General Spanning Trees and Reachability Query Evaluation

Yangjun Chen

Dept. Applied Computer Science, University of Winnipeg

515 Portage Ave., Winnipeg, Manitoba, Canada R3B 2E9

y.chen@uwinnipeg.ca

ABSTRACT

Graph reachability is fundamental to a wide range of applications, including CAD/CAM, CASE, office systems, software management, as well as geographical navigation and internet routing. Many applications involve huge graphs and requires fast answering of reachability queries. Several reachability labeling methods have been proposed for this purpose. They assign labels to the nodes, such that the reachability between any two nodes can be determined using their labels only. In this paper, we propose a new data structure, called a *general spanning tree* of a directed acyclic graph (*DAG*) to minimize label space. Different from a traditional spanning tree, an edge in a general spanning tree *T* of a DAG *G* may corresponds to a path in *G*. That is, for each edge $u \rightarrow v$ in *T*, we have a path from *u* to *v* in *G*. An algorithm is discussed to find such a tree with the least number of leaf nodes in $O(bn\sqrt{b})$ time, where *n* is the number of the nodes of *G*, and *b* is the number of the leaf nodes of *T*. It can be proven that *b* equals *G*'s width, defined to be the size of a largest node subset *U* of *G* such that for every pair of nodes $u, v \in U$, there does not exist a path from *u* to *v* or from *v* to *u*. Based on *T*, we are able to reduce the label space to O(*bn*) with O(log*b*) reachability query time. Our method can also be extended for graphs containing cycles.

1. INTRODUCTION

Given two nodes *u* and *v* in a directed graph *G*(*V*, *E*) (with $|V| = n$ and $|E| = e$), we want to know if there is a path from *u* to *v*. The problem is known as *graph reachability*, and well-explored in several fields of computation [1, 4, 8, 9, 23, 27, 31]. In many applications (e.g., geographical navigation), graph reachability is one of the most basic operations, which means fast processing is mandatary. A naive approach to this problem is to precompute the reachability between each pair of nodes - in other words, to compute and store the transitive closure (*TC*) of *G*, so that we can answer reachability queries in constant time. However, this requires $O(n^2)$ space, which makes it impractical for massive graphs.

Recently, interests in this problem is rekindled and several methods [5, 9, 23, 27] have been proposed to compress TC but without sacrificing much query time. The main idea of them is to label a graph in some way such that the reachability between nodes can be decided using their labels. In this sense, a transitive closure is compressed.

In this paper, we discuss a new concept of *general spanning trees*, based on which an approach for labeling a DAG is devised. Especially, an efficient algorithm is proposed to find a general spanning tree with the least number of leaf nodes, which equals *G*'s width *b*, defined to be the size of a largest subset of pairwise unreachable nodes in *G*. The time complexity of the algorithm is bounded by $O(bn\sqrt{b})$ time. The labeling time and the label space are bounded by O(*be*) and O(*bn*), respectively; and a reachability query takes O(log*b*) time.

The main idea of this algorithm is to partition a DAG into a minimal set of disjoint chains, which is in fact the problem of poset (partially ordered set) decomposition [12] since a poset can always be represented as a DAG. The size of an antichain (a largest set of pairwise unreachable nodes) can be determined in $O(n^{2.5})$ time (see Lemma 10.4.1 in [2], page 190.) Up to now, however, the best way to solve this problem is to transform it to a *min*-flow problem as Jagadish did [16] (the same idea has also been suggested by some other researchers; see [29], pages 272 - 274.) It requires $O(n^3)$ time. Recently, Chen et al. [5] proposed a new way to do the task. Their idea is to stratify a DAG into several bipartite graphs. But it fails to find a minimal set of disjoint chains in some cases (see the analysis given in the Appendix). Our algorithm is the first to break the $O(n^3)$ bottleneck.

The remainder of the paper is organized as follows. In Section 2, we review the related work. In Section 3, we define the concept of general spanning trees and show how it can be used to produce an efficient node labeling. Section 4 is devoted to the description of our algorithm to decompose a DAG into chains, based on which a general spanning tree with the least number of leaf nodes can be constructed. Section 5 concludes the paper.

C3S2E-09 2009, May 19-21, Montreal [QC, CANADA]
Editor: B C. DESAI, Copyright ©2009 ACM 978-1-60558-401-0/09/05 $5.00

2. RELATED WORK

Graphs reachability has applications in a wide range of areas. For example, in object-oriented programming, graph reachability is important in managing class inheritance hierarchies. Many approaches have be proposed, such as *PE-Encoding* [8] and *PQ-Encoding* [31], which use a bottom-up bit vector labeling scheme [3] with O(n) bits per label, where n is the number of nodes. In 1996, Teuhola [25] proposed a signature-based method, by which each node is assigned a signature (which is in fact a bit string) generated using a set of hash functions. The space complexity is O($l \cdot n$), 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. In the area of semantic web, Christophides et al. [10] applied efficient labeling schemes to the problem of encoding subsumption hierarchies. In the worst case, however, the label space of this method is of O(n^2).

Recently, reachability labeling has enjoyed much attention due to its application in geographic navigation and internet routing, as well as XML document processing [5, 6, 8, 9, 27]. The interval-based labeling scheme is one of the most widely used approach for tree structures. It assigns an interval to each node, and the ancestor-descendant relationships between two nodes can be decided by checking set containment relationships between their interval labels. For a tree structure, this approach answers reachability queries in constant time, and the labeling process is of linear complexity. This method is extended to DAGs by Agrawal et al. [1]. In their method, each node v is assigned a set of non-overlapping interval $L(v)$. A node u is reachable from v iff the interval associated with u is contained by some interval in $L(v)$. Although labels can be assigned efficiently, for large, complicated graphs, the size of $L(v)$ can be linear in the graph size. This method is further improved by Chen [4], by genearting $L(v)$ in such a way that all the intervals in $L(v)$ are sorted. So the reachability query time is reduced to O(log$n$).

In [5], Chen et al. suggested an interesting method to decompose a DAG into disjoint chains such that on each 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. In addition to this, v is associated with an index sequence $(1, j_1) \ldots (i-1, j_{i-1})\ (i+1, j_{i+1}) \ldots (k, j_k)$ such that for any node u with index (x, y) if $x = i$ and $y > j$ or $x \neq i$ but $y \geq j_x$ it is a descendant of v, where k is the number of the disjoint chains. The time complexity of this algorithm is bounded by O($n^2 + kn\sqrt{k}$). However, as shown in the Appendix, k is not minimized. The label space and the query time of this method are O(kn) and O(log$k$), respectively.

Jagadish's method [16] is also based on graph decomposition. In his method, a DAG G is transformed to a flow network F by splitting each node v in G into two nodes x_v and y_v with an edge from x_v to y_v. Corresponding to every edge $u \rightarrow v$ in G, draw an edge $y_u \rightarrow x_v$ in F. Now add an artificial "source" to F from which there is an edge to each of F's nodes that have no predecessors. Similarly, a "sink" is introduced with an edge to it from each node with no successors in F. In order to find a minimal set of disjoint chains, assign a non-negative integer flow to each edge in F such that the flow into each node is equal to the flow out of that node. This is a standard min-flow problem and can be solved in O(n^3) time by well-known techniques such as those discussed in [11, 13, 17].

The method proposed by Wang at el. [27] is intended to handle reachability queries for sparse graphs. This method consists of two schemes: *Dual-I* and *Dual-II*. By finding a spanning tree T in G, *Dual-I*, 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 edges that do not appear in the spanning tree of G. Another node u with (a_u, b_u) and (x_u, y, z_u) is reachable from v iff $a_u \in [a_v, b_v)$, or $N(x_v, z_u) - N(y_v, z_u) > 0$. 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 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*. Obviously, this method is only suitable for sparse graphs. If t is in the order of O(n) or higher, the size of labels is more than O(n^2) and the query time is O(log$n$). Moreover, O(n^3) time is needed to generate labels, worse than any traditional matrix-based method.

The 2-*hop labeling* proposed by Cohen et al. [9] assigns to each node v 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 $C_{in}(v) \cap C_{out}(v) \neq \phi$. 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^4). An important issue with regard to the 2-hop approach is the complexity of its labeling process. Finding optimum 2-hop labeling is equivalent to solving the weighted set covering problem, which is NP-hard. So a heuristic method is suggested in [9], which greedily finds the largest uncovered submatrix in the transitive closure matrix in each step. However, it is still an extremely time consuming process for massive graphs.

There are some other graph labeling methods, such as the method discussed in [22, 23], which reduces 2-hop's labeling complexity from O(n^4) to O(n^3), but is still not applicable to massive graphs. The method proposed in [6] is a geometry-based algorithm to find high-quality 2-hop covers. It has the same theoretical computational complexities as the method discussed in [27] and is only applicable for sparse graphs, too.

In the following table, we compare our labeling approach with existing approaches.

In the above table, *graph-traversal* represents no-labeling at all, which imposes no space overhead, but requires searching a whole graph to answer reachability queries. k_1 is the largest length of an interval sequence of the interval-based method [1], and k_2 is the number of the disjoint chains produced by Chen's method [5]. t is the number of non-tree edges and b is the width of a DAG. Finally, using the method discussed in [10], a matrix multiplication can be done in O($n^{2.376}$) time.

	query time	labeling time	space overhead
graph-traversal	O(e)	0	0
graph-decomposition (Jagadish)	O(log$b$)	O(n^3)	O(bn)
interval-based	O(logk_1)	O(k_1e)	O(k_1n)
dual-I	O(1)	O($n + e + t^3$)	O($n + t^2$)
dual-II	O(log$t$)	O($n + e + t^3$)	O($n + t^2$)
2-hop	O($e^{1/2}$)	O(n^4)	O($ne^{1/2}$log$n$)
matrix-multiplication	O(1)	O($n^{2.376}$)	O(n^2)
graph-decomposition (Chen et al.)	O(logk_2)	O(k_2e)	O(k_2n)
ours	O(log$b$)	O(be)	O(bn)

3. GENERAL SPANNING TREE AND TREE ENCODING

Our method is based on the DAG encoding. For a cyclic graph (a graph containing cycles), we can find all the strongly connected components (*SCC*) in linear time [24] and then collapse each of them into a representative node. Clearly, all of the nodes in an *SCC* is equivalent to its representative as far as reachability is concerned (see pp. 567 - 569 in [16]).

Definition 1. (*general spanning trees*)**.** Let G be a DAG. A tree (forest) T is called a general spanning tree if the following two conditions are satisfied:

1. T covers G, i.e., for each node $v \in G$, we have $v \in T$.
2. For each edge $u \rightarrow v$ in T, there exists a path from u to v in G. □

Since an edge $u \rightarrow v$ in G is also a path, a traditional spanning tree is a special case of general spanning trees.

As an example, consider the graph G shown in Fig. 1(a), for which a general spanning tree T can be found as shown in Fig. 1(b). In T, special attention should be paid to the edge $h \rightarrow i$, which corresponds to a path from h to i in G. We also notice that the number of the leaf nodes in T is 3 while any (traditional) spanning tree of G has at least 4 leaf nodes (see Fig. 1(c) for illustration.)

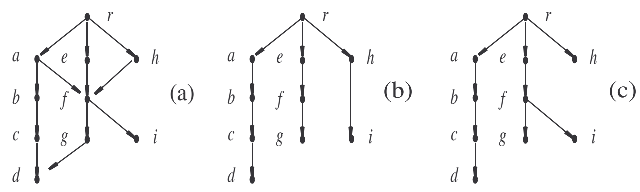
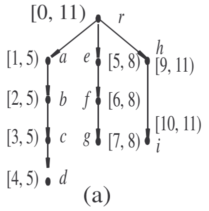


Fig. 1. Illustration for general spanning trees

As with Dual-I labeling [27], we will assign each node v in T an interval [*start*, *end*), where *start* is v's preorder number and *end* - 1 is the largest preorder number among all the nodes in $T[v]$ (the subtree rooted at v). So another node u labeled [*start'*, *end'*) is a descendant of v (with respect to T) iff *start'* $\in$ [*start*, *end*) [27]. See Fig. 2(a).

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 $[a', b')$, which shows that v is a descendant of u. 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 $[a, b)$.



r:	[0, 11)	e:	[4, 5)[5, 8)[10, 11)
a:	[1, 5)[6, 8)[10, 11)	f:	[4, 5)[6, 8)[10, 11)
b:	[2, 5)	g:	[4, 5)[7, 8)
c:	[3, 5)	h:	[4, 5)[6, 8)[9, 11)
d:	[4, 5)	i:	[10, 11)

(b)

Fig. 2. Illustration for tree labeling

In a next step, for each node v in T, we will generate an interval sequence $L[v]$ (along a reverse topological order of G) by using the method discussed in [4]. $L[v]$ satisfies the following properties:

(1) Let $L(v) = [a_1, b_1), ..., [a_l, b_l)$ for some l. Then, for any i, $j \in \{1, ..., l\}$, $b_i \leq a_j$ if $i < j$. That is, $[a_i, b_i) \prec [a_j, b_j)$ 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 with respect to G. There exists an interval $[a_i, b_i)$ ($1 \leq i \leq l$) in $L(v_l)$ such that $a \in [a_i, b_i)$.

(3) $l \leq k$, where k is the number of the leaf nodes in T.

All the interval sequences make up a label space, as shown in Fig. 2(b).

In this way, the space overhead is decreased to O(kn) with the query time being bounded by O(log$k$).

An interesting question is: can we always find a general spanning tree with the least number of leaf nodes in an efficient way? In the next section, we answer this question.

4. MAIN ALGORITHM

The main idea of our method is to decompose G into a minimal set of disjoint chains such that on each chain if u is above v then there exists a path from u to v in G. We follow Chen's method, working in three phases: DAG stratification, chain generation, and virtual node resolution. But our main procedures are quite different from those of Chen's.

First, for the chain generation, we distinguish between two kinds of virtual nodes and handle them in different ways.

Second, for the virtual node resolution, a new data structure, the so-called *combined alternating graph*, is constructed so that the number of virtual nodes resolved at each level is maximized.

In the following, we first present our algorithm to find disjoint chains in 4.1. Then, we briefly describe how to generate a general spanning tree in 4.2. In 4.3, we prove the correctness of our algorithm and analyze its computational complexities. (The analysis of Chen's algorithm is shifted to the Appendix.)

4.1 Finding a minimal set of disjoint chains

As with Chen's algorithm, our algorithm contains three phases: DAG stratification, chain generation, and virtual node resolution. In the following, they will be described in great detail.

4.1.1 DAG stratification

In the first phase, a DAG $G(V, E)$ will be stratified into several levels $V_0, ..., V_{h-1}$ such that $V = V_0 \cup ... \cup V_{h-1}$ and each node in V_i has its children appearing only in $V_{i-1}, ..., V_0$ ($i = 1, ..., h - 1$), where h is the height of G, i.e., the length of the longest path in G. For each node v in V_i, its level is said to be i, denoted $l(v) = i$. In addition, $C_j(v)$ ($j < i$) represents a set of links with each pointing to one of v's children, 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 $C_{i_1}(v) \cup ... \cup C_{i_k}(v)$. Let $V_i = \{v_1, v_2, ..., v_l\}$. We will use $\boldsymbol{C}_j^i$ ($j < i$) to represent $C_j(v_1) \cup ... \cup C_j(v_l)$.

In addition, we use $B_j(v)$ to represent a set of links with each pointing to one of v's parents, which appears in V_j.

4.1.2 Chain generation

In the second phase, a series of (undirected) bipartite graphs [2, 15] will be constructed. In this process, some virtual nodes may be introduced into the levels V_i ($i = 1, ..., h - 2$). Especially, we distinguish between two kinds of virtual nodes. One is the virtual nodes created for actual nodes; and the other is the virtual nodes generated for virtual nodes. They will be handled differently.

We begin our discussion with a summarization of some important concepts related to bipartite graphs, which are needed to define virtual nodes.

Definition 2. (*concepts related to matching*, [2]) Let $G(V, E)$ be a bipartite graph. Let M be a maximum matching of G. A node v is said to be *covered* by M, if some edge of M is incident to v. We will also call an uncovered node *free*. A path or cycle is *alternating*, relative to M, if its edges are alternately in $E\backslash M$ and M. A path is an *augmenting path* if it is an alternating path with free origin and terminus. □

In addition, it is well known that using the Hopcroft-Karp algorithm [15] a maximum matching of G can be found in $O(|E|\sqrt{|V|})$ time.

Also, the following symbols are also used for ease of explanation:

$V_i' = V_i \cup \{$virtual nodes introduced into $V_i\}$.

$\boldsymbol{C}_i = \boldsymbol{C}_{i-1}^i \cup \{$all the new edges from the nodes in V_i to the virtual nodes introduced into $V_{i-1}\}$

$G(V_i, V_{i-1}'; \boldsymbol{C}_i)$ - the bipartite graph containing V_i and V_{i-1}'.

Definition 3. (*virtual nodes for actual nodes*) Let $G(V, E)$ be a DAG, divided into $V_0, ..., V_{h-1}$ (i.e., $V = V_0 \cup ... \cup V_{h-1}$). Let M_i be a maximum matching of the bipartite graph $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$ and v be a free actual node (in V_{i-1}') relative to M_i ($i = 1, ..., h - 1$). Add a virtual node v' into V_i. In addition, for each node $u \in V_{i+1}$, a new edge $u \to v'$ will be created if one of the following two conditions is satisfied:

1. $u \to v \in E$; or
2. There exists an edge (v_1, v_2) covered by M_i such that v_1 and v are connected through an alternating path relative to M_i; and $u \in B_{i+1}(v_1)$ or $u \in B_{i+1}(v_2)$.

v is called the source of v', denoted $s(v')$. □

A virtual edge from v' to v is also generated to indicate the relationship between v and v'. Besides, a new edge $u \to v'$ will be marked with '*directly connectable*' if one of the following conditions are satisfied:

3. $u \to v \in E$; or
4. There is an alternating path of length 1, which connects v_1 and v. That is, $v_1 \to v \in E$.

We mark these edges with 'directly connectable' because it is possible for us to directly connect u and v to remove v'. to facilitate the virtual node resolution process.

The following example helps for illustration.

Example 1. Consider the graph shown in Fig. 3(a). It can be divided into three levels as shown in Fig. 3(b). The bipartite graph made up of V_1 and V_0, $G(V_1, V_0; \boldsymbol{C}_1)$, is shown in Fig. 3(c) and a possible maximum matching M_1 of it is shown in Fig. 3(d).

Relative to M_1, we have two free nodes i and a. For them, two virtual nodes i' and a' will be constructed. Then, $V_1' = \{b, e, h, i', a'\}$. In addition, four new edges (d, i'), (d, a'), (g, i'), and (g, a') will be constructed. But all of them will not be marked with 'directly connectable'.

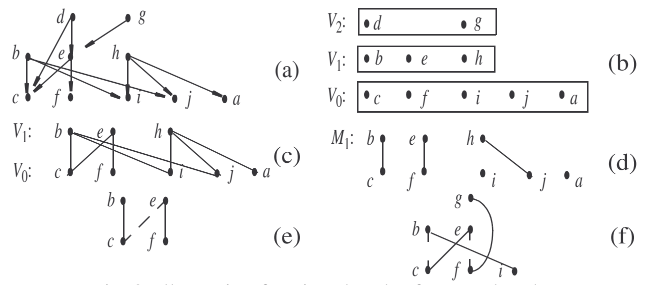


Fig. 3. Illustration for virtual nodes for actual nodes

The motivation of constructing such a virtual node (e.g., i') is that it is possible to connect f to d or g to form part of a chain if we transfer the edges on an alternating path: $b \to c \to e \to f$ (see Fig. 3(e), where a solid edge represents an edge belonging to M_1 while a dashed edge to $\boldsymbol{C}_1\backslash M_1$), or $h \to j \to b \to c \to e \to f$. Then, we can connect d or g to f, as well as b or h to i without increasing the number of chains, as illustrated in Fig. 3(f). This can be achieved by the virtual node resolution process (see 4.1.3).

Definition 4. (*virtual nodes for virtual nodes*) Let M_i be a maximum matching of the bipartite graph $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$ and v' be a free virtual node (in V_{i-1}') relative to M_i ($i = 1, ..., h - 1$). Add a virtual node v'' into V_i. Set $s(v'')$ to be $w = s(v')$. Let $l(w) = j$. For each node $u \in V_{i+1}$, a new $u \to v'$ will be

created if there exists an edge (v_1, v_2) covered by M_{j+1} such that v_1 and w are connected through an alternating path relative to M_{j+1}; and $u \in B_{i+1}(v_1)$ or $u \in B_{i+1}(v_2)$. □
Again, a virtual edge from v'' to v' will be generated.

Example 2. Consider the graph shown in Fig. 4(a).
This graph can be divided into four levels as shown in Fig. 4(b). The first bipartite graph consisting of V_1 and V_0, $G(V_1, V_0; \boldsymbol{C}_1)$, is shown in Fig. 4(c) and a possible maximum matching M_1 of it is shown in Fig. 4(d). Relative to M_1, we have a free node f. For it, a virtual nodes f' will be constructed. Then, $V_1' = \{b, f', d, h\}$ (see Fig. 4(e)). Assume that the maximum matching found for $G(V_2, V_1'; \boldsymbol{C}_2)$ is as shown in Fig. 4(f). A virtual node f'' for f' will be established. So $V_2' = \{f'', e, g\}$. Especially, we are able to connect node f'' and node p for the following reason:
i) $s(f'') = s(f') = f$;
ii) $(b, c) \in M_1$;
iii) f is connected to b through an alternating path: $f \rightarrow b$; and
iv) $p \in B_3(c)$.
The corresponding bipartite graph $G(V_3, V_2'; \boldsymbol{C}_3)$ is shown in Fig. 4(g). The unique maximum matching of $G(V_3, V_2'; \boldsymbol{C}_3)$ is shown in Fig. 4(h). □

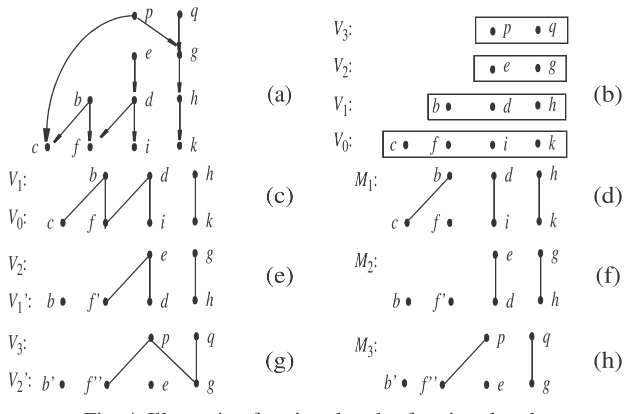


Fig. 4. Illustration for virtual nodes for virtual nodes

By using virtual nodes, a set of chains can be generated by doing the following two steps.
(1) The first bipartite graph $G(V_1, V_0; \boldsymbol{C}_1)$ is made up of V_0, V_1, and $\boldsymbol{C}_1$ (= $\boldsymbol{C}_0^1$). Let M_1 be a maximum matching of $G(V_1, V_0; \boldsymbol{C}_1)$. Construct a set of virtual nodes D_1 for all the free nodes relative to M_1.
(2) All the other bipartite graphs will be recursively established as follows. Let $V_{i-1}' = V_{i-1} \cup D_{i-1}$ ($1 < i \leq h - 1$; $V_0' = V_0$). Let $\boldsymbol{C}_i = \boldsymbol{C}_{i-1}^i \cup$ {all the new edges incident to the virtual nodes in D_{i-1}}. Then, the ith bipartite graph $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$ is made up of V_{i-1}', V_i, and $\boldsymbol{C}_i$. Find a maximum matching M_i of $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$. Construct D_i, the set of the virtual nodes for all the free nodes relative to M_i.

The result $M_1 \cup M_2 \cup ... \cup M_{h-1}$ is a set of chains, which may contain some virtual nodes.

Example. 3. Continued with Example 1. The bipartite graph made up of V_2 and V_1' is shown in Fig. 5(a).
A possible maximum matching M_2 is shown in Fig. 5(b). So $M_1 \cup M_2$ is a set of chains as shown in Fig. 5(c). □

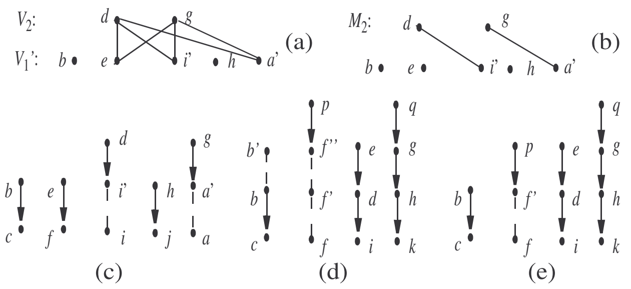


Fig. 5. Bipartite graph, maximum matching and chains

In the same way, by unifying M_1, M_2, and M_3 shown in Fig. 4, we get a set of disjoint chains shown in Fig. 5(d).

4.1.3 Virtual node resolution

In the third phase, we will remove all the virtual nodes. This will be done top-down level by level; and at each level any virtual node, which does not have a parent along a chain, will be simply eliminated. In addition, we call a virtual node v' a transit virtual node if one of the following two conditions is satisfied.
1. Let u, v', w be three consecutive nodes on a chain. $u \rightarrow v'$ is a marked edge (i.e., a directly connectable edge); or
2. w is a virtual node.

In both cases, we connect u and w and then remove v'. It is because in case (1), both u and w are actual nodes and we have $u \rightarrow w \in E$ or there exists a actual node x such that $u \rightarrow x \in E$ and $x \rightarrow w \in E$. In case (2), w is a virtual node, working as a 'transfer' of reachability.
For example, since node f' in Fig. 5(d) is a virtual node, node f'' is a transit virtual node. It can be directly removed, leading to a set of chains as shown in Fig. 5(e). But node f' cannot be removed in this way since it is not a transit virtual node.
In the following, we discuss how to resolve a non-transit virtual node, for which more effort is needed.
First, we define a new concept.

Definition 5. (*alternating graph*) Let M_i be a maximum matching of $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$. The alternating graph $\vec{G}_i$ with respect to M_i is a directed graph with the following sets of nodes and edges:

$$V(\vec{G}_i) = V_i \cup V_{i-1}', \text{ and}$$

$$E(\vec{G}_i) = \{u \rightarrow v \mid u \in V_{i-1}', v \in V_i, \text{ and } (u, v) \in M_i\} \cup$$
$$\{v \rightarrow u \mid u \in V_{i-1}', v \in V_i, \text{ and } (u, v) \in \boldsymbol{C}_i \backslash M_i\}. \ \square$$

Example 4. Consider the graph shown in Fig. 3(a) once again. Relative to M_1 of $G(V_1, V_0; \boldsymbol{C}_1)$ shown in Fig. 3(d), nodes i and a are two free nodes. The alternating graph $\vec{G}_1$ with respect to M_1 is shown in Fig. 6(a).
It is redrawn in Fig. 6(b) for a clear explanation. □

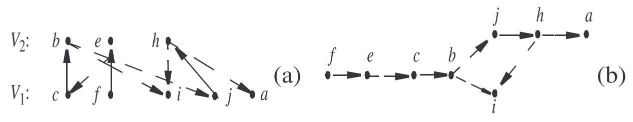


Fig. 6. An alternating graph

In order to resolve the non-transit virtual nodes in V_i', we will combine $\vec{G}_{i+1}$ and $\vec{G}_i$ by connecting some nodes v' in $\vec{G}_{i+1}$ to some nodes u in $\vec{G}_i$ if the following conditions are satisfied.

(i) v' is a non-transit virtual node appearing in V_i'. (Note that $V(\vec{G}_{i+1}) = V_{i+1} \cup V_i$'.)

(ii) There exist a node x in V_{i+1} and a node y in V_i such that $(x, v') \in M_{i+1}$, $x \rightarrow y \in \boldsymbol{C}_{i+1}$, and $(y, u) \in M_i$.

We denote this combined graph by $\vec{G}_{i+1} \oplus \vec{G}_i$.

For illustration, consider $G(V_2, V_1'; \boldsymbol{C}_2)$ shown in Fig. 5(a). Assume that the found maximum matching M_2 is as shown in Fig. 5(b). Then, the alternating graph $\vec{G}_2$ (with respect to M_2) is a graph shown in Fig. 7(a). $\vec{G}_2 \oplus \vec{G}_1$ is shown in Fig. 7(b). Note that i' and a' are two non-transit virtual nodes.

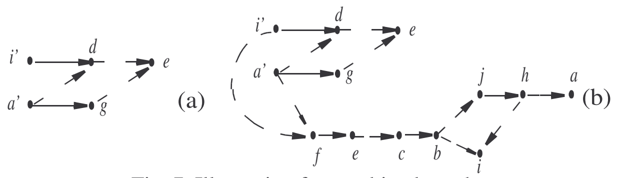


Fig. 7. Illustration for combined graph

We also notice that a node in $\vec{G}_{i+1}$ and a node in $\vec{G}_i$ may share the same node name. But they will be handled as different nodes. For example, node e in $\vec{G}_2$ and node e in $\vec{G}_1$ are different.

In Fig. 7(b), we connect node a' (in $\vec{G}_2$) to node f (in $\vec{G}_1$) for the following reason.

(1) a' is a non-transit virtual node introduced into V_1.

(2) $(g, a') \in M_2$, $g \rightarrow e \in \boldsymbol{C}_2$, and $(e, f) \in M_1$.

As mentioned above, we connect a' to f since it is possible for us to transfer the edges on an alternating path (relative to M_1) starting from node f (relative to M_1) and terminating at free node i or a (in V_0), which will make i or a covered without increasing the number of chains.

The same analysis applies to node i' (in $\vec{G}_2$), which is also connected to node f (in $\vec{G}_1$).

In order to resolve as many non-transit virtual nodes (appearing in V_i') as possible, we need to find a maximum set of node-disjoint paths (i.e., no two of these paths share any nodes), each starting at a non-transit virtual node (in $\vec{G}_{i+1}$) and ending at a free node in $\vec{G}_{i+1}$, or ending at a free node in $\vec{G}_i$. For example, to resolve a' and i', we need first to find two paths in the above combined graph, as shown in Fig. 8(a).

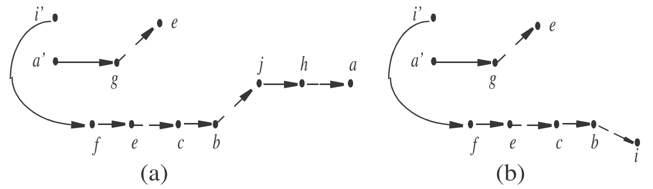


Fig. 8. Illustration for node-disjoint paths

(In Fig. 8(b), we show another two node-disjoint paths.)

By transferring the edges on such a path, the corresponding virtual node can be removed as follows.

(1) Let $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_k$ be a found path. Transfer the edges on the path.

(2) If v_k is a node in $\vec{G}_{i+1}$, we simply remove the corresponding virtual node v_1.

(3) If v_k is a node in $\vec{G}_i$, connect the parent of v_1 along the corresponding chain to v_2. Remove v_1.

For instance, by transferring the edges on the path from a' to e (in $\vec{G}_2$) in Fig. 8(a), we will connect g to e (in $\vec{G}_2$). a' will be removed. By transferring the edges on the path from i' to a in Fig. 8(a), we will connect h (in $\vec{G}_1$) to a, b to j, e to c, and d to f. Then, i' is removed. Note that a is in $\vec{G}_1$ and d is the parent of i' along a chain (see Fig. 5(c)). In this way, we will change the chains shown in Fig. 5(c) to the chains shown in Fig. 9(a) with all the virtual nodes being removed. The number of chains is still 5.

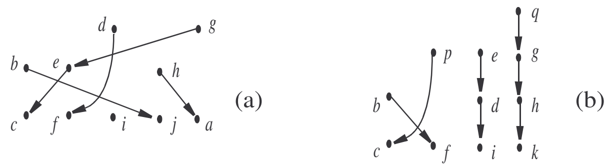


Fig. 9. Minimum sets of chains

By resolving node f' in the chain set shown in Fig. 5(e), we will get a set of disjoint chains shown in Fig. 9(b).

It remains to show how to find a maximal set of node-disjoint paths in $\vec{G}_{i+1} \oplus \vec{G}_i$.

For this purpose, we define a maximum flow problem over $\vec{G}_{i+1} \oplus \vec{G}_i$ (with multiple sources and sinks) as follows.

1) Each non-transit virtual node in $\vec{G}_{i+1}$ is designated as a *source*. Each free node (in $\vec{G}_{i+1}$) relative to M_{i+1}, or free node (in $\vec{G}_i$) relative to M_i is designated as a *sink*.

2) Each edge $u \rightarrow v$ is associated with a capacity $c(u, v) = 1$. (If (u, v) is not an edge in $\vec{G}_{i+1} \oplus \vec{G}_i$, $c(u, v) = 0$.)

Generally, to find a maximum flow in a network, we need $O(n^3)$ time [11, 13, 17]. However, a network as constructed above is a 0-1 network. In addition, for each node v, we have either $d_{in}(v) \leq 1$ or $d_{out}(v) \leq 1$, where $d_{in}(v)$ and $d_{out}(v)$ represent the indegree and outdegree of v in $\vec{G}_{i+1} \oplus \vec{G}_i$, respectively. It is because each path in $\vec{G}_{i+1} \oplus \vec{G}_i$ is an alternating path relative to M_{i+1} or relative to M_i. So each node except sources and sinks is an end node of an edge covered by M_{i+1} or by M_i. As shown in ([14], Theorem 6.3 on page 120), it needs only O($\sqrt{n}\, e$) time to find a maximum flow in such kind of networks. Especially, a maximum flow exactly corresponds to a maximal set of disjoint paths (see the proof of Lemma 6.4 in [14], page 120.)

According to the above discussion, we give the following algorithm for resolving virtual nodes. We assume that each virtual node has a parent along a chain. Otherwise, it can be simply eliminated.

Algorithm *virtual-resolution*(S)

input: S - a chain set obtained by executing the *chain generation* process.

output: a set of chains containing no virtual nodes.

```
begin
1.   for $i = h - 2$ downto 1 do
2.   { for any transit virtual node $v'$ in $V_i'$ do
3.     {
4.       let $u$, $v'$, $w$ be three consecutive nodes on a chain;
5.       connect $u$ and $w$;
6.     }
7.     construct $\vec{G}_{i+1} \oplus \vec{G}_i$; (*Begin to handle non-transit
       virtual nodes.*)
8.     find a maximal set of node disjoint paths: $P_1$, ... $P_l$;
9.     for $j$ = 1 to $l$ do
10.    { let $P_j = v_1 \to v_2 \to ... \to v_k$;
11.        if $v_k$ is a free node relative to $M_i$ then
12.          {transfer the edges on $P_j$; remove $v_1$;}
13.        else (* $v_k$ is a free node relative to $M_{i-1}$.*)
14.          {let $u$ be a node such that $(u, v_1) \in M_i$;
15.           transfer the edges on $P_j$; remove $v_1$;
16.           connect $u$ to $v_2$;
17.          }
18.    removed any unsolved virtual node;
19.  }
end
```

In the main **for**-loop of the above algorithm, we first handle transit virtual nodes (lines 2 - 6). Then, we construct $\vec{G}_i \oplus \vec{G}_{i-1}$ to resolve all the non-transit virtual nodes (see line 7.) For this purpose, we search for a maximal set of node disjoint paths (see line 8). We also distinguish between two kinds of node disjoint paths: paths ending at a free node relative to M_i, and paths ending at a free node relative to M_{i-1}. For the first kind of paths, we simply transfer the edges on a path and then remove the corresponding virtual node (see line 12). For the second kind of paths, we need to do something more to connect the parent of the corresponding virtual node (along the chain) to the second node of the path (see line 16). In line 18, we remove all those virtual nodes, which cannot be resolved. Each of such virtual nodes leads to splitting of a chain into two chains.

Note that removing a transit virtual node will not increase the number of chains. Also, resolving a non-transit virtual node using a node disjoint path does not lead to a chain splitting. So the number of increased chains during the virtual node resolution process is minimum since the number of node disjoint paths is maximum.

4.2 Construction of general spanning trees

Using the algorithm discussed in 4.1, a general spanning tree can be easily generated. We need only to slightly change the Algorithm *virtual-resolution*(S). After all the virtual nodes in V_i are resolved, connect each of all those nodes, which do not have a parent anymore along a chain, to one of its parents in V_{i+1}.

For example, by resolving the virtual nodes on the chains shown in Fig. 5(c), a general spanning tree will be created as shown in Fig. 10, where node r is a virtual root. We also notice that edge $d \to f$ corresponds to a path: $d \to e \to f$ in the graph shown in Fig. 3(a).

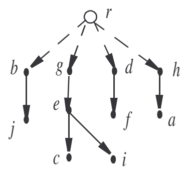


Fig. 10. A general spanning tree

4.3 Correctness and computational complexities

In this subsection, we prove the correctness of our algorithm and analyze its computational complexities.

4.3.1 Correctness

Lemma 1. For a DAG G, which can be divided into two levels: V_0 and V_1, the number of disjoint chains found by our algorithm is minimum.

Proof. In this case, our algorithm will use the Hopcroft-Karp algorithm to find a maximum matching M_1 of $G = G(V_1, V_0; \boldsymbol{C}_1)$. Let $free_{M_1}(V_i)$ (i = 0, 1) be the set of the free nodes in V_i, relative to M_1. Then, the set of the disjoint chains equals $M_1 \cup free_{M_1}(V_0) \cup free_{M_1}(V_1)$. A maximum antichain (a set of pairwise unreachable nodes in G) can be constructed as follows.

Let U_i (i = 0, 1) be a subset of V_i such that each node in it is covered by M_1. Let W be all the nodes appearing on the alternative paths starting from a node in $free_{M_1}(V_0)$. Then, $free_{M_1}(V_0) \cup free_{M_1}(V_1) \cup (U_0 \cap W) \cup (U_1 \backslash (U_1 \cap W))$ makes up a maximum anti-chain. To see this, we need to show that any node v in $free_{M_1}(V_1)$ is not connected to any node u in $U_0 \cap W$. Otherwise, we would have an augmenting path with origin v and terminus u, contradicting that M_1 is a maximum matching (see Theorem 5.1.4 in [2], page 57). Obviously, any node in $free_{M_1}(V_0)$ is not connected to any node in $U_1 \backslash (U_1 \cap W)$. □

Proposition 1. The number of the chains generated for a DGA by our algorithm is minimum.

Proof. We will prove the proposition by induction on h.

Initial step. When $h = 2$, the proposition holds according to Lemma 1.

Induction step. Assume that for any DAG of height k, the proposition holds. Now we consider the case when $h = k + 1$. First, we construct another graph G' from $G(V, E)$ as follows:

1. Stratify G, dividing V into $V_0, ..., V_{h-1}$ (i.e., $V = V_0 \cup ... \cup V_{h-1}$).
2. Find a maximum matching M_1 of $G(V_1, V_0; \boldsymbol{C}_1)$. Construct virtual nodes for all the nodes in $free_{M_1}(V_0)$, and add them into V_1. Then, we remove V_0.

So G' is of height k. According to the induction hypothesis. A minimal set S of disjoint chains can be found. We partition S into two subsets S_1 and S_2. Any chain in S_1 ends at a node in V_1' and any chain in S_2 ends at a node in V_i $(1 < i \le k)$. Resolving the virtual nodes in S_1 by constructing $\vec{G}_2 \oplus \vec{G}_1$, and connecting $u \in V_1$ to $v \in V_0$ for every pair (u, v) if (u, v) is still in M_1, we will get a set S' of disjoint chains for G. To show that S' is minimum, we need to explain that for any free node v in $free_{M_1}(V_0)$, which cannot be connected to any chain in S_1, there exists no path from any terminating node u of a chain in S_2 to v. Otherwise, there must exist a sequence of nodes:

$$v^{(0)}, v^{(1)}, ..., v^{(m)}$$

satisfying the following conditions:

i) $v^{(0)} = v$;

i) $v^{(i)}$ is a virtual node constructed for $v^{(i-1)}$ $(1 \le i \le m)$, or for a node w which is connected to $v^{(i-1)}$ through an alternating path relative to M_i (a maximum matching of $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$; and

iii)$v^{(m)}$ is connected to u.

Then, by solving the virtual nodes using Algorithm *virtual-resolution*(S), u and v will appear on a same chain, or the corresponding chain (with u as the terminating node) will be extended downwards. Contradiction. □

Finally, we note that an antichain of G can be constructed top-down level by level by using the method shown in the proof of Lemma 1.

4.3.2 Computational complexity

Now we analyze the computational complexities of our algorithm. The cost of the whole process can be divided into three parts:

- cost_1: the time for stratifying a DAG.
- cost_2: the time for generating disjoint chains, which may contain virtual nodes.
- cost_3: the time for resolving virtual nodes.

As shown in [5], cost_1 is bounded by $O(n + e)$.

cost_2 mainly contains two parts. One part: cost_{21} is the time for finding a maximum matching of every $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$ ($i = 1, ..., h - 1$; $V_0' = V_0$). The other part: cost_{22} is the time for checking whether, for each actual free node appearing in V_{i-1}', there exists an edge (v_1, v_2) covered by M_i such that v_1 and v are connected through an alternating path relative to M_i.

The time for finding a maximum matching of $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$ is bounded by

$$O(\sqrt{|V_i| + |V_{i-1}|} \cdot |\boldsymbol{C}_i|). \text{ (see [15])}$$

Therefore, cost_{21} is bounded by

$$O(\sum_{i=1}^{h-1} (\sqrt{|V_i| + |V_{i-1}|} \cdot |\boldsymbol{C}_i|))$$

$$\le O(\sqrt{b}\sum_{i=1}^{h-1} b \cdot |V_i|) = O(bn\sqrt{b}).$$

cost_{22} can be analyzed as follows. We construct a small boolean $n_i \times m_i$ matrix A_i, where n_i is the number of free actual nodes in V_{i-1} and m_i is the number of all the covered actual nodes in V_i. Each entry $a_{jk} = 1$ in A_i indicates that there exists an alternating path (relative to M_i) connects node j and k. Using the algorithm discussed in [10] for matrix multiplication, cost_{22} can be estimated by

$$O(\sum_{i=1}^{h-1} |V_{i-1}|^{2.376}) \le O(bn\sqrt{b}).$$

During the virtual-resolution process, the virtual nodes are resolved level by level. At each level, the number of the nodes in $\vec{G}_i \oplus \vec{G}_{i-1}$ is bounded by $O(|V_{i+1}| + 2|V_i'| + |V_{i-1}'|)$; and the number of its edge is $O(|\boldsymbol{C}_i| + |\boldsymbol{C}_{i-1}|)$. So, the time for finding a maximal set of node-disjoint paths in $\vec{G}_i \oplus \vec{G}_{i-1}$ is bounded by $O(\sqrt{|V_{i+1}| + 2|V_i'| + |V_{i-1}'|}\ (|\boldsymbol{C}_i| + |\boldsymbol{C}_{i-1}|))$. So the total cost of the virtual node resolution is in the order of

$$\sum_{i=2}^{h-1} \sqrt{|V_{i+1}'| + 2|V_i'| + |V_{i-1}'|} \cdot (|\boldsymbol{C}_i| + |\boldsymbol{C}_{i-1}|)$$

$$= O(\sqrt{b}\sum_{i=2}^{h-1} b \cdot |V_{i+1}|) = O(bn\sqrt{b}). \quad \square$$

From the above analysis, we get the following proposition.

Proposition 2. The time complexity of the whole process to decompose a DAG into a minimized set of disjoint chains is bounded by $O(bn\sqrt{b})$. □

The space complexity of the whole process is bounded by $O(e + bn)$ since the number of the newly added edges in each bipartite graph $G(V_i, V_{i-1}'; \boldsymbol{C}_i')$ is bounded by $O(b|V_{i-1}|)$, and the size of each matrix A_i is bounded by $O(|V_{i-1}|^2)$.

5. CONCLUSION

In this paper, a new concept of general spanning trees is proposed and an efficient algorithm for finding such a tree in a DAG is discussed. The main idea of the algorithm is to decompose a DAG into a minimal set of node-disjoint paths. Our scheme is inspired by Chen's method. However, the main procedures in our algorithm are quite different from theirs. First, we distinguish between two kinds of virtual nodes and handle them in different ways when generating chains. Second, for the virtual node resolution, a new data structure, the so-called combined alternating graph, is constructed so that the number of virtual nodes resolved at each level is maximized. So we can always find a general spanning tree with the least number of leaf nodes, which enable us to effectively reduce label spaces.

REFERENCES

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

[2] A.S. Asratian, T. Denley, and R. Haggkvist, *Bipartite Graphs and their Applications*, Cambridge University, 1998.

[3] K.S. Booth and G.S. Leuker, "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms," *J. Comput. Sys. Sci.*, 13(3):335-379, Dec. 1976.

[4] Y. Chen, Graph Decomposition and Recursive Closures, in *Proc. CaiSE 2003 Forum at 15th Conf. on Advanced Information Systems Engineering*, June 2003, Klagenfurt/Velden, Austria, pp. 5-8.

[5] Y. Chen and Y. Chen, An Efficient Algorithm for Answering Graph Reachability Queries, *Proceedings of ICDE*, 2008, pp. 893 - 902.

[6] 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.

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

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

[9] 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.

[10] D. Coppersmith, and S. Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation*, vol. 9, pp. 251-280, 1990.

[11] T.H. Corman, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, 2nd. ed., McGraw-Hill Book Company, Boston, 2007.

[12] R.P. Dilworth, A decomposition theorem for partially ordered sets, *Ann. Math.* **51** (1950), pp. 161-166.

[13] E.A. Dinic, Algorithm for solution of a problem of maximum flow in a network with power estimation, Soviet Mathematics Doklady, 11(5):1277-1280, 1970.

[14] S. Even, *Graph Algorithms*, Computer Science Press, Inc., Rockville, Maryland, 1979.

[15] J.E. Hopcroft, and R.M. Karp, An n2.5 algorithm for maximum matching in bipartite graphs, *SIAM J. Comput*. 2(1973), 225-231.

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

[17] A.V. Karzanov, Determining the Maximal Flow in a Network by the Method of Preflow, *Soviet Math. Dokl.*, Vol. 15, 1974, pp. 434-437.

[18] T. Keller, G. Graefe and D. Maier, "Efficient Assembly of Complex Objects," *Proc. ACM SIGMOD Conf.*, Denver, Colo., 1991, pp. 148-157.

[19] W. Kim, "Object-Oriented Database Systems: Promises, Reality, and Future," *Proc. 19th VLDB conf.*, Dublin, Ireland, 1993, pp. 676-687.

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

[21] E.L. Lawler, *Combinatorial Optimization and Matroids*, Holt, Rinehart, and Winston, New York (1976).

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

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

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

[25] 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.

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

[27] 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.

[28] S. Warshall, "A Theorem on Boolean Matrices," *JACM*, 9. 1(Jan. 1962), 11 - 12.

[29] D. West, Parameters of partial orders and graphs: packing, covering and representation, in: I. Rival (ed.), Graphs and Orders, Dordrecht-Reidel, 1985, pp. 267-350.

[30] C. Zhang, J. Naughton, D. DeWitt, Q. Luo and G. Lohman, "On Supporting Containment Queries in Relational Database Management Systems, in *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, California, USA, 2001.

[31] 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.

APPENDIX

In the Appendix, we analyze Chen's algorithm [5] and show why it fails to find a minimal set of disjoint chains.

As mentioned earlier, Chen's algorithm works in three phases: DAG stratification, chain generation, and virtual node resolution. However, it suffers from the following two problems.

1. The virtual nodes for actual nodes and the virtual nodes for virtual nodes are handled in the same way. So the reachability between nodes cannot be properly transferred by using virtual nodes.
2. The number of resolved virtual nodes at each level is not maximized.

To see the first problem, we give Chen's definition for virtual nodes below.

Definition (*virtual nodes according to M_i*, [5]) Let $G(V, E)$ be a DAG, divided into $V_0, ..., V_{h-1}$ (i.e., $V = V_0 \cup ... \cup V_{h-1}$). Let v be a free node relative to M_i. Add a virtual node v' into V_i ($i = 1, ..., h - 1$), labeled as follows.

1. If there exist some covered nodes $u_1, ..., u_k$ (relative to M_i) in V_i' such that each u_g ($g = 1, ..., k$) shares a covered parent node w_g (i.e., $(w_g, u_g) \in M_i$) with v, label v' with
   $v[(w_1, \{(n_{11}, S_{11}), ..., (n_{1j_1}, S_{1j_1})\}), ..., (w_k, u_k, \{(n_{k1}, S_{k1}), ..., (n_{kj_k}, S_{kj_k})\})]$,
   where n_{gj} ($g = 1, ..., k$; $j = 1, ..., j_g$) is an odd number to indicate a position on the alternating path starting at w_g, and S_{gj} is a set containing all the parents of the node pointed to by n_{gj}, which appear in V_{i+1}.
2. If no such a covered node exists, v' is labeled with $v[\ ]$. $\square$

In addition, for a virtual node v' (generated for v), an edge $u \to v'$ is established for every $u \in S_{11} \cup ... \cup S_{1j_1} \cup ... \cup S_{k1} ... \cup S_{kj_k}$. v' will also inherit the edges incident to v except the edges from a node in V_i to v. That is, for each parent w of v, an edge $w \to v'$ will be added if w does not appear in V_i. A virtual edge $v' \to v$ will also be constructed.

In terms of this definition, the virtual node f'' in Fig. 4(h) will not be connected to node p since it is not a parent of node f. Therefore, the chain set produced by Chen's algorithm for the graph shown in Fig. 4(a) will contain 5 chains as shown in Fig. 11. But the graph can be decomposed into 4 chains as shown in Fig. 9(b).

Now we analyze the second problem of Chen's algorithm, which resolves virtual nodes level by level as follows.

1. Let v' be a virtual node. If v' does not have a parent along a chain, remove v' from the corresponding chain.
2. If v' has a parent along a chain, resolve it according the following rule.

(i) Assume that v' is reached along an edge (u, v'). Assume that v' is labeled with
$v[(w_1, \{(n_{11}, S_{11}), ..., (n_{1j_1}, S_{1j_1})\}), ..., (w_k, u_k, \{(n_{k1}, S_{k1}), ..., (n_{kj_k}, S_{kj_k})\})]$.

(ii) If there exists an n_{ij} such that u is a parent of the node pointed to by n_{ij}, do the following operations:

- Transfer the edges on the alternating path starting at w_i and ending at the $(n_{ij} + 1)$th node w. (An alternating path relative to M_i of $G(V_{i+1}, V_i'; \boldsymbol{C}_i)$ is a path with edges alternatively appearing in $\boldsymbol{C}_i \backslash M_i$ and M_i.)
  Add $w_i \to v$.
- Remove $u \to v'$ and v'.
- Add $u \to w$.

Otherwise, remove v' and connect u to the child node of v' along the chain.

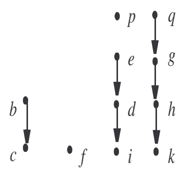


Fig. 11. Illustration for the first problem of Chen's algorithm

We apply this process to the chains shown in Fig. 5(c). Assume that a' is resolved first, we will get another set of chains as shown in Fig. 12(a). It is obtained by transferring edges appearing on the alternating path shown in Fig. 12(b), where a solid edge represents an edge belonging to M_1 while a dashed edge to $\boldsymbol{C}_1 \backslash M_1$.

In a next step, we will further resolve i'. It will transfer the edges on an alternative path shown in Fig. 14(c), which will make a not covered as shown in Fig. 14(d). However, a cannot be connected to d. The final result is a set of six chains as shown in Fig. 14(e). But the graph can be decomposed into a set of five chains as shown in Fig. 14(f). The main reason for this failure is that the alternating path used for resolving a' and the one used for j' share common nodes.

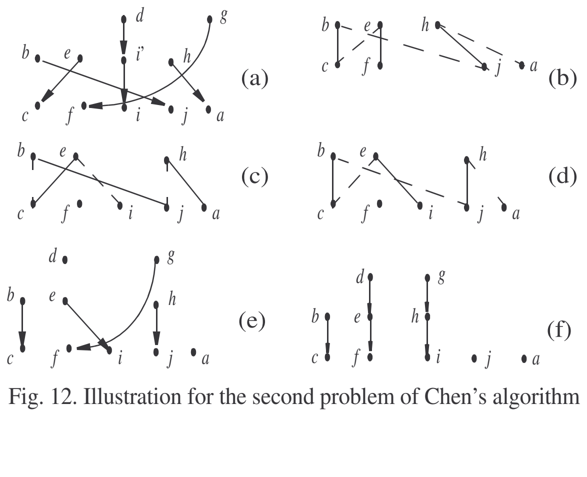


Fig. 12. Illustration for the second problem of Chen's algorithm