text{com}} \)). Merge \( A_v \) with each \( A_{v_i} \) for the child node \( v_i \) (\( i = 1, \ldots, k \)) as follows. Assume \( A_{v_i} = p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_k \) and \( A_v = q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q_n \) each stored as a linked list as shown in Fig. 8. Assume that both \( A_v \) and \( A_{v_i} \) are increasingly ordered. (As we will see soon, any interval sequence generated by the following algorithm has this nice property. It contains only the intervals not on the same path in \( T \)).

![Fig. 8. Linked lists associated with nodes in \( G_{\text{com}} \)](image)

We step through both \( A_v \) and \( A_{v_i} \) from left to right. Let \( p_i = [a_i, b_i] \) and \( q_i = [a_i, b_i] \) be the intervals encountered. We will conduct the following checkings.

1. If \( a_i \geq b_j \), insert \( q_j \) into \( A_v \) after \( p_i \) and move to \( q_{j+1} \).
2. If \( a_i \in [a_j, b_j] \), remove \( p_i \) from \( A_v \) and move to \( p_{i+1} \). (*\( p_i \) is subsumed by \( q_j \)*)
3. If \( a_i \notin [a_j, b_j] \), ignore \( q_j \) and move to \( q_{j+1} \). (*\( q_j \) is subsumed by \( p_i \); but it should not be removed from \( A_{v_i} \).*)
4. If \( a_i \geq b_j \), ignore \( p_i \) and move to \( p_{i+1} \).
5. If \( a_i = a_j \) and \( b_i = b_j \), ignore both \( p_i \) and \( q_i \), and move to \( p_{i+1} \) and \( q_{j+1} \), respectively.

The above process is designed based on the following observations:

(i) Let \([a_i, b_j]\) and \([a_j, b_j]\) be the intervals for two nodes \( v_i \) and \( v_j \) of \( T \), respectively. If \( v_i \) and \( v_j \) are on the same path, then \( a_i \in [a_j, b_j] \) or \( a_j \in [a_i, b_i] \). If \( v_i \) and \( v_j \) are not on the same path, then \( a_i \geq b_j \) or \( a_j \geq b_i \). In addition, two intervals coming from two different interval sequences can be identical.

(ii) Initially each \( A_v \) contains only one interval and is trivially sorted. Then, when we merge a sorted interval sequence into another sorted interval sequence as above, the result interval sequence must also be sorted.

The formal description of the above process is given below.

**Algorithm interval-sequence-merge\((A_1, A_2)\)**

**Input:** \( A_1 \) and \( A_2 \) - two linked lists associated with \( v_1 \) and \( v_2 \).

**Output:** \( A \) - modified \( A_1 \), containing all the interval in \( A_1 \) and \( A_2 \) with all the subsumed intervals removed.

**begin**

1. \( p \leftarrow \text{first-element}(A_1) \); let \( p = [a, b] \);
2. \( q \leftarrow \text{first-element}(A_2) \); let \( q = [a', b'] \)
3. **while** \( p \neq \text{nil} \) **do**
4. **while** \( q \neq \text{nil} \) **do**
5. **if** \( (a \geq b') \) **then**
6. \{ insert \( q \) into \( A_1 \) before \( p \);
7. \( q \leftarrow \text{next}(q) \); (*next\( q \) represents the interval next to \( q \) in \( A_2 \)*)
8. **else if** \( (a \in [a', b']) \)
9. **then** \( p' \leftarrow p \); (*\( p \) is subsumed by \( q \); remove \( p \) from \( A_1 \)*)
10. **else if** \( (a' \geq b) \)
11. **then** \( [p' \leftarrow \text{next}(p)]; \) (*next\( p' \) represents the interval next to \( p' \) in \( A_1 \)*)
12. **else if** \( (a' = a \) and \( b = b' \))
13. **then** \( [q \leftarrow \text{next}(q)]; \) (*\( q \) is subsumed by \( p' \); move to the next element of \( q \).*)
14. **else** \( (a' \geq b) \)
15. **then** \( [p' \leftarrow \text{next}(p)]; \)
16. **else** \( (a = a' \) and \( b = b' \))
17. **then** \( [p' \leftarrow \text{next}(p); q' \leftarrow \text{next}(q)]; \})
18. **if** \( p = \text{nil} \) and \( q = \text{nil} \) **then**
19. \{ attach the rest of \( A_2 \) to the end of \( A_1 \); \}

**end**

The following example helps for illustration.

**Example 4.** Assume that \( A_1 = [3, 4][4, 5][7, 8] \) and \( A_2 = [2, 4][8, 9] \). Then, the result \( A \) by merging \( A_2 \) into \( A_1 \) is equal to \([2, 4][4, 5][7, 8][8, 9]\). Fig. 9 shows the entire merging process.

![Fig. 9. An entire merging process](image)

In each step, the \( A_1 \)-interval pointed by \( p \) and the \( A_2 \)-interval pointed by \( q \) are compared. In the first step, \([3, 4]\) in \( A_1 \) will be checked against \([2, 4]\) in \( A_2 \) (see Fig. 9(a)). Since \([3, 4]\) is subsumed by \([2, 4]\), it is removed from \( A_1 \). In the second step, \([4, 5]\) in \( A_1 \) will be checked against \([2, 4]\) in \( A_2 \). Since \([2, 4]\) is smaller than \([4, 5]\), it will be inserted into \( A_1 \) before \([4, 5]\) (see Fig. 9(b)). In the third step, \([4, 5]\) is smaller than \([8, 9]\) and we move to \([7, 8]\) in \( A_1 \) (see Fig. 9(c)). In the fourth step, \([7, 8]\) in \( A_1 \) is checked against \([8, 9]\) in \( A_2 \). Again, \([7, 8]\) is smaller than \([8, 9]\) (see Fig. 9(d)). We move to the next interval in \( A_1 \), and \( p \) becomes nil (see Fig. 9(e)). In this case, \([8, 9]\) will be attached to \( A_1 \) (see line 18 in Algorithm interval-sequence-merge\( ()\), forming the result \( A = [2, 4][4, 5][7, 8][8, 9]\) (see Fig. 9(e)).

In the following, we prove that for each \( L(v) \in G_{\text{com}} \), generated by using Algorithm interval-sequence-merge\( ()\) to merge the interval sequences of \( v \)'s children into \( v \)'s initial sequence (it contains
Lemma 1. Let $A_1$ and $A_2$ be two interval sequences sorted in increasing order. Let $A$ be the result obtained by merging $A_2$ into $A_1$ using Algorithm $\text{interval-sequence-merge}()$. Then, $A$ is also sorted increasingly.

Proof. During the execution of the algorithm, some intervals may be removed from $A_1$ and some intervals of $A_2$ may be inserted into $A_1$. Obviously, removing an interval from $A_1$ will not change the ordering of $A_1$. Let $q$ be an interval of $A_2$ inserted into $A_1$. It may be done at line 6 or at line 18. If it is done at line 6, there must be an interval $p$ in $A_1$ such that $q < p$. Consider the interval $p'$ before $p$. We have $p' < q$; otherwise, $q$ will be inserted before $p'$ or will not be inserted into $A_1$ at all. In this case, the lemma holds. If $q$ is inserted into $A_1$ by executing line 18, all the intervals in $A_1$ must be used up before the line 18 is carried out. We notice that in this case, all the intervals in $A_1$ are increasingly ordered and smaller than all the remaining intervals in $A_2$, which are originally in increasing order. Therefore, the lemma still holds.

Lemma 2. Let $A_1$ and $A_2$ be two interval sequences sorted in increasing order. Let $A$ be the result obtained by merging $A_2$ into $A_1$ using Algorithm $\text{interval-sequence-merge}()$. Let $v$ be a node in a subtree of $T$, which is rooted at some node labeled with an interval in $A_2$, then there must be an interval in $A$ such that the subtree rooted at it contains $v$.

Proof. Assume that $v$ is in a subtree rooted at $u$ labeled $[a, b]$ that appears in $A_2$. If $[a, b]$ appears in $A$, the lemma holds. Suppose that $[a, b]$ does not appear in $A$. In this case, there must exist $[a', b')$ in $A_1$, which subsumes $[a, b]$. Notice that $[a', b')$ cannot be subsumed by any interval in $A_2$ since it subsumes $[a, b]$. Otherwise, we will have an interval $[a'', b'']$ in $A_2$ such that $a' < [a'', b'']$. But we have $[a, b] < [a'', b'']$. It contradicts the fact that $[a', b')$ subsumes $[a, b]$ (i.e., $a \in [a', b')$ and so $a > a'$). In the latter case, we have $b > b''$. It also contradicts the fact that $[a', b')$ subsumes $[a, b]$ (from which we must have $b \leq b'$). Therefore, $[a', b')$ will appear in $A$. Since $v$ is in the subtree rooted at $[a, b]$, it must be in the subtree rooted at $[a', b')$. Thus, the lemma holds.

Lemma 3. Let $A_1$ and $A_2$ be two interval sequences sorted in increasing order. Let $A$ be the result obtained by merging $A_2$ into $A_1$ using Algorithm $\text{interval-sequence-merge}()$. If $v$ is a node in a subtree of $T$, which is rooted at some node labeled with an interval in $A_1$, then there must be an interval in $A$ such that the subtree rooted at it contains $v$.

Proof. Similar to Lemma 2.

Based on the merging operation discussed above, the interval sequences for all the nodes in $G_{com}$ can be computed along a reverse topological order. For instance, for the graph shown in Fig. 7(a), a possible reverse topological order is: $c \rightarrow g \rightarrow k \rightarrow d \rightarrow f \rightarrow e \rightarrow h \rightarrow a$. Along this order, we will generate an interval sequence for each node in $G_{com}$ as shown in Fig. 7(b).

Based on the above lemmas, the following lemma is easy to prove.

Lemma 4. Let $v$ be a node in $G_{com}$. Denote by $L(v)$ the interval sequence that is generated by using Algorithm $\text{interval-sequence-merge}()$ to merge the interval sequences of $v$’s children into $v$’s initial sequence along a reverse topological order of $G_{com}$. Then, for any descendant node $u$ (labeled $[a, b]$) of $v$ with respect to $G$, there exists an interval $[x, y]$ in $L(v)$ such that $a \geq x \leq y$. Let $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ be a reverse topological sequence of $G_{com}$. We prove the lemma by induction on the ordinal number $m$ in the reverse topological sequence.

Basis step. When $m = 1$, $v_1$ is a leaf node in $G_{com}$ and $L(v_1)$ contains only one interval associated with $v_1$. The lemma holds.

Induction hypothesis. Suppose that when $m < k$ the lemma holds. That is, for $i = 1, \ldots, k$, $L(v_i)$ contains all the intervals covering all the descendants of $v_i$ with respect to $G$.

Consider $m = k + 1$. According to the property of the reverse topological sequence, all the child nodes of $v_{k+1}$ must appear in $\{v_1, v_2, \ldots, v_k\}$. Then, according to Lemma 1, Lemma 2 and Lemma 3, as well as the induction hypothesis, we can see that $L(v_{k+1})$ contains all the interval covering all the descendants of $v_{k+1}$. It completes the proof.

Now we consider all the nodes of $G_{core}$, each node is associated with an interval sequence as shown in Fig. 10.

In this figure, we notice that the interval sequence for node $a$ is $[0, 12)$, which is the interval initially assigned to it. It is because when we merge all its children’s interval sequences into it, they are all absorbed into $[0, 12)$ (since all the intervals appearing in them are subsumed by $[0, 12)$).

Finally, in terms of Lemma 4, each interval sequence associated with a node of $G_{core}$ satisfies the condition specified in Definition 4. So all such interval sequences make up the core label of $G_{core}$.

In the following, we analyze the computational complexities of this process.

Lemma 5. The time complexity of Algorithm $\text{interval-sequence-merge}(A_1, A_2)$ is bounded by $O(\max(|A_1|, |A_2|))$.

Proof. During the execution of the algorithm, each interval in $A_1$ and $A_2$ is visited at most once.

Proposition 3. Let $G$ be a directed graph. Then, the size of its core label is bounded by $O(s \min\{b, s\})$. That is,

$$\sum_{v \in G_{core}} |L(v)| = O(s \min\{b, s\}),$$

where $b$ is the number of the leaf nodes of $T$, and $s$ is the number of the start nodes of all non-tree edges in $G$.

Proof. First, we claim that $|L(v)|$ for $v \in G_{core}$ is bounded by $b$.

Assume that $L(v)$ contains $b + 1$ intervals that are different from each other. Then, there must exist two intervals $p$ and $q$ so that $p$...
subsumes $q$ or vice versa. Therefore, one of them will be removed. Obviously, $|L(v)|$ for $v \in G_{core}$ is also bounded by $s$. So $|L(v)|$ for $v \in G_{core}$ is bounded by $\min\{b, s\}$. This completes the proof. \hfill \qed

**Proposition 4.** The time for generating the core label for $G$ is bounded by $O(t \min\{b, s\})$.

**Proof.** We consider the time for generating interval sequences for the nodes in $G_{com}$. In terms of Lemma 5 and the proof of Proposition 3, the time for this task can be estimated as follows:

$$O\left(\sum_{v \in G_{com}} d_v \cdot \min\{b, s\}\right) = O(\beta \min\{b, s\}),$$

where $d_v$ represents the outdegree of $v$ in $G_{com}$, and $\beta$ is the number of the edges in $G_{com}$.

In terms of Proposition 2, $\beta = O(t)$. Therefore, the time for generating interval sequences for the critical nodes in $G_{core}$ is bounded by $O(t \min\{b, s\})$. \hfill \qed

### 3.3.2 Non-tree labeling

Based on the core label of $G$, we assign non-tree labels to nodes, which would enable us to answer reachability queries quickly. Find a spanning tree $T$ in $G$. Let $v$ be a node in $T$. Consider the set of all the critical nodes in $T[v]$, denoted $C_v$. We denote by $v'$ a critical node in $C_v$ which is closest to $v$. We further denote by $v^*$ the lowest ancestor of $v$ (in $T$), which has a non-tree incoming edge. The following two lemmas are critical to our non-tree labeling method.

**Lemma 6.** Any critical node in $C_v$ appears in $T[v']$.

**Proof.** Assume that there exists a critical node $u$ in $C_v$ which does not appear in $T[v']$. Let $u_1$, ..., $u_k$ be all the critical nodes in $T[v']$. Consider the lowest common ancestor node of $u$, $u_1$, ..., $u_k$. It must be an ancestor node of $v'$, which is closer to $v$ than $v'$, contradicting the fact that $v'$ is the closest critical node (in $T[v]$) to $v$. \hfill \qed

**Lemma 7.** Let $u$ be a node, which is not an ancestor of $v$ in $T$; but $v$ is reachable from $u$ via some non-tree edges. Then, any way for $u$ to reach $v$ must be through $v^*$.

**Proof.** This can be seen from the fact that any node which reaches $v$ via some non-tree edges is through $v^*$ to reach $v$. \hfill \qed

Let $V_{core} = \{v_1, ..., v_g\}$. We store the core label of $G$ as a list: $s_1 = L(v_1), ..., s_g = L(v_g)$ (see Fig. 12(a) for illustration). Then, we define a function $\phi : V_{core} \rightarrow [1, ..., g]$ such that for each $v \in V_{core}$ $\phi(v) = i$ iff $s_i = L(v)$.

**Definition 5 (non-tree labels)** Let $v$ be a node in $G$. The non-tree label of $v$ is a pair <$x, y>$, where

- $x = i$ if $v'$ exists and $\phi(v') = i$. If $v'$ does not exist, let $x$ be the special symbol “-“.
- $y = [a, b]$ if $v^*$ exists and labeled $[a, b]$. If $v^*$ does not exist, let $y$ be “-“.

**Example 5.** Consider $G$ and $T$ shown in Fig. 1. The core label of $G$ with respect to $T$ is shown in Fig. 12(a). The values of the corresponding $\phi$-function are shown in Fig. 12(b).

Fig. 13 shows both the tree labels and the non-tree labels. For instance, the non-tree label of node $r$ is <3, - > because (1) $r' = e$; (2) $\phi(r') = \phi(e) = 3$ (see Fig. 12(b)); and (3) $r^*$ does not exist. Similarly, the non-tree label of node $f$ is <4, [6, 9)>.

Special attention

$$s_1.L(a) = [0, 12) \quad \phi(a) = 1$$

$$s_2.L(h) = [2, 4)[4, 5)[6, 9)(9, 12) \quad \phi(b) = 2$$

$$s_3.L(i) = [2, 4)[4, 5)(6, 9) \quad \phi(c) = 3$$

$$s_4.L(j) = [3, 4)[4, 5)](7, 8) \quad \phi(d) = 4$$

$$s_5.L(k) = [3, 4)[4, 5) \quad \phi(e) = 5$$

$$s_6.L(l) = [2, 4)(8, 9) \quad \phi(f) = 6$$

Fig. 12. Core label of $G$

should be paid to the non-tree label of node $e$: <3, [6, 9)>. First, we note that $e$ is $e$ itself. So $\phi(e) = \phi(e) = 3$. Furthermore, $e^*$ is also $e$ itself. Therefore, the tree label of $e^*$ is in fact the tree label of $e$ itself.

Fig. 13. Graph with non-tree labeling

**Proposition 5.** Assume that $u$ and $v$ are two nodes in $G$, labeled ($[a_1, b_1)$, <$x_1$, $y_1>$) and ($[a_2, b_2)$, <$x_2$, $y_2>$), respectively. Node $v$ is reachable from $u$ iff one of the following conditions holds:

(i) $[a_2, b_2)$ is subsumed by $[a_1, b_1)$, or

(ii) There exists an interval $[a, b]$ in $s_{x_1}$ such that $[a_2, b_2)$ is subsumed by $[a, b]$.

**Proof.** The proposition can be derived from the following two facts:

1. $v$ is reachable from $u$ by tree edges iff $[a_2, b_2)$ is subsumed by $[a_1, b_1)$.

2. In terms of Lemma 6, $v$ is reachable from $u$ via non-tree edges iff $v'$ exists and its interval sequence contains an interval $[a, b)$ which subsumes $[a_2, b_2)$. Furthermore, in terms of Lemma 7, $[a_2, b_2)$ subsumes $[a_2, b_2)$ iff $v^*$ exists and its interval is subsumed by $[a, b)$. \hfill \qed

Now we consider node $c$ and $e$ in the graph shown in Fig. 13. To check whether node $c$ labeled (2, 4), <$-, [2, 4)> is a descendant of node $e$ labeled ([6, 9), <3, [6, 9)>), we will first check whether $2 \in [6, 9)$. Since $2 \in [6, 9)$, we will check whether there is an interval in $L(e) = [2, 4)[4, 5)[6, 9)$ (note that $\phi(e) = 3$), which subsumes $[2, 4)$. Since $[2, 4)$ in $L(e)$ subsumes $[2, 4)$, we know that node $c$ is reachable from node $e$.

Finally, we notice that each interval sequence in the core table of $G$ contains only the intervals not on the same path in $T$ and they are also increasingly ordered. Therefore, to check a given interval is subsumed by an interval in $L(v)$ for some node $v$, we need only $O(\log |L(v)|)$ time. But $|L(v)|$ is bounded by $\min\{b, s\}$, so we require only $O(\log \min\{b, s\})$ time for reachability checking.

**Proposition 6.** Let $v$ and $u$ be two nodes in $G$. It needs $O(\log \min\{b, s\})$ time to check whether $u$ is reachable from $v$ via non-tree edges or vice versa.
3.4 Graph labeling: Core-II

We can store the core label of $G$ as a $d \times g$ boolean matrix $M$, where $d$ is the number of the end nodes of all non-tree edges and $g$ the number of the nodes in $G_{core}$.

Let $u_1, u_2, ..., u_d$ be all the end nodes of the non-tree edges. Let $v_1, v_2, ..., v_g$ be all the nodes in $G_{core}$. Assign each $u_i$ an index, denoted $\text{index}(u_i)$ (i.e., $u_1, u_2, ..., u_d$ will be assigned contiguous integers, starting from 0). Assign each $v_j$ an index, denoted $\text{index}(v_j)$. An entry $M[\text{index}(u_i), \text{index}(v_j)]$ is set to 1 if there exists an interval $[a', b')$ in $L(v_j)$ such that for $u_i$’s interval $[a, b]$ we have $a \in [a', b')$; otherwise, it is set to 0. For example, according to the core label shown in Fig. 12(a), we will establish a matrix as shown below:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Obviously, we need $O(d \cdot g \cdot \log(\min(b, s) ))$ time to generate $M$. Since $g = O(s)$ (see Proposition 2), $O(d \cdot g \cdot \log(\min(b, s))) = O(d \cdot s \cdot \log(\min(b, s)))$. Also, the size of $M$ is $O(d \cdot s)$. But it takes only a constant time to check whether a node is reachable from another node via non-tree edges.

4. CONCLUSION

In this paper, a new approach is proposed to compress transitive closure. Its main idea is to recognize a subset of nodes in $G$ and assign them labels in such a way that the reachability via non-tree edges can be determined by checking such labels only. The approach consists of two schemes, called Core-I labeling and Core-II labeling, respectively. For a graph $G$ with $n$ nodes and $e$ edges, the Core-I labeling scheme needs $O(n + e + t \cdot \min(b, s))$ labeling time, where $b$ is the number of the leaf nodes of a spanning tree (or a spanning forest) of $G$, $t$ is the number of non-tree edges, and $s$ is the number of the start nodes of all non-tree edges in $G$. The space overhead and the query time are bounded by $O(n + s \cdot \min(b, s))$ and $O(\log(\min(b, s)))$, respectively. The Core-II labeling scheme has constant query time, but needs $O(n + e + s \cdot \min(b, s) + d \cdot s \cdot \log(\min(b, s)))$ time, where $d$ is the number of the end nodes of all non-tree edges in $G$. The space overhead of Core-II is bounded by $O(n + d \cdot s)$. In general, we have $d \leq \min(n, t)$ and $s \leq \min(n, t)$.

REFERENCES


[23] H. Wang, H. He, J. Yang, P.S. Yu, and J. X. Yu, Dual Labeling: Answering Graph Reachability Queries in Constant time, in Proc. of Int. Conf. on Data Engineering, Atlanta, USA, April -8 2006.
