H K . N C C P
International Journal of Intelligent Information and Management Science
ISSN: 2307-0692 *Volume 3, Issue 1, February 2014*

DAG Deduction and Decomposition: A New Way to Evaluate Reachability Queries

Haihui HUANG, Xin WEN, Zheng LUO, Huixue QIAO
College of Software, CQUPT, Chongqing, China

Abstract: Let $G(V, E)$ be a digraph (directed graph) with n nodes and e arcs. Digraph $G^* = (V, E^*)$ is the reflexive, transitive closure if $v \rightarrow u \in E^*$ iff there is a path from v to u in G. Efficient storage of G^* is important for supporting reachability queries which are not only common in graph databases, but also serve as fundamental operations used in many graph algorithms. A lot of strategies have been suggested based on the graph labeling, by which each node is assigned with certain labels such that the reachability of any two nodes through a path can be determined by their labels. Among them are interval labeling, chain decomposition, 2-hop labeling, and path-trees. However, due to the very large size of many real world graphs, the computational cost and size of labels using existing methods would prove too expensive to be practical. In this paper, we propose a new approach to deduct and decompose a graph into two graphs: a transitive core graph and a residue graph. Both are much smaller than the original graph. In this way, we transform any reachability query into two queries. One is over the transitive core graph and another is over the residue graph. While the former can always be evaluated in constant time, the latter can be done by using any existing method, but over a much smaller graph. We demonstrate both analytically and empirically the efficiency and effectiveness of our method.
Keyword: Directed Graphs; spanning Trees; Reachability Queries; Transitive Closure Compression.

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 evaluation of recursive queries in deductive databases, type checking in object-oriented databases, XML query processing, social network, transportation network, internet traffic analyzing, semantic web, and metabolic network [22], graph reachability is one of the most basic operations, and therefore needs to be efficiently supported.

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 as a boolean matrix M such that $M[i, j] = 1$ if there is a path from i to j; otherwise, $M[i, j] = 0$. Then, a reachability query can be answered in constant time. However, this requires $O(n^2)$ space, which makes it impractical for massive graphs, where $n = |V(G)|$. Another method is to compute the shortest path from u to v over a graph on demand. Therefore, it needs only $O(e)$ space, but with high query processing cost - $O(e)$ time in the worst case, where $e = |E(G)|$.

There is much research on this issue to reduce space overhead but still keep a constant query time, such as those discussed in [1, 2, 4, 5, 6, 8, 9, 10, 22]. All of them reduce the space requirement to some extent. But the worst space overhead is still in the order of $O(n^2)$. In the case of large graphs, they cannot be efficient.

In this paper, we investigate the problem from a different angle: to deduct and decompose G into several components such that the existing labeling techniques can be utilized for each smaller graph without sacrificing too much query time.

Concretely, we decompose G into two smaller graphs: a *transitive core graph* G_{core} and a *residue graph* G_r. When a query q is submitted, we will first evaluate q against G_{core}, by which two paths of constant length in G_{core} will be searched. If the query can be answered in this process, the task is done. Otherwise, a new query q' is formed and evaluated against G_r. This can be conducted by using any existing method. If we use the method discussed in [5], the whole query time of this method is bounded by $O(\kappa)$, where κ is the length of a path explored when evaluating q against G_{core}, bounded by a constant. Later we will see that $|E(G_{core})|$ is bounded by $O(kn)$. So the total space overhead is bounded by $O(\kappa n + n_r w_r)$, where n_r stands for the number of the nodes in G_r, and w_r for the width of G_r, defined to be the size of a largest node subset U of G_r such that for any pair of nodes $u, v \in U$ there is neither a path from u to v nor from v to u.

More importantly, it is a very flexible method. For different applications, we can control the graph decomposition, i.e., to set k to different constants, to get a trade-off of query time for space. We will show that it is a biased trade-off of time for space. While the query time increases linearly, the space overhead decreases qua-

draticaly, in the sense that both the number of the nodes and the width of G_r are decreased.

The remainder of the paper is organized as follows. In Section II, we review the related work. In Section III, we discuss the main step of our method to deduct and decompose a directed acyclic graph (*DAG*), based on which a transitive closure can be effectively compressed. In Section IV, we address an interesting problem on how to find spanning trees such that a graph can be deducted in as few steps as possible. In Section V, we show a recursive graph decomposition to generate a series of spanning trees which may share common arcs, from which the transitive core graph is established. Also, how to evaluate reachability queries over such a graph is discussed. In Section VI, we discuss the maintenance of compressed transitive closures. Section VII is devoted to the experiments. Finally, a short conclusion is set forth in Section VIII.

2. Related Work

In the past two decades, many interesting labeling-based strategies have been proposed to reduce both the precomputation time and storage cost with reasonable answering time. In the following, we review some representative ones.

2.1. Chain Decomposition Methods

In [8], Jagadish suggested a method to decompose a DAG into node-disjoint chains. On a chain, if node v appears above node u, there is a path from v to u in G. 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 with $O(\mu n)$ space overhead and O(1) query time, where μ is the number of chains. However, to find a set of chains for a graph, Jagadish's algorithm first finds a minimized set of node-disoint paths by solving a *min* flow problem, and then stitches some paths together to form a chain. This algorithm needs $O(n^3)$ time (see page 566 in [8]). In addition, the number μ of the produced chains is normally much larger than the minimal number of chains. In the worst case, μ is $O(n)$.

The method discussed in [5] greatly improves Jagadish's method. It needs only $O(n^2 + w^{1.5}n)$ time to decompose a DAG into a minimum set of node-disjoint chains, where w represents G's width. Its space overhead is $O(wn)$ and its query time is bounded by a constant. In [6], the concept of the so-called general spanning tree is introduced, in which each arc corresponds to a path in G. Based on this data structure, the real space requirement becomes smaller than $O(wn)$, but the query time increases to $\log w$.

2.2. Interval based Methods

In [1], Agrawal *et al.* proposed a method based on interval 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 $a \le b' < b$. This idea originates from Schubert *et al.* [1]. 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 $a \le y < b$. The length of such a sequence (associated with a node in G) is bounded by $O(l)$, where l is the number of the leaf nodes in T. So the time and space complexities are bounded by $O(le)$ and $O(ln)$, respectively. The querying time is bounded by $O(\log l)$. In the worst case, $l = O(n)$.

The method discussed in [22] can be considered as a variant of the interval based method, and called *Dual-I*, specifically designed for sparse graphs $G(V, E)$. As with Agrawal *et al.*'s, it first finds a spanning tree T, 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 $\boldsymbol{N}$ (called a *TLC* matrix) is maintained, where t is the number of non-tree arcs (arcs not appearing in T). 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 $\boldsymbol{N}(x_v, z_u) - \boldsymbol{N}(y_v, z_u) > 0$. The size of all labels is bounded by $O(n + t^2)$ and can be produced in $O(n + e + t^3)$ time. The query time is O(1). As a variant of *Dual-I*, one can also store $\boldsymbol{N}$ 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*.

2.3. 2-hop Labeling

The method proposed by Cohen *et al.* [4] labels a graph based on the so-called 2-*hop covers*. It is also designed 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_1h_2. Using this method to label a graph, the worst space overhead is in the order of $O(n)$. 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 [4], 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 $C_{in}(v) \cap C_{out}(v) \neq F$. Using this method, the overall label size is increased to

$O(n\sqrt{e}\log n)$. In addition, a reachability query takes $O(\sqrt{e})$ time because the average size of each label is above $O(\sqrt{e})$. The time for generating labels is $O(n^4)$.

2.4. Path-tree Decomposition

Recently, Jin *et al.* [9, 10] discussed a new method, by which a DAG G is decomposed into a set of node-disjoint paths. Then, a weighted directed graph G_w (called *path-graph* in [9]) is constructed, in which each node represents a path and there is an arc (i, j) if on path i there is a node connected to a node on path j. The weight associated with (i, j) is the number of such connections. Then, find a maximum spanning tree T_w (called path-tree) of G_w and label the nodes in T_w with an interval in a way similar to Agrawal *et al.*'s. Together with the labels assigned to the node on all the paths, the intervals can be used to check part of reachability. To be a complete strategy, each node v has to be associated with a set, denoted $R^c(v)$, such that all the descendants of v, which appear on a path are dominated by a node in $R^c(v)$. In the worst case, the size of $R^c(v)$ is bounded by l, the number of the leaf nodes of a spanning tree of G. Therefore, the space complexity of this method is $O(ln)$. The query time and the labeling time are bounded by $O(\log^2 l)$ and $O(le)$, respectively. Theoretically, both the space requirement and the query time of this method is worse than Agrawal's [1].

2.5. GRAIL

The method proposed by Yildirim *et al.* [23] is a light-weight indexing structure. It traverses G for several times to create an interval sequence for each node, used as a filter as follows. Let $L_u = L_u^1, \ldots, L_u^k$ and $L_v = L_v^1, \ldots, L_v^k$ be the interval sequences of u and v, respectively. If there exists i ($i \in \{1, \ldots, k\}$) such that $L_u^i \not\subset L_v^i$, u is definitely not a descendant of v. But if for all $i \in \{1, \ldots, k\}$ $L_u^i \subseteq L_v^i$, it cannot be determined whether u is a descendant of v, or *vice versa*. In this case, the whole G will be searched in the depth-first manner, but with the label sequences used to prune the search space. The labeling time of this method is bounded by $O(k(n + e)$. If k is chosen as a constant, the index size is proportional to $O(n)$ and can be established very fast. But in the worst case, the query time is $O(e)$ as if no index is established.

There are some other graph labeling methods, such as the method using signatures [20], *PE-Encoding* [3] and *PQ-Encoding* [24]. The idea of the signature-based method [20] is to assign to each node a signature (which is in fact a bit string) generated using a set of hash functions. The space complexity is $O(ln)$, where l is the length of a signature. But this encoding method suffers from the so-called signature conflicts (two nodes are assigned the same signature). Moreover, in the case of DAGs, a graph needs to be decomposed into a series of trees; and no formal decomposition was reported in that paper. The *PE*-Encoding [3] and the *PQ*-Encoding [24] are similar to the 2-hop labeling, but with higher computational complexities. The methods discussed in [15, 16] reduces 2-hop's labeling complexity from $O(n^4)$ to $O(n^3)$, but are still not applicable to massive graphs. The method proposed in [2] is a geometry-based algorithm to find high-quality 2-hop covers. It also improves the 2-hop labeling by avoiding the computation of transitive closures, which is required by Cohen's to find 2-hop covers. However, it has the same theoretical computational complexities as Cohen's method [4]. Finally, the method discussed in [21] is suitable only for planar graphs with $O(n\log n)$ labeling time and $O(n\log n)$ space. The query time is $O(1)$. Finally, the query evaluation mechanism of deductive databases can be adapted to handle this problem.

In the following table, we compare our labeling method with the representative approaches.

Table 1. Comparison of Strategies

	Query time	Labeling time	Space overhead
Graph traversal	$O(e)$	0	$O(e)$
Jagadish [8]	$O(1)$	$O(n^3)$	$O(\mu n)$
Interval-based[1]	$O(\log n)$	$O(ne)$	$O(\lambda n)$
Dual-I [22]	$O(1)$	$O(n + e + t^3)$	$O(n + t^2)$
Dual-II [22]	$O(\log t)$	$O(n + e + t^3)$	$O(n + t^2)$
2-hop [4]	$O(e^{1/2})$	$O(n^4)$	$O(n\log n)$
Matrix-based[25]	$O(1)$	$O(n^3)$	$O(n^2)$
Tree-path [9]	$O(\log^2\lambda)$	$O(\lambda e)$	$O(\lambda n)$
GRAIL [23]	$O(e)$	$O(ke)$	$O(kn)$
Chen [5]	$O(1)$	$O(n^2 + w^{1.5}n)$	$O(wn)$
ours	$O(k)$	$O(\kappa e + w_r^{1.5}n_r)$	$O(\kappa n + w_r n_r)$

Note that in the above table κ and k are two different constants.

In the worst case, both μ and λ are in the order of $O(n)$ and t is in the order of $O(e)$.

3.Graph Deduction

In this section, we discuss a new graph decomposition approach to compress transitive closures. First, we give some basic definitions related to spanning trees in Subsection *A*. Then, in Subsection *B*, we demonstrate our basic graph decomposition based on the concept of *critical nodes*, as well as a method for checking the reachability by using such a graph decomposition. Finally, we show how to efficiently recognize the critical nodes in a graph in Subsection *C*.

3.1. Basic Definition

Without loss of generality, we assume that G is *acyclic* (i.e., G is a DAG.) However, if G contains cycles, we can find all the *strongly connected components* (*SCCs*) of G by using Tarjan's algorithm in O(e) time [18] and collapse each of them into a representative node, and transform G to a DAG [14]. Clearly, each node in an *SCC* is equivalent to its representative node as far as reachability is concerned.

We also use $u \rightarrow v$ to stand for an arc from u to v in a directed graph (and (u, v) for an edge in an undirected graph with u and v as endpoints.)

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 arcs (E_{tree}): arcs appearing in T.
- cross arcs (E_{cross}): any arc u $\rightarrow$ v such that u and v are not on the same path in T.
- forward arcs ($E_{forward}$): any arc u $\rightarrow$ v not appearing in T, but there exists a path from u to v in T
- back arcs (E_{back}): any arc u $\rightarrow$ v not appearing in T, but there exists a path from v to u in T.

All cross, forward, and back arcs are referred to as non-tree arcs. (But in a DAG, we do not have back arcs since a back arc implies a cycle.) For illustration, consider the DAG shown in Figure 1. For it, we may find a spanning tree as shown by the solid arrows in Figure 1. (In the figure, each non-tree arc is represented by a dashed arrow.)

As in [22], we can assign each node v in T an interval $[\alpha_v, \beta_v)$, where α_v is v's preorder number (denoted $pre(v)$) and β_v - 1 is equal to the largest preorder number among all the nodes in $T[v]$. So another node u labeled $[\alpha_u, \beta_u)$ is a descendant of v (with respect to T) iff $\alpha_u \in [\alpha_v, \beta_v)$ [22], as illustrated in Fig. 1. If $\alpha_u \in [\alpha_v, \beta_v)$, we say, $[\alpha_u, \beta_u)$ is subsumed by $[\alpha_v, \beta_v)$. This method is called the tree labeling.

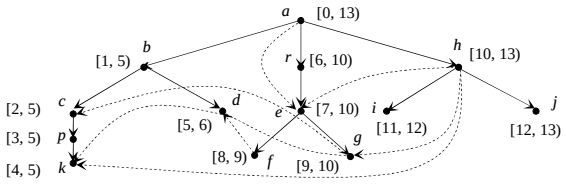


Figure 1.A Spanning Tree and Intervals

3.2. Graph Decomposition and Reachability Checking

In this subsection, we discuss a kind of decomposition of $G(V, E)$: a spanning tree T and a subgraph G_c such that $|V(G_c)| < |V|$. What we want is to transform the reachability checking of any two nodes in G to a checking over T and a checking over G_c. Obviously, G_c has to contain E_{cross}. But some arcs from T need to be included and carefully recognized. For this purpose, we introduce some new concepts.

Denote by V' the set of all the *endpoints* of the cross arcs. We have $V' = V_{start} \cup V_{end}$, where V_{start} contains all the *start nodes* while V_{end} all the *end nodes* of the cross arcs. For example, for the graph shown in Figure 1, we have $V_{start} = \{h, g, f, d\}$ and $V_{end} = \{e, g, c, d, k\}$. No attention is paid to the forward arc (a, e) in the graph since it can be simply removed without impacting the checking of reachability.

The first concept is the so-called crossing range, which is a second pair of integers associated with each node $v \in V$, defined below.

Definition 1 (*crossing range*) Let T be a spanning tree (forest) of G. Let v be a node with the children $v_1, \ldots, v_j$ in G. Let $[\alpha_i, \beta_i)$ $(i = 1, \ldots, j)$ be the interval of v_i. Set $a_v = min_i\{\alpha_i\}$ and $b_v = max_i\{\alpha_i\}$. Then, $\{a_v, b_v\}$ is called the crossing range of v.

For technical convenience, for any node v without child nodes in G, both its a_v and b_v are set to be α_v. For example, with respect to the spanning tree shown in Figure 1, the crossing ranges of the nodes in G can easily be computed, as shown in Figure 2.

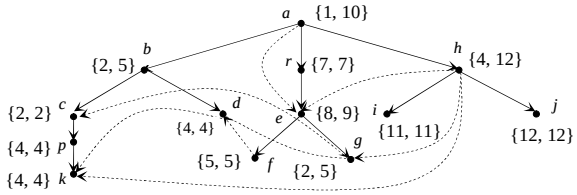


Figure 2.Start Nodes, End Nodes, and Crossing Ranges

Definition 2 (*critical nodes*) A node v in a spanning tree T of G is *critical* if the following conditions are satisfied:

- There is a subset U of V_{start} with $|U| > 1$ such that for any two nodes $u_1, u_2 \in U$ they are not related by the ancestor/descendant relationship and v is the lowest common ancestor of all the nodes in U.
- For each $u \in U$, its crossing range $\{a_u, b_u\}$ is not within $T[v]$. That is, a_u or b_u is a preorder number not appearing in $T[v]$.

All the critical nodes with respect to T are denoted by $V_{critical}$. For example, in the spanning tree shown in Fig. 1, node e is the lowest common ancestor of $\{f, g\}$ and both f and g are in V_{start}. In addition, the crossing ranges of f and g are not within $T[e]$ (see Figure 2). So e is a critical node. We also notice that node a is the lowest common ancestor of $\{d, f, g, h\}$. But the crossing ranges of all the four nodes are in $T[a]$. Thus, a is not a critical node. In the same way, we can check all the other nodes and find that $V_{critical} = \{e\}$.

The reason for imposing second condition in the above definition is that if any cross arc going out of a node in $T[v]$ reaches only a node in $T[v]$, then the reachability

between v and any other node in G can be checked by the tree labeling. So it is not necessary to include v in G_c if $v \notin V_{start} \cup V_{end}$.

Now we consider a tree (forest) structure T_c, called a *critical* tree of G (with respect to T), which contains all the nodes in $V_{critical} \cup V_{start} \cup V_{end}$. In T_c, there is an arc from u to v if there is a path P from u to v in T and P contains no other node in $V_{critical} \cup V_{start} \cup V_{end}$, as illustrated in Figure 3(a).

Denote $T_c \cup E_{cross}$ by G_c (see Figure 3(b).) Then, T and G_c make up a decomposition of G. It can be seen that $V(G_c)$ is much smaller than V.

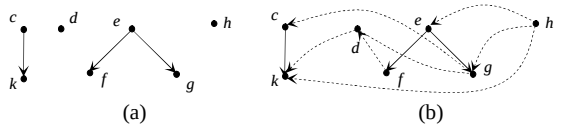


Figure 3. Illustration for T_c

For any two node u, v appearing on a path in T, their reachability can be checked using their associated intervals. However, our question is, if they are not on the same path in T, can we check their reachability by using G_c?

To answer this question, we need another concept, the so-called *anchor nodes*.

First, for any critical node v, we slightly change its crossing range as follows.

- Assume that U is a subset of V_{start} such that v is the lowest common ancestor of all the nodes in it and satisfies condition (1) and (2) in Definition 2.
- Set $a_v \leftarrow min\{min_{u \in U}\{a_u\}, a_v\}$;
  $b_v \leftarrow max\{max_{u \in U}\{b_u\}, b_v\}$.

For instance, node e's original crossing range is {8, 9} (see Fig. 2). The crossing ranges of node f and g are {5, 5} and {2, 5}, respectively. So e's original range will be changed to {2, 9}.

Next, we denote by $C(v)$ all the critical nodes in $T[v]$ plus all those start nodes of the cross arcs which appear in $T[v]$. We consider a maximal subset of $C(v)$ such that each node in it does not have an ancestor in $C(v)$. Denote such a subset as $C_s(v)$. It can be seen that in $C_s(v)$ there is at most one node u such that its crossing range is not within $T[v]$. Otherwise, a new critical node in $T[v]$ will be created (see Definition 2), which is an ancestor of u and in $C(v)$, contradicting the fact that $u \in C_s(v)$ and thus has no ancestor in $C(v)$.

Definition 3 (*anchor nodes*) Let G be a DAG and T a spanning tree of G. Let v be a node in T. We associate two nodes with v as below.

- A node $y \in C_s(v)$ is called an anchor node (of the first kind) of v if its crossing range is not within T[v], denoted v*. If such a node does not exist, v* is set to be the special symbol "-".
- 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 arc. w is denoted v**. If such a node does not exist, v** is set to be "-".

For example, in the graph shown in Fig. 1, $r^* = e$. It is because node e is a critical node in $C_s(r)$ and its crossing range {2, 9} (note that the crossing range of a critical node is changed) is not within $T[r]$. But r^{**} does not exist since it does not have an ancestor which has a cross incoming arc. 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 that graph.

3.3. Recognizing Critical Nodes

From the discussion in the previous subsection, we know that all the critical nodes need to be recognized to construct G_c. Now we discuss an efficient algorithm for this task.

We will search T bottom up and produce a subtree T' of T such that only the critical nodes and the nodes from V_{start} are included. Initially, T' is set to ∅, and all the nodes in V_{start} are marked. Then, during the traversal of T, any node belonging to V_{start} or any critical node, once it is recognized, will be inserted into T'. To this end, each node v inserted into T' will be associated with two links, denoted *parent*(v) and *left-sibling*(v), respectively. *parent*(v) is used to point to the parent of v in T' while *left-sibling*(v) points to a node in T' created just before v, which is not a descendant of v in T.

Concretely, *parent*(v) and *left-sibling*(v) will be created as below.

- Let v be the node currently inserted into T'.
- If v is not the first node inserted into T', we do the following:

Let v' be the node inserted just before v. If v' is not a child (descendant) of v, create a link from v to v', called a *left-sibling* link and denoted as *left-sibling*(v) = v'. If v' is a child (descendant) of v, we will first create a link from v' to v, called a *parent* link and denoted as *parent*(v') = v. Then, we will go along the left-sibling chain starting from v' until we meet a node v'' which is not a child (descendant) of v. For each encountered node u except v'', set *parent*(u) ← v. Finally, set *left-sibling*(v) ← v''.

Figure 4 is a pictorial illustration of this process.

In Figure 4(a), we show the navigation along a left-sibling chain starting from v' when we find that v' is a child (descendant) of v. This process stops whenever we meet v'', a node that is not a child (descendant) of v. Figure 4(b) shows that the left-sibling link of v is set to point to v'', which is previously pointed to by the left-sibling link of v's left-most child.

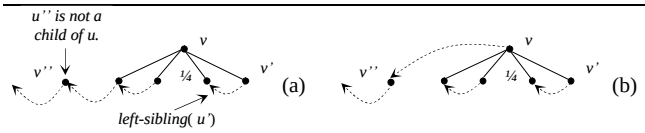


Fig. 4: Illustration for the Construction of *T'*

Extending the above process with the recognition of critical nodes and the computation of crossing ranges, we get an efficient algorithm for finding all the critical nodes.

Algorithm *find-critical*(*T*)
begin

1. $T' \leftarrow \varnothing$. Mark any node in *T*, which belongs to V_{start}.
2. Let *v* be the first marked node encountered during the bottom-up searching of *T*. Insert *v* in *T'*.
3. Let *u* be the currently encountered node in *T*. Let *u'* be the node inserted into *T'* just before *u*. Do (4) or (5), depending on whether *u* is a marked node or not.
4. If *u* is marked, then insert *u* into *T'* and do the following.
   (a) If *u'* is not a child (descendant) of *u*, set *left-sibling*(*u*) = *u'* (i.e., a link from *u* to *u'*).
   (b) If *u'* is a child (descendant) of *u*, we will first set *parent*(*u'*) = *u*. Then, we will go along a left-sibling chain starting from *u'* until we meet a node *u''* which is not a child (descendant) of *u*. For each encountered node *w* except *u''*, set *parent*(*w*) ← *u*. Also, set *left-sibling*(*u*) ← *u''*. (See Fig. 6(b) for illustration.) Calculate initial a_u and b_u according to Definition 1. Let *W* be the set of all the encountered nodes during the navigation along the left-sibling chain (not including *u''*). Set $a_u \leftarrow min\{min_{w\in W}\{a_w\}, a_u\}$ and $b_u \leftarrow max\{max_{w\in W}\{b_w\}, b_u\}$.
5. If *u* is a non-marked node, then do the following.
   (c) If *u'* is not a child (descendant) of *u*, *u* is ignored.
   (d) If *u'* is a child (descendant) of *u*, we will go along a left-sibling chain starting from *u'* until we meet a node *u''* which is not a child (descendant) of *u*. If there are more than one node in *W* such that their crossing ranges not within *T*[*u*], insert *u* into *T'*, and compute a_u and b_u as (4.b). Otherwise, *u* is ignored.

end

In the algorithm, each node *v* belonging to V_{start} is simply inserted into *T'*, by which its $\{a_v, b_v\}$ is computed. (See 4.a and 4.b. in the algorithm.) For a node not belonging to V_{start}, we will check whether it satisfy the conditions given in Definition 2. If it is the case, it will be inserted into *T'*. At the same time, its crossing range will be calculated. Otherwise, it will be ignored. (See 5.c and 5.d in the algorithm.)

Obviously, the algorithm requires only O(*e*) time since each node in *T* is accessed at most two times and for each node *v* *out-degree*(*v*) arcs will be visited.

4. Finding Optimal Spanning Trees

For a given DAG *G*(*V*, *E*) , we can find different spanning trees by exploring *G* in different ways. Especially, for different spanning trees, the size of G_c can be different. Clearly, what we want is to find such a spanning tree that G_c is minimized. But, how to find such a spanning tree?

Let $\Im(G)$ be the family including all the spanning trees of *G*. For $T \in \Im(G)$, denote by $r_T(v)$ the number of the cross arcs coming to *v* with respect to *T*. We define

$$\boldsymbol{R}(T) = \sum_{v\in V} r_T(v) .$$

Intuitively, the smaller $\boldsymbol{R}(T)$ is, the smaller the size of G_c. So our optimization problem is to find a *T* such that $\boldsymbol{R}(T)$ is minimum. Unfortunately, there are exponentially many spanning trees for a given DAG. So it is unlikely to find an optimal one in polynomial time. In fact, it is *NP*-complete.

In the following discussion, we will first prove the NP-completeness of the problem. Then, we will present a top-down algorithm to find a spanning tree of *G* with fewer cross arcs than a traditional depth-first search. Next, we will discuss a heuristic which can be integrated into our top-down algorithm to mitigate the problem to some extent.

4.1. NP-completeness

First, we notice that

$$\boldsymbol{R}(T) = e - n + 1 - \sum_{v\in V} f_T(v) ,$$

where $f_T(v)$ is the number of the forward arcs coming to *v* with respect to *T*. Thus, minimizing $\boldsymbol{R}(T)$ is equivalent to maximizing

$$\boldsymbol{F}(T) = \sum_{v\in V} f_T(v) .$$

Therefore, to show the NP-completeness of minimizing $\boldsymbol{R}(T)$, we can show the NP-completeness of maximizing $\boldsymbol{F}(T)$.

Let *P* be a path in *T*. Let *u*, *v* be two nodes on *P*. We call the forward arc from *u* to *v* an attached arc of *P*. Obviously, to maximize $\boldsymbol{F}(T)$, we need to maximize the number of the attached arcs of each path in *T*.

Now we consider a much easier problem to find a *T* such that it has a path with the maximal number of attached arcs, and show that even this problem is NP-complete.

For this purpose, we define the following decision problem:

Input: A DAG *G* and a positive integer $k \le n$.

Question: Is there a spanning tree *T* such that it contains a path *P* of length *k* with the number of the attached arcs of *P* equal to $(k-1)(k-2)/2$.

We call this problem a *maximum P-attachment* problem.

Proposition 2: The maximum *P*-attachment is NP-complete.

Proof: It is easy to see that the problem is in NP: An algorithm can generate all spanning trees *T* of *G* and check each *T* to see whether it has a maximum P-attachment.

The completeness for NP is shown by a reduction from the basic NP-complete problem SATISFIABILITY [7]. Let an instance of SATISFIABILITY be given by a

collection of clauses $C = \{c_1, \ldots, c_k\}$. Each c_i is of the form $x_{i1} \vee x_{i2} \vee \ldots, x_{ik_i}$, where x_{ij} is a literal. We form a DAG in two steps:

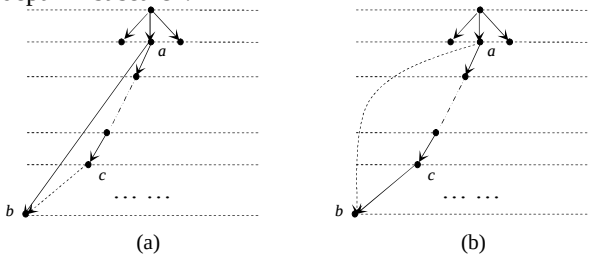
1. Generate an undirected graph G', whose nodes are pairs of integers $[i, j]$, for $1 \le i \le k$ and $1 \le j \le k_i$. A node $[i, j]$ is connected to another node $[k, l]$ if both of the following hold:
   - $i \ne j$, and
   - $x_{ij} \ne \neg x_{kl}$.
2. Explore G' in the depth-first manner to change it to a DAG G'' as below:
   - If an edge (u, v) in G' is explored from u to v, create an arc $u \to v$ in G''.
   - In G'', reverse the direction of any back arc. (Then, the resulting G'' must be a DAG.)

Obviously, the DAG can be constructed in polynomial time.

Now we claim that there is a satisfying truth assignment for C if and only if there is spanning tree containing a path P of length k such that the number of the attached arcs of P equal to $(k-1)(k-2)/2$. It is because if C is satisfiable, there must be a clique of size k. Exploring the clique in the depth-first search and then reverse any back arc, we will get a path of length k with the number of the attached arcs equal to $(k-1)(k-2)/2$.

Next assume that T is a spanning tree of G'', which contains a path P of length k with the number of the attached arcs equal to $(k-1)(k-2)/2$. Assigning a value to the variable in each literal x corresponding to a node on P such that x is *true* while a value to the variable in any other literal y such that y is *false*, we get a satisfying truth assignment for C for the following reason. First, each node corresponds to a literal in a different clause. Second, for each pair of literals represented by two nodes on the path, they are not negation of each other. **4.2. A Top-down Slgorithm**

In this *subsection*, we give our top-down algorithm to explore G, which is able to find a spanning tree with more forward arcs than a traditional depth-first search. The main idea behind the algorithm is to recognize a kind of "triangles" as illustrated in Figure 5(a), during a depth-first search.



(a) (b)

Figure 5. Illustration for "Triangles" Encountered during a DFS

In Figure 5(a), assume that node c is the current node along a path *from* a to c, and b is one of c's children, but has been visited before (along the arc from a to b). We can remove the tree arc $a \to b$ and make $c \to b$ a tree arc. Then, $a \to b$ becomes a forward arc as illustrated in Figure 5(b).

In order to find such kind of transformations, we arrange a boolean array B such that $B[i] = 1$ indicates that node i is on the current path during the depth-first search. Otherwise, $B[i] = 0$. For simplicity, we assume that G is a rooted graph. If it is not the case, a virtual root is created and connected to all those nodes that have no incoming arcs. Then, by the current path, we mean the path from the root to the currently encountered node. Let $v_1 \to v_2 \to \ldots \to v_k$ be the current path. And we are going to access one of v_k's children. At this moment, in B all $B[v_j]$'s must be set to 1 ($j = 1, \ldots, k$) while all the other entries are 0.

In the following algorithm, three special data structures are used:

S – a stack to control the depth-first search;

$C(v)$ – a list containing all the children of v in G.

c-list(v) – a list of all those children of v in G, which have not yet been visited.

$C_T(v)$ – a list containing all the children of v in T.

Algorithm *DFS-f*(G)

```
begin
1.    Each entry of B is set 0;
2.    c-list(v) := C(v) for each v;
3.    push(root, S); mark root ; B[root] := 1;
4.    while (S ≠ f) do {
5.      v := top(S);
6.      while c-list(v) ≠ f do {
7.        let u be the first node in c-list(v), chosen according
          to a heuristic if any;
8.        if u is marked then {
9.          let u' be the parent of u in T;
10.         if B[u'] = 1 then {
11.             remove u from C_T(u'); add u to C_T(v);
12.         }
13.         remove u from c-list(v);
14.       }
15.       else {   add u to C_T(v);
16.                push(u, S); mark u; B[u] := 1; v := u; }
17.     }
18.     w := pop(S); B[w] := 0;
19.   }
end
```

In the above algorithm, the stack S is used to keep the current path. Then, for each node w in S we have $B[w] = 1$. Let v be the node at the top of S (i.e., *top*$(S) = v$; see line 5.) We will check the first element u in *c-list*(v) (note that initially *c-list*(v) contains all the children of v; see line 2.) Two cases need to be distinguished: u is marked (showing that v has been visited before), or not marked. If u is marked, we will check whether its parent u' (in the spanning tree T created up to now) is on the current path by checking $B[u']$ (see lines 7 – 8.) If is the case, a transformation will be conducted (see line 11.)

Otherwise, u is simply removed from *c-list*(v) (see line 13.) If u is not marked, it will be added to T as one of v's children (see line 15.) Then, u is pushed into S and marked (see line 16.) In a next step, one of u's children will be visited (see the assignment statement: $v := u$ in line 16). We repeat this process until we meet a node v' with *c-list*(v') = f . In this case, the top element of S is popped out and the corresponding entry in S is set to 0 (see line 18.)

4.3. Heuristics

Now we discuss a kind of heuristics, which enables us to get a spanning tree with even more forwards arcs. For this purpose, we arrange a preprocessor to stratify the nodes of a graph into different levels to get some information on the structure of the graph.

Let $G(V, E)$ be a DAG. We decompose V into subsets $V_0, V_1,..., V_h$ such that $V = V_0 \cup V_1 \cup ... \cup V_h$ and each node in V_i has its children appearing only in $V_0, ..., V_{i-1}$ ($i = 0, ..., h - 1$), where h is the height of G, i.e., the length of the longest path in G. For each node v in V_i, we say, its level is i, denoted $l(v) = i$. We also use $C_j(v)$ ($j > i$) to represent a set of links with each pointing to one of v's chidren, which appears in V_j. Therefore, for each v in V_i, there exist $i_1, ..., i_k$ ($i_l < i$, $l = 1, ..., k$) such that the set of its children equals $C_{i_1}(v) \cup ... \cup C_{i_k}(v)$. Such a DAG stratification can be done in O(e) time, by using the following algorithm, in which we use $G_1 \backslash G_2$ to stand for a graph obtained by deleting the arcs of G_2 from G_1; and $G_1 \cup G_2$ for a graph obtained by adding the arcs of G_1 and G_2 together. In addition, $d(v)$ represents v's out-degree.

Algorithm *graph-stratification*(G)
begin
1. V_0 := all the nodes with no outgoing arcs;
2. **for** $i = 0$ to $h - 1$ **do**
3. {W := all the nodes that have at least one child in V_i;
4. **for** each node v in W **do**
5. {let $v_1, ..., v_k$ be v's children appearing in V_i;
6. $C_i(v) := \{v_1, ..., v_k\}$;
7. **if** $d_{in}(v) > k$ **then** remove v from W;};
8. $G := G\backslash\{v \to v_1, ..., v \to v_k\}$;
9. $d(v) := d(v) - k$;
10. $V_{i+1} := W$;
11. }

end

In the above algorithm, we first determine V_0, which contains all those nodes having no outgoing arcs (see line 1). In the subsequent computation, we determine $V_1, ..., V_h$. In order to determine V_i ($i > 0$), we will first find all those nodes that have at least one child in V_{i-1} (see line 3), which are stored in a temporary variable W. For each node v in W, we will then check whether it also has some children not appearing in V_{i-1}, which can be done in a constant time as demonstrated below. During the process, the graph G is reduced step by step, and so does $d(v)$ for each v (see lines 8 and 9). First, we notice that after the jth iteration of the out-most for-loop, V_0 ..., V_j are determined. Denote $G_j(V, E_j)$ the changed graph after the jth iteration of the out-most for-loop. Then, any node v in G_j, except those in $V_0 \cup ... \cup V_j$, does not have children appearing in $V_0 \cup ... \cup V_{j-1}$. Denote $d_j(v)$ the out-degree of v in G_j. Thus, in order to check whether v in G_i has some children not appearing in V_{i-1}, we need only to check whether $d_i(v)$ is strictly larger than k, the number of the child nodes of v appearing in V_{i-1} (see line 7).

During the process, each arc is accessed only once. So the time complexity of the algorithm in bounded by O(e).

As an example, consider the graph shown in Figure 1. Applying the above algorithm to this graph, we will generate a stratification of the nodes as shown in Figure 6.

In Figure 6, the nodes of the DAG shown in Fig. 1 are divided into seven levels: $V_0 = \{k\}$, $V_1 = \{p\}$, $V_2 = \{d, c\}$, $V_3 = \{f, g\}$, $V_4 = \{e, f, i\}$, $V_5 = \{b, r, h\}$, and $V_6 = \{a\}$. Associated with each node at each level is a set of links pointing to its children at different levels below.

In terms of the graph stratification, we define a heuristic. Let v be a node appearing at level i, with a set of links pointing to its children: $C_{i_1}(v)$ ..., $C_{i_k}(v)$ ($i_l < i$, $l = 1, ..., k$). We will associate it with a number $\sigma(v)$, calculated as below:

$$\sigma(v) = (i - i_1)|C_{i_1}(v)| + \ldots + (i - i_k)|C_{i_k}(v)|,$$

in which a child node at a lower level receives a larger weight since the possibility for a node to be incident to a forward arc increases as its level decreases.

Then, our intention is to choose first the node with the highest priority number at each step in a *DFS-f* search. However, for the purpose of heuristics, we will use the following number to take all the descendants of v into account:

$$\varpi(v) = \frac{1}{hd^2 |V|} \sum_{u \in desc(v)} s(u),$$

where d is the largest out-degree of the nodes in G, and *desc*(v) is a set containing all the descendants of v, including v itself. Note that if we work bottom-up all $\varpi(v)$'s can be produced in O(e) time.

This heuristic can be used in the *DFS-f* search in such a way that each time we choose a child from *c-list*(v) the node with the largest ϖ–value is selected. The tie is resolved arbitrarily.

For example, if we use this heuristic to control a *DFS-f* search of the graph shown in Fig. 1, we will create a spanning tree as shown by the solid arcs in Figure 4.

The search starts at root a, and then go to h since $\varpi(h)$ is larger than both $\varpi(r)$ and $\varpi(b)$. From h we will go to e (since $\varpi(e)$ is larger than both $\varpi(i)$ and $\varpi(j)$), and then

go to g (since $\varpi(g)$ is larger than $\varpi(f)$). Repeating this process, we will generate that spanning tree. However, with respect to it we have only 5 cross arcs. But with respect to the spanning tree shown in Fig. 1, we have 7 cross arcs. For this example, a set of 5 cross arcs is in fact minimum since there are only 3 arcs, for which there is a path of length ≥ 2 connecting their endpoints; and altogether there are 8 non-tree arcs.

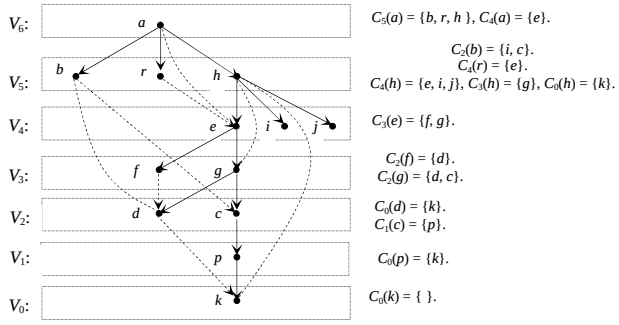


Figure 6. Graph Stratification

5. Recursive Graph Deduction

We note that G_c itself can be decomposed, leading to a further space decrement. Repeating this operation, we will get a recursive decomposition of G. In this subsection, we elaborate this process.

Let G_0 be a DAG. Denote by T_0 a spanning tree of G_0. Denote by E^0_{cross} the set of all the cross arcs with respect to T_0. Then, as discussed in Section III, T_0 and $G_1 = T^0_c \cup E^0_{cross}$ make up a decomposition of G_0, where T^0_c is the critical tree of G_0. Recursively decomposing G_1, we will find a series of tree structures:

$$T_0, T_1, ..., T_{k-1}, \ (k \geq 1),$$

such that T_0 is a spanning tree of G_0 and each T_i (i = 1, ..., k - 1) is a spanning tree of $G_i = T^{i-1}_c \cup E^{i-1}_{cross}$, where T^{i-1}_c is the critical tree of G_{i-1}, and E^{i-1}_{cross} is a set of all the cross arcs with respect to T_{i-1}. We refer to G_k as the *residue* graph of G, denoted as G_r. It can be a graph or a tree.

In this way, we are able to associate each node v in G_0 with two sequences: an interval sequence and an anchor node sequence to check reachability:

$$[a^v_0, b^v_0), ..., [a^v_j, b^v_j), (j \leq k - 1),$$

where each $[a^v_i, b^v_i)$ is an interval generated by labeling T_i;

$$(x^v_0, y^v_0), ..., (x^v_l, y^v_l), (l \leq j),$$

where each x^v_i is a pointer to an anchor node of the first kind (a node appearing in G_{i+1}) while each y^v_i a pointer to an anchor node of the second kind (also, a node in G_{i+1}). Each (x^v_i, y^v_i) can be generated as described in Section III.

We notice that the anchor node sequences imply a graph, in which there exists an arc $u \rightarrow v$ iff there is an entry $<x, y>$ in the anchor node sequence associated with u such that $x = v$, or $y = v$. The arc is labeled with $\{i, *\}$ or $\{i, **\}$ with i used to indicate that $<x, y>$ is the ith entry in the corresponding anchor node sequence. If $x = v$, the arc is labeled with $\{i, *\}$. If $y = v$, the arc is labeled with $\{i, **\}$. We refer to such a graph as a *transitive core* graph of G (or simply *core* graph of G) and denote it by G_{core}.

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, referred to as P_v, contains only the arcs labeled with $(i, *)$ while the path starting from u, referred to as P_u, contains only the arcs labeled with $(i, **)$. Each time we reach two nodes v' and u' through two arcs labeled respectively with $(i, *)$ and $(i, **)$, we will check whether $[a^{v'}_i, b^{v'}_i)$ subsumes $[a^{u'}_i, b^{u'}_i)$. (Remember that each node in $G_0 = G$ is associated with an interval sequence $[a^v_0, b^v_0), ..., [a^v_m, b^v_m)$ for some $m \geq 0$.) If it is the case, v is an ancestor of u. Otherwise, we traverse along P_v and P_u, reaching v'' and u'' through two arcs labeled respectively with $(i + 1, *)$ and $(i + 1, **)$ and checking $[a^{v''}_{i+1}, b^{v''}_{i+1})$ against $[a^{u''}_{i+1}, b^{u''}_{i+1})$. We continue this process. After l steps for some l, we will meet two nodes v''' and u''' such that v''' does not have an out-going arc labeled with $(l + 1, *)$ or u''' does not have an out-going arc labeled with $(l + 1, **)$. If $[a^{v'''}_l, b^{v'''}_l)$ subsumes $[a^{u'''}_l, b^{u'''}_l)$, v is an ancestor of u. Otherwise, we further check whether $l = k$. If it is the case, we will check whether u''' is reachable from v''' in G_r.

6. Experiment

In this section, we report the test results. We conducted our experiments on a DELL desktop PC equipped with Pentium III 1.0 Ghz processor, 512 MB RAM and 20GB hard disk. The programs are compiled using Microsoft virtual C++ compiler version 6.0, running standalone.

6.1.On the Tested Methods

In the experiments, we have tested eight methods:

Chain decomposition by Chen *et al.* (*CD* for short) [5],

- Tree encoding by Agrawal *et al.* (*TE* for short) [1],
- 2-hop labeling by Cohn et al. (2-*hop* for short) [4],
- Dual labeling by Wang et al. (*Dual-II* for short) [22],

- Matrix multiplication by Warren (*MM* for short) [25],
- Tree-path by Jin et al. (*TPath* for short) [9],
- GRAIL by Yildirim et al. [23]
- Recursive DAG decomposition (discussed in this paper, *RDD* for short).

The theoretical computational complexities of these methods are listed in Table 1 (in Section II).

In the experiments, Jagadish's chain decomposition is not included. It is because Chen's method works in a similar way, but has a much better labeling time. For the dual labeling, we implemented Dual-II, instead of Dual-I for tests. For non-sparse graphs, Dual-I needs even more space than any traditional matrix-based method; no compression in any sense.

6.2.Conclusion

In this paper, a new method is proposed to compress transitive closures to support reachability queries. The main idea behind it is to decompose G into a series of spanning trees: T0, …, Tk-1 (for some $k \geq 1$), and a residue graph Gr, which enables us to associate two sequences with each node in G: an interval sequence and an anchor node sequence. Especially, in terms of the anchor sequences, a directed graph, called a transitive core graph of G, can be constructed, which can be used to control the process of reachability checking. The method needs $O(ke + \omega r 1.5nr)$ time to create a compressed transitive closure with $O(kn + \omega rnr)$ space requirement, and O(k) query time, where n is the number of the nodes in Gr, and ωr is the width of Gr, defined to be the size of a largest node subset U of Gr such that for any pair of nodes u, $v \in U$ there does not exist a path from u to v or from v to u.

An extensive experiment is conducted to test different strategies over different kinds of graphs and real graphs, which shows that our method is promising. Our method is also a flexible strategy. For different applications, k can be set to different constants to reduce space overhead. But the query time is still bounded by a constant.

References

[1] Agrawal, A. Borgida and H.V. Jagadish, "Efficient management of transitive relationships in large data and knowlarc bases," Proc. of the 1989 ACM SIGMOD Intl. Conf. on Management of Data, Oregon, 1989, pp. 253-262.

[2] J. Cheng, J.X. Yu, X. Lin, H. Wang, and P.S. Yu, Fast computation of reachability labeling for large graphs, in Proc. EDBT, Munich, Germany, May 26-31, 2006.

[3] N.H. Cohen, "Type-extension tests can be performed in constant time," ACM Transactions on Programming Languages and Systems, 13:626-629, 1991.

[4] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, Reachability and distance queries via 2-hop labels, SIAM J. Comput, vol. 32, No. 5, pp. 1338-1355, 2003.

[5] Y. Chen and Y.B. Chen, An Efficient Algorithm for Answering Graph Reachability Queries, in Proc. 24th Int. Conf. on Data Engineering (ICDE 2008), IEEE, April 2008, pp. 892-901.

[6] Y. Chen, General Spanning Trees and Reachability Query Evaluation, in Proc: 2nd Canaidan Conference on Computer Science and Software Engineering (C3S2E'09), ACM, Montreal, Canada, May 19-21, 2009, pp. 243-252.

[7] M.R. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman & Co., 1990.

[8] H.V. Jagadish, "A Compression Technique to Materialize Transitive Closure," ACM Trans. Database Systems, Vol. 15, No. 4, 1990, pp. 558 - 598.

[9] R. Jin, N. Ruan, Y. Xiang, and H. Wang, Path-Tree: An Efficient Reachability Indexing Scheme for Large Directed Graphs, ACM Transaction on Database Systems, Vol. No.1, 2011, pp. 1-52.

[10] R. Jin, Y. Xiang, N. Ruan, and H. Wang, "Efficiently Answering Reachability Queries on Very Large Directed Graphs," Proc. of ACM SIGMOD Intl. Conf. on Management of Data, Vancouver, Canada, 2008.

[11] D.E. Knuth, The Art of Computer Programming, Vol.1, Addison-Wesley, Reading, 1969.

[12] H.A. Kuno and E.A. Rundensteiner, "Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation," IEEE Transactions on Knowlarc and Data Engineering, vol. 10. No. 5, 1998, pp. 768-792.

[13] W.C. Lee and D.L Lee, "Path Dictionary: A New Access Method for Query Processing in Object-Oriented Databases," IEEE Transactions on Knowlarc and Data Engineering, vol. 10. No. 3, 1998, pp. 371-388.

[14] I. Munro. Efficient determination of the transitive closure of directed graphs. Information Processing Letters, vol. 1 (2), pp. 56-58, 1971.

[15] R. Schenkel, A. Theobald, and G. Weikum, HOPI: an efficient connection index for complex XML document collections, in Proc. EDBT, 2004.

[16] R. Schenkel, A. Theobald, and G. Weikum, Efficient creation and incrementation maintenance of HOPI index for complex xml document collection, in Proc. ICDE, 2006.

[17] M.A. Schubert and J. Taugher, "Determing type, part, colour, and time relationship," 16 (special issue on Knowlarc Representation):53-60, Oct. 1983.

[18] R. Tarjan: Depth-first Search and Linear Graph Algorithms, SIAM J. Compt. Vol. 1. No. 2. June 1972, pp. 146 -140.

[19] R. Tarjan: Finding Optimum Branching, Networks, 7. 1977, pp. 25 -35.

[20] J. Teuhola, "Path Signatures: A Way to Speed up Recursion in Relational Databases," IEEE Trans. on Knowledge and Data Engineering, Vol. 8, No. 3, June 1996, pp. 446 - 454.

[21] M. Thorup, "Compact Oracles for Reachability and Approximate Distances in Planar Digraphs," JACM, 51, 6(Nov. 2004), 993-1024.

[22] 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.

[23] H. Yildirim, V. Chaoji, and M.J. Zaki, GRAIL: Scalable Reachability Index for Large Graphs, in Proc. VLDB Endowment, 3(1), 2010, pp. 276-284.

[24] Y. Zibin and J. Gil, "Efficient Subtyping Tests with PQ-Encoding," Proc. of the 2001 ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Application, Florida, October 14-18, 2001, pp. 96-107.

[25] H.S. Warren, "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations," Commun. ACM 18, 4 (April 1975), 218 - 220.

Subscriptions and Individual Articles:

User	**Hard copy:**
Institutional:	800 (HKD/year)