Recursive Graph Deduction and Reachability Queries

Yangjun Chen
Dept. Applied computer Science
University of Winnipeg, Winnipeg, Manitoba, Canada, R3b 2E9

Abstract - In this paper, we discuss an adjustable strategy for the transitive closure compression to support graph reachability queries, asking whether a given node u in a directed graph G is reachable from another node v through a path. The main idea behind it is to precompute the reachability between every pair of nodes – in other words, to compute and store the transitive closure (TC) 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(|V|^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 a new method to compress transitive closures to support reachability queries. The main idea behind it is to find a series of graph deductions G_i(V_i, E_i) (= G_i) G_1(V_1, E_1), ..., G_i(V_i, E_i) with n_i > n_{i+1} (i = 0, ..., k - 1), where n_0 = |V|. Each node v will be associated with an interval sequence \{ α^u_v, β^v_u \} (j ≤ k - 1) with each \{ α^u_v, β^v_u \} used to check reachability in G_i. Together with a subgraph structure (called the core graph of G) and a small matrix (called the core matrix), the reachability checking can be done in O(k) time. The size of the compressed transitive closure is bounded by O(kn + bkn_k√b_k), where b_k is the width of G_k, defined to be the size of a largest node subset U of G_k such that for every pair of nodes u, v ∈ U, there does not exist a path from u to v or from v to u in G_k. The time for generating such a compressed transitive closure is bounded by O(e_0 + kn_0 + n_k^2 + b_kn_k√b_k), where e_0 = |E_0|. For different applications, we can adjust k to different constants to get effective space reduction, but still have constant query time.

Keywords: recursive graph deduction, transitive closure, reachability queries

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 recursive queries [7, 12, 26, 27], object-oriented databases [15], as well as XML query processing, transportation network, internet traffic analyzing, semantic web, computer vision, and metabolic network [28], 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(|V|^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 a new method to compress transitive closures to support reachability queries. The main idea behind it is to find a series of graph deductions G_i(V_i, E_i) (= G_i) G_1(V_1, E_1), ..., G_i(V_i, E_i) with n_i > n_{i+1} (n_i = |V|, i = 0, ..., k - 1), and associate each node v with an interval sequence \{ α^u_v, β^v_u \} (j ≤ k - 1) with each \{ α^u_v, β^v_u \} used to check reachability in G_i. In addition, a subgraph G_core (called the core graph of G) and a small matrix M_core (called the core matrix of G) are constructed. To check whether node u is reachable from node v through a path in G, we will search two paths in G_core:

\[ v_0 = v \rightarrow v_1 \rightarrow ... \rightarrow v_j \] (0 ≤ j ≤ k)
\[ u_0 = u \rightarrow u_1 \rightarrow ... \rightarrow u_j. \]

Along these two paths, we first check whether \( α^v_{v_j} \in \{ α^u_v, \beta^v_u \} \). If it is the case, u must be a descendant of v. Otherwise, we traverse to v_1 and u_1, respectively; and check \( α^v_{v_j} \) against \( \{ α^u_v, \beta^v_u \} \). We repeat this process until we meet v_j and u_j with one of the following conditions satisfied:
1. \( j < k \) but we cannot further traverse from v_j or from u_j to a next node.
2. \( j = k \).

In case (1), we report that u is reachable from v if \( \alpha^v_j \in \{ \alpha^u_v, \beta^v_u \} \). Otherwise, u cannot be a descendant of v. In case (2), we need to check M_core to know whether u_k is reachable from v_k. The reachability of u_k from v_k implies the reachability of u from v.

Obviously, the above process needs only O(k) time to check reachability. But the space overhead is dramatically reduced to O(\( \sum_{j=0}^{k} n_j + b_kn_k \)) ≤ O(kn_0 + b_kn_k), where b_k is the width of G_k, defined to be the size of a largest node subset U of G_k such that for every pair of nodes u, v ∈ U, there does not exist a path from u to v or from v to u in G_k. It is a biased trade-off of time for space. While the query time increases linearly, the space overhead decreases quadratically (in the sense that both b_k and n_k are reduced at each step of graph deduction.) The time for generating a compressed transitive closure is bounded by O(e_0 + \( \sum_{j=0}^{k} n_j^2 + b_kn_k\sqrt{b_k} \)) ≤ O(e_0 + kn_0 + n_k^2 + b_kn_k\sqrt{b_k}), where e_0 = |E_0|. More importantly, our
method provides a flexible strategy to compress transitive closures. For different applications, we can set \( k \) to be different constants to get effective space deduction, but with constant query time.

The rest of the paper is organized as follows. In Section 2, we review some related work. In Section 3, we show a basic method for the TC compression based on a kind of graph decomposition, which will be used in our strategy. In Section 4, we discuss our first graph deduction. Section 5 is devoted to the recursive graph deduction. The paper concludes in Section 6.

2 Related work

In the past two decades, many interesting labeling-based methods have been proposed to speed up the 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 [19], Wang et al. discussed an interesting approach, called Dual-I, for sparse graphs \( G(V, E) \). Its space overhead is bounded by \( O(n + r) \) and can be produced in \( O(n + e + r') \) time, where \( r \) is the number of non-tree edges. The query time is \( O(1) \). As a variant of Dual-I, one can also store a matrix based on the so-called 2-hop viewing, which can reduce the space overhead from a practical viewpoint, but increases the query time to \( \log n \). 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 \( h_1 h_2 \). The main theoretical barrier of this method is that finding a 2-hop cover of minimum size is an NP-hard problem.

- Strategies for non-sparse graphs

In [13], Jagadish suggested an interesting method to decompose a DAG (directed acyclic graph) into node-disjoint chains. Then, each node \( v \) is assigned an index \( (i, j) \), where \( i \) is a chain number, on which \( v \) appears, and \( j \) indicates \( v \)'s position on the chain. These indexes can be used to check reachability efficiently. For this method, the space overhead and the query time are bounded by \( O(kn) \) and \( O(1) \), respectively, where \( k \) is the number of chains. However, to find a minimized set of chains for a graph, Jagadish's algorithm needs \( O(n^2) \) time (see page 566 in [13]). The method discussed in [8, 9] greatly improves Jagadish's method. It uses only \( O(n^2 + bn) \) time to decompose a DAG into a minimum set of node-disjoint chains, where \( b \) represents \( G \)'s width. Its space overhead is \( O(bn) \). In [9], the concept of the so-called general spanning tree is introduced, in which each edge corresponds to a path in \( G \). Based on this, the real space requirement becomes smaller than \( O(bn) \), but the query time increases to \( \log b \).

In [1], Agrawal et al. proposed a method based on interval labeling. As with the dual labeling, this method first figures out a spanning tree \( T \) and assign to each node \( v \) in \( T \) an interval \((a, b)\), where \( b \) is \( v \)'s postorder number (which reflects \( v \)'s relative position in a postorder traversal of \( T \)); and \( a \) is the smallest postorder number among \( v \) and \( v \)'s descendants with respect to \( T \) (i.e., all the nodes in \( T[v] \), the subtree rooted at \( v \)). Another node \( u \) labeled \((a', b')\) is a descendant of \( v \) (with respect to \( T \)) iff \( a \leq b < b' \). This idea originates from Schubert et al. [21]. In a next step, each node \( v \) in \( G \) will be assigned a sequence \( L(v) \) of intervals such that another node \( u \) in \( G \) with interval \((x, y)\) is a descendant of \( v \) (with respect to \( G \)) iff there exists an interval \((a, b)\) in \( L(v) \) such that \( a \leq y < b \). The length of a sequence associated with a node in \( G \) is bounded by \( O(\kappa') \), where \( \kappa' \) is the number of the leaf nodes in \( T \). So the time and space complexities are bounded by \( O(\kappa' e) \) and \( O(\kappa' n) \), respectively. The querying time is bounded by \( O(\log \kappa') \). In the worst case, \( \kappa' = O(n) \).

3 About TC compression

In this section, we briefly show the method discussed in [8, 9] to compress TC based on a graph decomposition, by which \( G \) is not deducted, but directly decomposed into a minimum set of node-disjoint chains. On a chain, if node \( v \) is above node \( u \), then there is a path from \( v \) to \( u \) in \( G \). Without loss of generality, we assume that \( G \) is acyclic (i.e., \( G \) is a DAG). If not, we will find all the strongly connected components (SCCs) of \( G \) and collapse each of them into a representative node. Obviously, each node in an SCC is equivalent to its representative node as far as reachability is concerned. This process takes \( O(e) \) time using Tarjan’s algorithm [18].

For illustration, consider the DAG shown in Fig. 1(a). Its transitive closure (stored as a 0-1 matrix) is shown in Fig. 1(b). Obviously, it requires \( O(n^3) \) space.

![Fig. 1. A DAG and a matrix representing its transitive closure](image)

Using the algorithm discussed in [8, 9], however, we can always decompose a DAG into a minimum set of node-disjoint chains, as illustrated in Fig. 2(a), in which the 1st, 2nd and 3rd chains are in fact three paths. But the 4th is a non-trivial chain (i.e., it is not a path since node \( a \) is connected to node \( i \) through a path of length 2, instead of an edge.) The fifth chain contains only a single node. We also remark that the width of the graph is 5 since there exists a subset \( U = \{b, f, g, i, j\} \), in which each pair of nodes are not connected. So it
is not possible to decompose the graph into a set with fewer chains.

We can then assign an index to each node in $G$ as follows:
1. Number each chain and number each node on a chain.
2. The $j$th node $v$ on the $i$th chain will be assigned a pair $(i, j)$ as its index, denoted $index(v)$.

In addition, each node $v$ on the $i$th chain will be associated with an index sequence of length $b - 1$: $(i, j_1) \ldots (i, j_b)$ such that any node with index $(x, y)$ is a descendant of $v$ if $x = i$ and $y \geq j$ or $x \neq i$ but $y \geq j_0$, where $b$ is the number of the node-disjoint chains, equal to $G$'s width [8]. (Here, a node is considered to be an ancestor of itself.)

### Basic definitions

Let $G$ be a directed graph. We use $V(G)$ and $E(G)$ to represent its node set and edge set, respectively. It is well known that the preorder traversal of $G$ introduces a spanning tree (forest) $T$. With respect to $T$, $E(G)$ can be classified into four groups:

- **tree edges ($E_{tree}$):** edges appearing in $T$.
- **cross edges ($E_{cross}$):** any edge $(u, v)$ such that $u$ and $v$ are not on a path in $T$.
- **forward edges ($E_{forward}$):** any edge $(u, v)$ not appearing in $T$, but there exists a path from $u$ to $v$ in $T$.
- **back edge ($E_{back}$):** any edge $(u, v)$ not appearing in $T$, but there exists a path from $v$ to $u$ in $T$.

All cross, forward, and back edges are referred to as non-tree edges. If $G$ is a DAG, we do not have back edges since a back edge implies a cycle.

For illustration, consider the DAG shown in Fig. 1(a) once again. For it, we may find a spanning tree as shown in Fig. 3, in which each solid arrow stands for a tree edge while a dashed arrow for a non-tree edge.

![Fig. 3. A spanning tree and intervals](image)

As in [19], we can assign each node $v$ in $T$ an interval $[\alpha, \beta]$, where $\alpha$ is $v$'s preorder number (denoted $pre(v)$) and $\beta$ - 1 is equal to the largest preorder number among all the nodes in $T[v]$. So another node $u$ labeled $[\alpha', \beta']$ is a descendant of $v$ (with respect to $T$) iff $\alpha' \in [\alpha, \beta]$ [19], as illustrated in Fig. 3. If $\alpha' \in [\alpha, \beta]$, we say $[\alpha', \beta']$ is subsumed by $[\alpha, \beta]$.

Let $T$ be a spanning tree of $G$. Let $e = (u, v)$ be an edge. By $G \setminus e$ we denote a graph obtained from $G$ by removing one of the following two operations:
1. If $e$ is a forward edge, remove $e$ from $G$.
2. If $e$ is a tree edge (an edge in $T$), remove $e$ and $v$; and any edge incident to $v$ becomes incident to $u$.

Especially, for a tree edge $e$, $Te$ is still a spanning tree of $G \setminus e$, based on which we define the following concept.

**Definition 1 (graph deduction)** A graph $G_1$ is a deduction of $G$ if there are graphs $G^{(0)}, \ldots, G^{(n)}$ and forward or tree edges $e_i \in G^{(i)}$ such that $G^{(0)} = G, G^{(n)} = G_1,$ and $G^{(i+1)} = G^{(i)} \setminus e_i$.

The graph deduction is a special kind of graph minors [10]. Our purpose is to find a graph deduction $G_1$ of $G$ such that:

i) the reachability matrix of $G_1$, $M_{G_1}$, is smaller than $M_G$;

ii) the reachability of $G$ can be checked with help of $M_{G_1}$;

iii) the query time is increased only by a constant.

In the following, we discuss how such a $G_1$ can be found.

### Critical nodes and critical subgraphs

Our main idea is to recognize a subset of nodes, the so-called critical nodes, which enable us to construct a graph deduction satisfying all the above properties.

We denote by $E'$ the set of all cross edges. Denote by $V'$ the set of all the end points of the cross edges. That is, $V' =$
Based on the concept of critical nodes, we can now define our letting a node with respect to graph deduction Let have altogether 11 anti-subsuming subsets as shown in Fig. 4(b).

As an example, consider the spanning tree shown by the solid arrows in Fig. 3. With respect to this spanning tree, we have altogether 11 anti-subsuming subsets as shown in Fig. 4(b).

Definition 2 (anti-subsuming subset) A subset \( S \subseteq V_{\text{start}} \) is called an anti-subsuming set iff \(|S| > 1\) and no two nodes in \( S \) are related by an ancestor-descendant relationship with respect to \( T \).

Definition 3 (critical nodes) 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 \( v_1, v_2, ..., v_k \).

For example, in the spanning tree shown in Fig. 3, 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. We denote by \( V_c \) all the critical nodes. Based on the concept of critical nodes, we can now define our graph deduction \( G_1 \), called a critical subgraph of \( G \).

Let \( T \) be a spanning tree of \( G \). Denote by \( T_r \) a reduction of \( T \) obtained by removing all those nodes \( v \notin V_c \cup V_{\text{end}} \). Deleting a node \( v \) entails connecting \( v \)'s parent to each of \( v \)'s children. So, removing a node in this way corresponds to the elimination of a tree edge.

For example, for the spanning tree \( T \) shown in Fig. 3, its \( T_r \) is a tree shown in Fig. 5. It is obtained by removing the nodes \( b, r, i \), and \( j \) one by one. Note that none of them belongs to \( V_c \cup V_{\text{end}} \). (Remember that \( V_c = \{a, d, e, f, g, h\} \) and \( V_{\text{end}} = \{c, d, e, g, k\} \).

Definition 4 (critical subgraph) Let \( G(V, E) \) be a DAG. Let \( T \) be a spanning tree of \( G \). The critical subgraph \( G_c \) of \( G \) with respect to \( T \) is graph with node set \( V(T_r) \) and edge set \( E(T_r) \cup E_{\text{cross}} \).

In Fig. 6(a), we show a critical subgraph of the graph shown in Fig. 3, with respect to the corresponding spanning tree.

Again, using the algorithm discussed in [8, 9], we can decompose it into a set of node-disjoint chains as illustrated in Fig. 6(b); and construct a reachability matrix as shown in Fig. 6(c), which is much smaller that the matrix shown in Fig. 1(b), and even smaller than that shown in Fig. 2(b). However, it seems that such a matrix can only be used to check the reachability between the nodes in \( G_c \).

The question is: can such a matrix also be used to check the reachability between the nodes in \( G \)? In the next section, we answer this question.

4.3 Evaluation of reachability queries

For any two nodes \( u, v \) appearing on a path in \( T \), their reachability can be checked using their associated intervals. However, if they are not on the same path, we have to use \( G \)'s reachability matrix \( M_G \) to check whether \( u \) is reachable from \( v \) through a path in \( G \), or vice versa.

Now what we have is \( M_{G_c} \), rather than \( M_G \). How can we check the reachability from \( v \) to \( u \) in \( G \)?

To do this, we need another concept, the so-called anchor nodes.

Definition 5 (anchor nodes) Let \( G \) be a DAG and \( T \) a spanning tree of \( G \). Let \( v \) be a node in \( T \). Denote by \( C_v \) all the critical nodes in \( T[v] \). We associate two anchor nodes with \( v \) as below.

i) A node \( u \in C_v \) is called an anchor node (of the first kind) of \( v \) if \( u \) is closest to \( v \). \( u \) is denoted \( v^* \).

ii) A node \( w \) is called an anchor node (of the second kind) of \( v \) if it is the lowest ancestor of \( v \) (in \( T \)), which has a cross incoming edge. \( w \) is denoted \( v^{**} \).

For example, in the graph shown in Fig. 3, \( r^* = e \). It is because node \( e \) is critical and closest to node \( r \) in \( T[r] \). But \( r^{**} \) does not exist since it does not have an ancestor which has a cross incoming edge. In the same way, we find that \( e^* = e^{**} = e \). That is, both the first and second kinds of anchor nodes of \( e \) are \( e \) itself. We can easily recognize the anchor nodes for all the other nodes in the graph.

The following two lemmas are critical to our non-tree labeling method.

Lemma 1 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 \( v \).
Lemma 2 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 cross edges is through v** to reach v.

Definition 6 (non-tree labels) Let v be a node in G. The non-tree label of v is a pair c<, y>, where
- x = v* if v* exists. If v* does not exists, let x be the special symbol “−”.
- y = v** if v** exists. If v** does not exist, let y be “−”.

The purpose of the non-tree labeling is to use the anchor nodes u** and v* to check the reachability of u from v through cross edges.

Example 1 Consider G and T shown in Fig. 3. The non-tree labels of the nodes are shown in Fig. 7.

![Fig. 7. Non-tree labels](image)

In this figure, we can see that the non-tree label of node r is c<, −> because (1) r* = e; and (2) r** does not exist. Similarly, the non-tree label of node f is cf, e>. It is because f* is f itself; but f** is e (see Fig. 7).

Especially, we notice that node r and node d are not on the same path in T. But d is a descendant of r. Such a reachability has to be checked by using their anchor nodes. In fact, we have d** = d, index(d) = (1, 4), r* = e, and (e, 1) = 2 < 4 (see Fig. 6(c)), which shows that d** is a descendant of r*. By the following proposition, this indicates that d is a descendant of r.

Proposition 3 Assume that u and v are two nodes in G, labeled (A<, B>) and (C<, D>). Here, (A, B) (i = 1, 2) is the tree label while c<, y> (i = 1, 2) is the non-tree label. Node u is reachable from v iff one of the following holds:

(i) (A<, B>) is subsumed by (A<, B>) (i.e., A< = A<, B>)
(ii) index(y) = (x, y), and (x, y) ≤ y.

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

1) u is reachable from v through tree edges iff (A<, B>) is subsumed by (A<, B>);
2) (A<, B>) is reachable from v via non-tree edges iff u** exists and its index index(u**) is a pair (x, y) such that (v*, x) ≤ y.

Now we analyze the time complexity of the whole process. It mainly comprises three parts:

i) the cost to find a spanning tree of G(V, E);
ii) the cost to find all the critical nodes, and establish Gc; and
iii) the cost to decompose the critical graph of G into a minimum set of node-disjoint chains.

The cost of the first part is O(|V| + |E|). The second part is also bounded by O(|V| + |E|). According to [8, 9], the third part is bounded by O(m^2 + bm√b), where m = |V(G)|, and b is the width of Gc.

Proposition 4 Let G be a DAG with n nodes and e edges. Then, the labeling time of our method is bounded by O(n + e + m^2 + bm√b).

Proof. See the above analysis.

5 Recursive graph deduction

Now we discuss the recursive graph deduction.

5.1 Recursive deduction

In the previous section, we showed how to reduce G to Gc. Especially, using T and MGi, we can check the reachability of G in constant time.

However, it can be observed that Gc itself can be further reduced, leading to a further decrement of space requirement. In fact, using the method discussed in 4.1 and 4.2, we can find a series of graph deductions:

G0 = G, G1, ..., Gk, (k ≥ 1)

where G0 is a critical subgraph of Gi+1 (i = 1, ..., k).

In order to construct such critical subgraphs, a series of spanning trees have to be established:

T0, T1, ..., Tk.

where each Ti is a spanning tree of Gi (i = 0, ..., k - 1), used to construct Gi+1.

Finally, similar to MGi, we will have a reachability matrix for Gi, called the core matrix of G and denoted MGi.

To check reachability efficiently, each node v in G will be associated with two sequences: an interval sequence and an anchor node sequence:

1) [(Aglomer), (B*)], ..., [(Aglomer), (B*)] (i ≤ k - 1)

where each [(Aglomer), (B*)] is an interval generated by labeling Ti;

2) [(x0, y0), ..., (xj, yj)],

where each xj is a pointer to an anchor node of the first kind (a node appearing in Gi+1) while each yj a pointer to an anchor node of the second kind (also, a node in Gi).

Fig. 8(a) helps for illustration.

![Fig. 8. Illustration for anchor nodes and a spanning tree](image)
second kind. For each node appearing in $G_r$, an interval and a pair of anchor pointers will be created. So, if node $v$ appears in $G_0 = G, G_1, ..., G_l$ for some $j \leq k$, it will be associated with two sequences of length $j$, as described above.

**Example 3** Denote by $G_0$ and $G_1$ the graph shown in Fig. 3 and the critical graph shown in Fig. 5(a), respectively. Fig. 8(b) shows a possible spanning tree $T_1$ of $G_1$, and the corresponding tree labels.

With respect to $T_1$, we do not have any anti-subsuming subset (see Fig. 9(a)). Thus, we have no non-trivial critical nodes. The critical graph $G_3$ is shown in Fig. 9(b), containing only one edge $(e, k)$. Tackling this edge as a chain (as shown in Fig. 9(c)), we generate a reachability matrix as shown in Fig. 9(d). The non-tree labels of the nodes in $G_1$ are shown in Fig. 10(a). The interval sequence and the anchor node sequence for each node of $G$ are shown in Fig. 10(b).

![Fig. 9. Anti-subsuming subsets, and reachability matrix](image)

![Fig. 10. Non-tree labels and sequences associated with nodes](image)

5.2 Evaluation of reachability queries

Now we discuss how to use the interval sequences and anchor node sequences to check reachability. First, we notice that the anchor node sequences imply a graph, called a core graph of $G$ and denoted $G_{core}$, in which there exists an edge $(u, v)$ iff there is an entry $<x, y>$ in the anchor node sequence associated with $u$ such that $x = v$, or $y = v$. The edge is labeled with $[i, *)$ or $[i, **)$, depending on whether $x = v$, or $y = v$, where $i$ indicates that $<x, y>$ is the $i$th entry in the anchor node sequence. In Fig. 11, we show the core graph corresponding to the anchor node sequences shown in Fig. 10(b).

![Fig. 11. A core graph](image)

In the graph, edge $(r, e)$ labeled with $[1, *)$ represents that $e$ is an anchor node (of the first kind) of $r$, which appears in $G_l$ while edge $(a, c)$ labeled with $[2, **)$ represents that $c$ is an anchor node (of the second kind) of $a$, appearing in $G_2$.

An edge with multiple labels represents several edges with different labels. For example, the edge $(e, e)$ (represented as a loop) labeled with $[1, *], [1, **)$, and $[1, **)$ stands for two edges with each going from $e$ to $e$, but labeled differently.

Remark that each node $v$ in $G_{core}$ is associated with an interval sequence $\{a_0^r, b_0^r\}, ..., \{a_i^r, b_i^r\}$ for some $x \geq 0$. In order to check whether $v$ is an ancestor of $u$, we will search two paths in $G_{core}$, starting from $v$ and $u$, respectively. The path starting from $v$, denoted $P_1$, contains only the edges labeled with $(i, *)$ while the path starting from $u$, denoted $P_2$, contains only the edges labeled with $(i, **)$. Each time we reach two nodes $v'$ and $u'$ through two edges labeled respectively with $(i, *)$ and $(i, **)$, we will check whether $\alpha_i^v, \beta_i^v$ subsumes $\alpha_i^u, \beta_i^u$. If it is the case, $v$ is an ancestor of $u$. Otherwise, we traverse along $P_1$ and $P_2$, reaching $v''$ and $u''$ through two edges labeled respectively with $(i + 1, *)$ and $(i + 1, **)$ and checking $\alpha_i^{v''}, \beta_i^{v''}$ against $\alpha_i^{u''}, \beta_i^{u''}$. We continue this process. After $j$ steps for some $j$, we will meet two nodes $v''''$ and $u''''$ such that $v'''''$ does not have an out-going edge labeled with $(j + 1, *)$ or $u'''''$ does not have an out-going edge labeled with $(j + 1, **)$. If $\alpha_i^{v''''}, \beta_i^{v''''}$ subsumes $\alpha_i^{u''''}, \beta_i^{u''''}$, $v''''$ is an ancestor of $u$. Otherwise, we will check whether $M_{core}(v''''' \leq z$, where $index(u''''') = (w, z)$.

**Example 4** Consider the graph shown in Fig. 3 once again. To check whether $g$ is an ancestor of $k$, we will explore two paths in the graph shown in Fig. 11, starting from $g$ and $c$, respectively. First, we check that $\alpha_1^g, \beta_1^g = \{8, 9\}$ against $\alpha_1^c, \beta_1^c = \{3, 4\}$ and find that $\{8, 9\}$ does not subsume $\{3, 4\}$ (see Fig. 10(b)). Then, we go from $g$ along an edge labeled with $(1, *)$ to $g$ itself; and from $k$ along an edge labeled with $(1, **)$ to $k$ itself (see Fig. 11). Now, we check that $\alpha_1^g, \beta_1^g = \{6, 8\}$ against $\alpha_1^c, \beta_1^c = \{5, 6\}$. Since $\{6, 8\}$ does not subsume $\{5, 6\}$, we will continue to explore the two paths labeled with $(1, *)$ and $(2, **)$, respectively, reaching two nodes $c$ and $k$ (see Fig. 11). Note that $index(k) = (1, 2)$ (see Fig. 9(d)). Since, $(c, 1) = 1 < 2$, $g$ is an ancestor of $k$.

**Proposition 4** Let $G$ be a DAG, and $G_0 = G, G_1, ..., G_l$ be a series of graph deduction as described above. Let $u$ and $v$ be two nodes in $G$. $u$ is reachable from $v$ through a path in $G$ iff there exist two paths in $G_{core}$:

$v_0 = v \rightarrow v_1 \rightarrow ... \rightarrow v_j \quad (0 \leq j \leq k)$

$u_0 = u \rightarrow u_1 \rightarrow ... \rightarrow u_j$

such that each $(v_{j,1}, v_j)$ is labeled with $(i, *)$, each $(u_{j,1}, u_j)$ is labeled with $(i, **)$, and one of the following two conditions is satisfied:

1. $j < k$, and $u_j$ is reachable from $v_j$ through a path in $T_j$; or
2. $j = k$, and $u_j$ is reachable from $v_j$ through a path in $G_k$.

**Proof.** If-part. We prove the if-part by induction on $k$.

**Basis step.** When $k = 0$, the proof is trivial.

**Induction hypothesis.** Assume that when $k = l$ the if-part holds. We consider the case when $k = l + 1$. If $j \leq l$, in terms
of the induction hypothesis, the if-part holds. Assume that \( j = l + 1 \). Since \( u_{i+1} \) is reachable from \( v_{i+1} \) through a path in \( G_{i+1} \), \( u_i \) must be reachable from \( v_i \) in \( G_i \) by Lemma 1 and 2. (Note that \( v_{i+1} \) is an anchor node of the first kind of \( v_i \) and \( u_{i+1} \) is an anchor node of the second kind of \( u_i \).) In terms of the induction hypothesis, \( u \) is reachable from \( v \).

Only-if-part. If \( u_0 = u \) is reachable from \( v_0 = v \), there will be a path in \( T_0 \) from \( v_0 \) to \( u_0 \) or \( u_1 \) is reachable from \( v_1 \) in \( G_1 \). Similarly, \( u_i \) is reachable from \( v_i \) in \( G_i \), there will be a path in \( T_i \) from \( v_i \) to \( u_i \), or \( u_2 \) is reachable from \( v_2 \) in \( G_2 \). Repeating this argument, we will get the proof.

The above proposition shows that to check whether \( u \) is reachable from \( v \), we need to search two paths in \( G_{core} \) and at each step to examine whether \( \alpha^i_0 \in \{ \alpha^i, \beta^i \} \). If \( i = k \), we have to check whether \( M_{core}(v_j, x) \leq y \), where \( index(u_j) = (x, y) \). Clearly, it needs \( O(k) \) time. The space complexity is easy to analyze. Since for each appearance of a node \( v \) in \( G_i \) an interval for \( v \) is created, the space overhead is bounded by \( O(k \sum_{j=0}^{k-1} n_j + b_k n_k) \), where \( n_i \) is the size of \( G_i \) and \( b_k n_k \) is the size of \( M_{core} \). According to [8, 9], the decomposition of \( G_k \) into a minimum set of node-disjoint paths needs \( O(n_k^2 + b_k n_k \sqrt{b_k}) \) time. So, the total labeling time is \( O(e_0 + \sum_{j=0}^{k-1} n_j + n_k^2 + b_k n_k \sqrt{b_k}) \).

6 Conclusion

In this paper, a new method is proposed to compress transitive closures to support reachability queries. Its main idea behind it is to find a series of graph deductions \( G_0(V_0, E_0) (= G), G_1(V_1, E_1), ..., G_k(V_k, E_k) \) with \( n_i > n_{i+1} (n_i = |V_i|), i = 0, ..., k - 1 \), and associate each node \( v \) with an interval sequence \( \{ \alpha^0_0, \beta^0_0 \}, ..., \{ \alpha^i, \beta^i \} (i \leq k - 1) \) with each \( \{ \alpha^i, \beta^i \} \) used to check reachability in \( G_i \). In addition, a subgraph \( G_{core} \) (called the core graph of \( G \)) and a small matrix \( M_{core} \) (called the core matrix of \( G \)) are constructed, based on which a reachability query can be answered in \( O(k) \) time. But the space for storing a compressed transitive closure is only \( k \) times larger than the original graph. For different applications, \( k \) can be set to different constants.

7 References


[19] 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.