An Efficient Algorithm for Answering Graph Reachability Queries

Yangjun Chen, Yibin Chen

Dept. Applied Computer Science, University of Winnipeg

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

y.chen@uwinnipeg.ca

***Abstract* — Given a directed graph *G*, to check whether a node *v* is reachable from another node *u* through a path is often required. In a database system, such an operation is called a *recursion computation* or *reachability checking* and not efficiently supported. The reason for this is that the space to store the whole transitive closure of *G* is prohibitively high. In this paper, we address this issue and propose an $O(n^2 + bn\sqrt{b})$ time algorithm to decompose a directed acyclic graph (*DAG*) into a minimized set of disjoint chains to facilitate reachability checking, where *n* is the number of the nodes and *b* is the DAG's width, defined to be the size of a largest node subset *U* of the DAG such that for every pair of nodes *u, v* ∈ *U*, there does not exist a path from *u* to *v* or from *v* to *u*. Using this algorithm, we are able to label a graph in O(*be*) time and store all the labels in O(*bn*) space with O(log*b*) reachability checking time, where *e* is the number of the edges of the DAG. The method can also be extended to handle cyclic directed graphs. Experiments have been performed, showing that our method is promising.**

I. INTRODUCTION

In numerous applications, including CAD/CAM, CASE, office systems, software management, as well as geographical navigation and ontology queries, data are normally organized into a directed graph (digraph for short) and the ancestor-descendant relationship of nodes (whether a node is reachable from another node through a path) are often enquired.

Let $G(V, E)$ be a directed graph. Digraph $G^* = (V, E^*)$ is the reflexive, transitive closure of G if $(v, u) \in E^*$ iff there is a path from v to u in G. Obviously, if a transitive closure (*TC* for short) is physically stored, the checking of the ancestor-descendant relationship can be done in a constant time. However, the materialization of a whole transitive closure is very space-consuming. Therefore, it is desired to find a way to compress a transitive closure, but without sacrificing too much the query time.

During the past several decades, a lot of research has been done on this issue and materialization of transitive closures in the database research community, including *join index* [5], *hashing* [24], *clustering* of composite objects [21], and *nested relations* (or NF^2 relations, see, e.g., [11]). In addition, *deductive database*s and *object-relational databases* can be considered as two quite different extensions to handle this problem [7, 17].

The so-called *graph labeling* methods discussed in [6, 9, 14, 28] are most related to our work, by which the nodes are assigned labels such that the reachability between nodes can be decided using their labels only. In this sense, a transitive closure is compressed in some way. In the following, we review them in some detail.

- *DAG decomposition*

In [14], Jagadish suggested an interesting method to decompose a DAG (directed acyclic graph) 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. For this method, the space overhead and the query time are respectively $O(kn)$ and $O(\log k)$. However, to find a minimized set of chains for a graph, Jagadish's algorithm needs $O(n^3)$ time (see page 566 in [14]). For this reason, Jagadish suggested a heuristic method to find all the disjoint paths of G and then stitch some paths together to form a chain. In doing so, the number of the produced chains is normally much larger than the minimum number of chains, increasing significantly both space and query time.

- *Tree encoding*

In [6], Chen described a method based on *tree encoding*. It works in two steps. In the first step, a spanning tree G_r of G (called a branching in [6]) is found by exploring G in the depth-first searching fashion. Then, each node v in G_r is associated with a pair (p, q), where p and q are the preorder and postorder numbers with respect to G_r, respectively. Another node u associated with (p', q') is a descendant of v (with respect to G_r) iff $p' > p$ and $q' < q$. In the second step, a pair sequence for each node v is generated by exploring G bottom-up and merging v's pair with the pair sequences of v's child nodes. A pair sequence generated in this way has the following properties:

- Its length is bounded by the number β of the leaf nodes of G_r.
- The pairs in it are increasingly sorted. A pair (p, q) is considered to be smaller that another pair (p', q') if $p < p'$ and $q < q'$.

Therefore, the query time is bounded by $O(\log\beta)$ and the space overhead is $O(\beta n)$. The time on generating such a data structure is bounded by $O(\beta e)$ since for each node v $O(d_v\beta)$ time

is needed to construct the pair sequence for it, where d_v represents the outdegree of v. If we corresponds a leaf node in G_r to the end point of some chain, we can see that β must be equal to or larger than the minimum number of disjoint chains.

- 2-*hop labeling*

The method proposed by Cohen *et al.* [9] labels a graph based on the so-called 2-*hop covers*. 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 on the order of $O(n\sqrt{e})$. The main theoretical barrier of this method is that finding a 2-hop cover of minimum size is an NP-hard problem. So a heuristic method is suggested in [9], 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 $C_{in}(v) \cap C_{out}(v) \neq \phi$. 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 $O(n^4)$.

- *Dual labeling*

Recently, Wang *at el*. proposed a new approach, called *Dual-I*, for sparse graphs [28]. It 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. When $t = e - n$ is on the order of $O(n)$, 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.

There are some other graph labeling methods, such as the method using signatures [26], *PE-Encoding* [8] and *PQ-Encoding* [31]. The idea of the signature-based method [26] 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 $O(l \cdot n)$, 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. The PE-Encoding [8] and the PQ-Encoding [31] are similar to the 2-hop labeling, but with higher computational complexities. The methods discussed in [22, 23] 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 [10] is a geometry-based algorithm to find high-quality 2-hop covers. It has the same theoretical computational complexities as the method discussed in [28] and is only applicable for sparse graphs, too.

In this paper, we propose a new algorithm for general cases. Similar to Jagadish's, we will decompose a DAG into disjoint chains. But we can decompose a graph into a minimized set of disjoint chains in $O(n^2 + bn\sqrt{b})$ time, where b is 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. This enables us to generate a compressed transitive closure in $O(be)$ time, improving the existing methods for the problems of practical size by one order of magnitude or more. The space overhead and the query time are bounded by $O(bn)$ and $\log b$, respectively.

As a by-product, our algorithm can also be used to decompose a finite poset P (partially ordered set) into disjoint chains since any finite poset can be represented as a DAG. According to Dilworth [12], the minimum number of chains is equal to the size of a largest antichain of P (i.e., the width of the corresponding DAG). It is well known that the size of a largest antichain can be determined in $O(e \cdot \sqrt{n})$ time [2]. But it does not mean that the minimized set of chains can be found in the same time (see page 190 in [2]). Up to now, the best approach for this task is based on the network flow algorithm [15, 19] and needs $O(n^3)$ time [15], similar to Jagadish's.

Since our data structure is of the same form as Jagadish's, the maintenance suggested by Jagadish's can be adapted to ours. (So this part of content will not be reported in this paper due to space limitation.)

The remainder of the paper is organized as follows. In Section 2, we show what is the DAG decomposition and how it can be used for the transitive closure compression. In Section 3, we give some basic concepts and techniques related to our algorithm. Section 4 is devoted to the description of our algorithm to decompose a DAG into chains. In Section 5, we report the experiment results. Finally, a short conclusion is set forth in Section 6.

II. TC COMPRESSION BASED ON DAG DECOMPOSITION

Our method is based on the DAG decomposition. For a cyclic graph (a graph containing cycles), we can find all the strongly connected components (*SCC*) in linear time [25] 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 [14]).

Consider the graph shown in Fig. 1(a). Its transitive closure is shown in Fig. 1(b). It is easy to see that we need $O(n^2)$ space to store such an enlarged graph.

This space requirement can be significantly reduced by decomposing a DAG G into a set of disjoint chains that covers all the nodes of G, as illustrated in Fig. 1(c). As we can see, on each chain, if node v appears above node u, there is a path from v to u in G. Based on such a chain decomposition, we can assign to each node an index as follows:

(1)Number each chain and number each node on a chain.

(2)The jth node on the ith chain will be assigned a pair (i, j)

as its index.

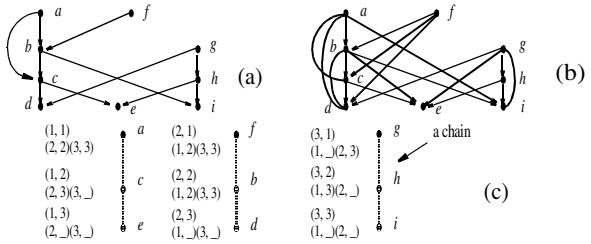

Fig. 1. DAG, transitive closure and graph encoding

In addition, each node v on the ith chain will be associated with an index sequence of length k - 1: $(1, j_1) \ldots (i-1, j_{i-1})$ $(i+1, j_{i+1}) \ldots (k, j_k)$ such that any node with index (x, y) is a descendant of v if $x = i$ and $y < j$ or $x \neq i$ but $y \leq j_x$, where k is the number of the disjoint chains. In this way, the space overhead is decreased to O(kn) (see Fig. 1(c) for illustration). In terms of Dilworth [12], the minimal k equals the width b of G. Once a minimized set of disjoint chains is determined, the index sequences for all nodes can be produced in O(be) time. This can be seen by the following inductive analysis.

First of all, we notice that each leaf node is exactly associated with one index, which is trivially sorted. Let v_1, ..., v_l be the child nodes of v, associated with the index sequences L_1, ..., L_l, respectively. Assume that $|L_i| \leq b$ ($1 \leq i \leq l$) and the indexes in each L_i are sorted according to the first element in each index. We will merge all L_i's into a new index sequence and associate it with v. This can be done as follows. First, make a copy of L_1, denoted L. Then, we merge L_2 into L by scanning both of them from left to right. Let (a_1, b_1) (from L) and (a_2, b_2) (from L_2) be the index pair encountered. We will perform the following checkings:

- If $a_2 > a_1$, we go to the index next to (a_1, b_1) and compare it with (a_2, b_2) in a next step.
- If $a_1 > a_2$, insert (a_2, b_2) just before (a_1, b_1). Go to the index next to (a_2, b_2) and compare it with (a_1, b_1) in a next step.
- If $a_1 = a_2$, we will compare b_1 and b_2. If $b_1 > b_2$, nothing will be done. If $b_2 > b_1$, replace b_1 with b_2. In both cases, we will go to the indexes next to (a_1, b_1) and (a_2, b_2), respectively.

We will repeatedly merge L_2, ..., L_l into L. Obviously, $|L| \leq b$ and the indexes in L are sorted. The time spent on this process is O($d_v b$), where d_v represents the outdegree of v. So the whole cost is bounded by $O(\sum_v d_v b) = O(be)$.

III. BIPARTITE GRAPH AND GRAPH STRATIFICATION

Our method for DAG decomposition is based on a DAG stratification strategy and an algorithm for finding a maximum matching in a bipartite graph. Therefore, the relevant concepts and techniques should be first reviewed and discussed.

A. Stratification of DAGs

Definition 1. (*DAG stratification*) Let $G(V, E)$ be a DAG. The stratification of G is a decomposition of V into subsets $V_1, V_2, ..., V_h$ such that $V = V_1 \cup V_2 \cup ... V_h$ and each node in V_i has its children appearing only in V_{i-1}, ..., V_1 (i = 2, ..., h), 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 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 $C_{i_1}(v) \cup ... \cup C_{i_k}(v)$.

Such a DAG decomposition can be done in O(e) time, by using the following algorithm, in which we use G_1/G_2 to stand for a graph obtained by deleting the edges of G_2 from G_1; and $G_1 \cup G_2$ for a graph obtained by adding the edges of G_1 and G_2 together. In addition, (v, u) represents an edge from v to u; and $d(v)$ represents v's outdegree.

Algorithm *graph-stratification*(G)
begin
1. V_1 := all the nodes with no outgoing edges;
2. **for** i = 1 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. $C_i(v)$:= {links to v_1, ..., v_k};
7. **if** $d(v) > k$ **then** remove v from W;
8. $G := G/\{(v, v_1), ..., (v, v_k)\}$;
9. $d(v) := d(v) - k$;}
10. $V_{i+1} := W$;
11. }
end

In the above algorithm, we first determine V_1, which contains all those nodes having no outgoing edges (see line 1). In the subsequent computation, we determine V_2, ..., V_h. In order to determine V_i ($i > 1$), 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_1 , ..., V_{j+1} are determined. Denote $G_j(V, E_j)$ the reduced graph after the jth iteration of the out-most **for**-loop. Then, any node v in G_j, except those in $V_1 \cup ... \cup V_{j+1}$, does not have children appearing in $V_1 \cup ... \cup V_j$. Denote $d_j(v)$ the outdegree of v in G_j. Thus, in order to check whether v appearing in G_{i-1} has some children not appearing in V_i, we need only to check whether $d_{i-1}(v)$ is strictly larger than k, the number of the child nodes of v appearing in V_i (see line 7).

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

As an example, consider the graph shown in Fig. 1(a). Applying the above algorithm to this graph, we will generate a stratification of the nodes as shown in Fig. 2.

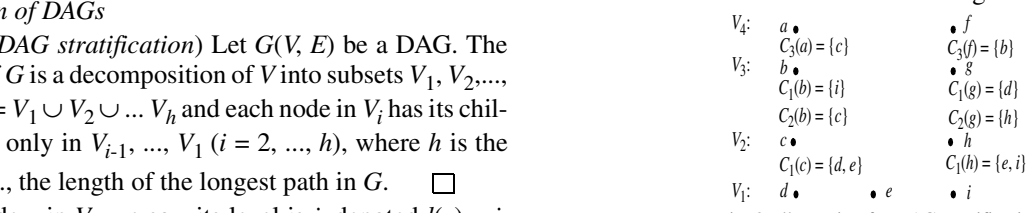

Fig. 2. Illustration for DAG stratification

In Fig. 2, the nodes of the DAG shown in Fig. 1(a) are divided into four levels: $V_1 = \{d, e, i\}$, $V_2 = \{c, h\}$, $V_3 = \{b, g\}$, and $V_4 = \{a, f\}$. Associated with each node at each level is a set of links pointing to its children at different levels.

B. Concepts of Bipartite Graphs

Now we restate two concepts from the graph theory which will be used in the subsequent discussion.

Definition 2. (*bipartite graph* [2]) An undirected graph $G(V, E)$ is bipartite if the node set V can be partitioned into two sets T and S in such a way that no two nodes from the same set are adjacent. We also denote such a graph as $G(T, S; E)$.

For any node $v \in G$, *neighbour*(v) represents a set containing all the nodes connected to v.

Definition 3. (*matching*) Let $G(V, E)$ be a bipartite graph. A subset of edges $E' \subseteq E$ is called a *matching* if no two edges have a common end node. A matching with the largest possible number of edges is called a *maximum matching*, denoted as M_G.

□

Let M be a matching of a bipartite graph $G(T, S; E)$. 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/M and M. A path is an *augmenting path* if it is an alternating path with free origin and terminus. In addition, we will use $free_M(T)$ and $free_M(S)$ to represent all the free nodes in T and S, respectively.

Much research on finding a maximum matching in a bipartite graph has be done. The best algorithm for this task is due to Hopcroft and Karp [13] and runs in $O(e\cdot\sqrt{n})$ time, where $n = |V|$ and $e = |E|$. The algorithm proposed by Alt, Blum, Melhorn and Paul [1] needs $O(n^{1.5}\sqrt{e/(\log n)})$ time. In the case of large e, the latter is better than the former.

IV. ALGORITHM DESCRIPTION

Now we begin to discuss how a DAG can be decomposed into a minimized set of disjoint chains. First, we present our main algorithm in 4.1. Then, in 4.2, we discuss how a kind of redundancy can be removed. Finally, we prove the correctness of the algorithm and analyze its time complexity in 4.3.

A. Main Algorithm

The main idea of the algorithm is to construct a series of bipartite graphs for $G(V, E)$ and then find a maximum matching for each of such bipartite graphs using Hopcroft-Karp algorithm. All these matchings make up a set of disjoint chains and the size of this set is equal to the maximum size of an antichain [12], i.e., the width of G.

During the process, some new nodes, called *virtual nodes*, may be introduced into V_i ($i = 2, ..., h$; $V = V_1 \cup V_2 \cup ... V_h$) to facilitate the computation. However, such virtual nodes will be eventually resolved to get the final result.

In the following, we first show how a virtual node is constructed. Then, the algorithm will be formally described.

We start our discussion with the following specification:

M_i - the found maximum matching of $G(V_{i+1}, V_i; \boldsymbol{C}_i)$, where $\boldsymbol{C}_i = C_i(v_1) \cup ... \cup C_i(v_k)$ with $v_l \in V_{i+1}$ ($l = 1, ..., k$).

M_i' - the found maximum matching of $G(V_{i+1}, V_i'; \boldsymbol{C}_i')$, where $V_i' = V_i \cup \{\text{all the virtual nodes added into } V_i\}$. $\boldsymbol{C}_i' = \boldsymbol{C}_i \cup \{(u, v) \mid u \in V_{i+1}, v \text{ is a virtual node in } V_i'\}$.

In addition, for a graph G, we will use $V(G)$ to represent all its nodes and $E(G)$ all its edges.

Definition 4. (*virtual nodes*) Let $G(V, E)$ be a DAG, divided into $V_1, ..., V_h$ (i.e., $V = V_1 \cup ... \cup V_h$). Let v be a free (actual or virtual) node in $free_{M_i}(V_i')$ (if $i = 1$, we take M_1 as M_1'). Add a virtual node v' into V_{i+1} ($i = 1, ..., h$ -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+2}.
2. If no such a covered node exists, v' is labeled with $v[\,]$. □

In addition, for a virtual node v' (generated for v), we will establish an edge (u, v') 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+1} to v. That is, for each parent w of v, we will establish an edge (w, v') if w appears in V_{i+2}. A virtual edge (v', v) will be constructed to facilitate the virtual node resolution process. Finally, we set V_{i+1}' to be $V_{i+1} \cup \{\text{all those virtual nodes}\}$, and $\boldsymbol{C}_{i+1}'$ to be $\boldsymbol{C}_{i+1} \cup \{(u, v) \mid u \in V_{i+2}, v \text{ is a virtual nodes in } V_{i+1}'\}$.

The following example helps for illustration.

Example 1. Consider the graph shown in Fig. 3(a). The bipartite graph made up of V_2 and V_1, $G(V_2, V_1; \boldsymbol{C}_1)$, is shown in Fig. 3(b) and a possible maximum matching M_1 of it is shown in Fig. 3(c).

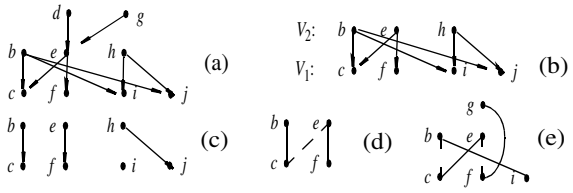

Fig. 3. A bipartite graph and a maximum matching

Relative to M_1, we have a free node i in V_1.

For the free node i, we will construct a virtual node i', labeled with $i[(b, \{(3, \{d, g\})\}), (h, \{(5, \{d, g\})\})]$ for the following reason.

(i) The covered node c and j share the parent b and h with i, respectively.

(ii) On the alternating path starting at b, the 3rd node e has two parents d, and g that appear in V_3. (Fig. 3(d) shows the alternating path starting at b, in which a solid edge represents an edge belonging to M_1 while a dashed edge to $\boldsymbol{C}_1/M_1$.) On the alternating path starting at h, the 5th node e has two parents d, and g that appear in V_3.

The motivation of constructing such a virtual node is that it

is possible to connect f to d or g to form part of a chain if we transfer the edges on the alternating path (starting at b and ending at the node pointed to by 3 + 1 = 4; or starting at h and ending at the node pointed to by 5 + 1 = 6). Then, we 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(e). This can be achieved by the virtual node resolution process (see below).

The bipartite graph made up of V_3 and V_2' is shown in Fig. 4(a). A possible maximum matching M_2' of this bipartite graph is shown in Fig. 4(b).

Now we consider $M_1 \cup M_2$'. It is a set of 4 paths shown in Fig. 4(c). In order to get the final result, all the virtual nodes appearing on those chains have to be resolved. □

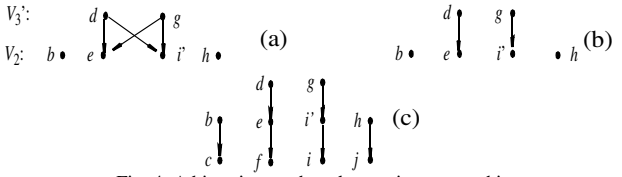

Fig. 4. A bipartite graph and a maximum matching

In the whole process, we may also need to generate virtual nodes for free virtual nodes themselves. However, this can be done in the same way as for actual nodes.

Example 2. Let's have a look at the graph shown in Fig. 1(a) once again. The bipartite graph made up of V_2 and V_1, $G(V_2, V_1; \boldsymbol{C}_1)$, is shown in Fig. 5(a) and a possible maximum matching M_1 of it is shown in Fig. 5(b).

Fig. 5. A bipartite graph and a maximum matching

Relative to M_1, we have a free node e.

For this free node, we will construct a virtual node e', labeled with $e[(c, \{(1, \{b\})\}), (h, \{(1, \{g\})\})]$, as shown in Fig. 6(a). In addition, two edges (b, e') and (g, e') are established according to Definition 4.

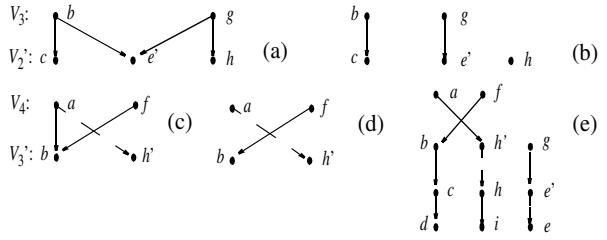

Fig. 6. Illustration for virtual node construction

The graph shown in Fig. 6(a) is the second bipartite graph, $G(V_3, V_2'; \boldsymbol{C}_2')$. Assume that the maximum matching M_2' found for this bipartite graph is a graph shown Fig. 6(b).

Relative to M_2', h is a free node, for which a virtual node h' labeled with $h[(g, \{(1, \{\ \}), (3, \{a\})\})]$ will be constructed as illustrated in Fig. 6(c). This shows the third bipartite graph, $G(V_4, V_3'; \boldsymbol{C}_3')$, which has a unique maximum matching M_3' shown in Fig. 6(d). Consider $M_1 \cup M_2' \cup M_3'$. This is a set of three chains as illustrated in Fig. 6(e). □

From the above discussion, we can see that the algorithm should be a two-phase process. In the first phase, we generate virtual nodes and chains. In the second phase, we resolve all the virtual nodes.

Algorithm *chain-generation*(G's stratification) (*phase 1*)
input: G's stratification.
output: a set of chains
begin
1. find M_1 of $G(V_2, V_1; \boldsymbol{C}_1)$; $M_1' := M_1$; $V_1' := V_1$; $C_1' := C_1$;
2. **for** $i = 2$ **to** h - 1 **do**
3. {construct virtual nodes for V_i according to M_{i-1}';
4. let U be the set of the virtual nodes added into V_i;
5. let W be the newly generated edges incident to the new nodes in V_i;
6. let W' be a subset of W, containing the edges from V_{i+1}; to U;
7. $V_i' := V_i \cup U$; $\boldsymbol{C}_i' := \boldsymbol{C}_i \cup W'$;
8. find a maximum matching M_i' of $G(V_{i+1}, V_i'; \boldsymbol{C}_i')$;
9. }
10. return $M_1 \cup M_2' \cup ... \cup M_{h-1}'$.

end

The algorithm works in two steps: an initial step (line 1) and an iteration step (lines 2 -8). In the initial step, we find a M_1 of $G(V_2, V_1; \boldsymbol{C}_1)$. In the iteration step, we repeatedly generate virtual nodes for V_i and then find a M_i' of $G(V_{i+1}, V_i'; \boldsymbol{C}_i')$. The result is $M_1 \cup M_2' \cup ... \cup M_{h-1}'$.

After the chains for a DAG are generated, we will resolve all the virtual nodes appearing on them.

We distinguish between two kinds of virtual nodes: anchored virtual nodes and unanchored virtual nodes. An anchored virtual node has a parent along the corresponding chain such as the node h' in Fig. 6(e). An unanchored virtual node does not have a parent.

The virtual nodes will be resolved along the chains level by level in a top-down way:

1. If v' is an unanchored node, remove v' from the corresponding chain. If its child along the chain is also a virtual node, then that virtual node becomes unanchored.
2. If v' is an anchored node, 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. Add (w_i, v).
   - Remove (u, v') and v'.
   - Add (u, w).

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

See the following example for a better understanding.

Example 3. Searching the chains shown in Fig. 6(e), we will first meet h' along the edge (a, h'), whose label is $h[(g, \{(1, \{\ \}), (3, \{a\})\})]$. Since a appears in the set indexed with 3, we will (i) transfer the edges on the alternating path starting at g and ending at the 4th node (which is node c) and add edge (g, h), (ii)

remove (a, h') and h', and (iii) add (a, c) (see Fig. 7(a) for illustration).

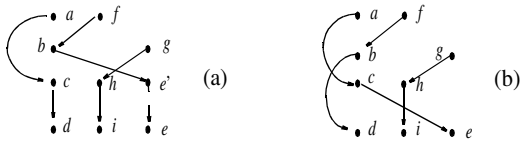

Fig. 7. Illustration for virtual node resolution

Next we will meet e' along the edge (b, e'), whose label is $e[(c, \{(1, \{b\})\}), (h, \{(1, \{g\})\})]$. Since b appears in the set indexed with 1, we will (i) transfer the edges on the alternating path starting at c and ending at the 2nd node (which is node d) and add edge (c, e), (ii) remove (b, e') and e', and (iii) add (b, d). The result is shown in Fig. 7(b). □

The following is a formal description of this process, in which we use U_i to stand for all the chain node on the ith level and represent all the chains as $U_1 \nabla ... \nabla U_h$.

Algorithm *virtual-resolution*(C) (*phase 2*)
input: C - a chain set obtained by executing the algorithm *chain-generation*, represented as $U_1 \nabla ... \nabla U_h$.
output: a set of chains containing no virtual nodes.
begin
1. **for** $i = h$ **downto** 2 **do**
2. { **for** each $v \in U_i$ **do**
3. { **if** v is unanchored virtual node **then** remove it;
4. **if** v is anchored virtual node **then** resolve it according to 2-(i) and (ii) given above;
5. }}
end

B. On the Construction of Virtual Nodes

The construction of virtual nodes dominates the cost. Especially, the label of a virtual node may contain redundant data, which can be easily removed. To have a clear picture, let's have a look at the label associated with i' in Fig. 4(a) once again. It is $i[(b, \{(3, \{d, g\}), (h, \{(5, \{d, g\})\})]$. To generate the first entry (b, $\{(3, \{d, g\})$, we will search an alternating path starting at b shown in Fig. 3(d). To generate the second entry (h, $\{(5, \{d, g\})\})$, we will search an alternating path as shown in Fig. 8, by which the first alternating path is searched for a second time.

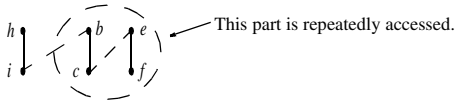

Fig. 8. Illustration for redundancy

To eliminate this kind of redundancy, we do the following:

(i) When we establish a label α for a virtual node, we assign an order number to each entry in α when it is created.
(ii) Each entry in α is augmented with an index. That is, an entry of the form $(w_i, \{(n_{i1}, S_{i1}), ..., (n_{ij_k}, S_{ij_k})\})$ in α will be changed to $(w_i, \{(n_{i1}, S_{i1}), ..., (n_{ij_k}, S_{ij_k})\}, (a_i, b_i))$, where a_i ($< i$) is a number for some entry in α and b_i is a number indicating the position on the corresponding alternating path, which shares the alternating path related to the entry numbered with a_i.

For example, the label $i[(b, \{(3, \{d, g\})\}), (h, \{(5, \{d, g\})\})]$ will be changed to

$i[(b, \{(3, \{d, g\})\}, (\_, \_)), (h, \{(1, 3)\})]$.

In the first entry of this label, the index is (_, _) since when we generate it we find no other alternating path sharing an segment with its alternating path. In the second entry, the index is (1, 3), indicating that part of the alternating path related to this entry (from the 3rd position to the end) is the same as the alternating path related to the entry numbered with 1.

Note that b_i can be a negative integer. To see this, assume that in the above label the entry (h, $\{(5, \{d, g\})\}$) is created before (b, $\{(3, \{d, g\})\}$). Then, the real label should be

$i[(h, \{(5, \{d, g\})\}, (\_, \_)), (b, \{(1, -3)\})]$.

The negative integer -3 in the second entry indicates that the second alternating path starts from the 3rd position on the first alternating path.

In this way, any redundancy can be avoided. In the following, we consider the edge inheritance.

As with the data structure $C_{i_1}(v) \cup ... \cup C_{i_k}(v)$ associated with node v to store its child nodes, we can associate v with another data structure $P_{j_1}(v) \cup ... \cup P_{j_l}(v)$ to store its parents, where $P_{j_r}(v)$ $(1 \le r \le l)$ represents a set of links with each pointing to one of v's parents, which appears in V_{j_r}. Both child and parent links can be organized into linked lists as illustrated in Fig. 9(a).

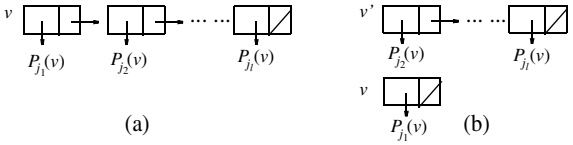

Fig. 9. Illustration for redundancy

When we create a virtual node v' for v (at level j_1 - 1), all the edges incident to v, except the edges from the nodes at level j_1 to v, will be inherited to v'. To do this, we simply graft part of the linked list associated with v to v' as illustrated in Fig. 9(b). Obviously, this operation needs only a constant time.

C. Correctness and Computational Complexities

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

Proposition 1. The number of the chains generated by Algorithm *chain-generation*(G's stratification) is minimum.

Proof. Let $S = \{l_1, ..., l_g\}$ be the set of the chains generated by *chain-generation*(G). For any chain l_i and any two nodes a and b on l_i, if a is above b, there must be a path from a to b. By the virtual node resolution, this property is not changed. Let S' = $\{l_1', ..., l_g'\}$ be the chain set after the virtual node resolution. Then, for any a' and b' on l_i', if a' is above b', we have a path from a' to b'.

Now we show that g is minimum.

First, we notice that the number of the chains produced by the algorithm *chain-generation* is equal to

$$N_h = |V_1| + |free_{M_1}(V_2)| + |free_{M_2'}(V_3)| + ... + |free_{M_{(h-1)}'}(V_h)| .$$

We will prove by induction on h that N_h is minimum.

Initial step. When $h = 1, 2$, the proof is trivial.

Induction step. Assume that for any DAG of height k, N_k is minimum. Now we consider the case when $h = k + 1$:

$$N_{k+1} = |V_1| + |free_{M_1}(V_2)| + |free_{M_2'}(V_3)| + ... + |free_{M_k'}(V_{k+1})| .$$

If $|free_{M_1}(V_1)| = 0$, no virtual node will be added into V_2. Therefore, $V_2 = V_2'$. In this case,

$$N_{k+1} = |V_2| + |free_{M_2}(V_3)| + |free_{M_3'}(V_4)| + ... + |free_{M_k'}(V_{k+1})| .$$

In terms of the induction hypothesis, it is minimum.

If $|free_{M_1}(V_1)| > 0$, we have $|V_1| > |V_2|$. In this case, we consider another graph G' constructed from $G(V, E)$ as follows:

1. Divide $|free_{M_1}(V_1)|$ into two groups: g_1 and g_2. In g_1, each node has at least one parent in V_2. In g_2, each node has no parent in V_2.
2. Let g_1' and g_2' be the virtual nodes generated for g_1 and g_2 with the newly created edges E_1 and E_2, respectively. Construct $G'(V', E')$ such that $V' = (V/V_1) \cup g_1' \cup g_2'$ and $E' = (E/\{(u, v) \mid u \in V_2, v \in V_1\}) \cup E_1 \cup E_2$.

We show that each decomposition of G' corresponds to a decomposition of G and they have the same size. Let $U_1 \nabla ... \nabla U_k$ be a decomposition of G'. We note that $U_1 = V_2 \cup g_1' \cup g_2'$. If any node in g_1' does not have a parent in the decomposition, we connect a node u in V_2 to a node v in V_1 if $(u, v) \in M_1$. Then, $(V_1 \cup g_1' \cup g_2') \nabla V_2 \nabla U_2 ... \nabla U_k$ is a decomposition of G with the same size as that of G'. Otherwise, let u_j be the parent of $v_j \in g_1$ ($j = 1, ..., l$ for some l). We change the edges in M_1 by resolving each v_j. In this way, we will have a decomposition of G with the same size as that of G'.

In a similar way, we can also show that each decomposition of G corresponds to a decomposition of G' and they have the same size.

G' is of height k. For G', the number of the chains produced by the algorithm *chain-generation* is equal to

$$N_k' = |V_2'| + |free_{M_2'}(V_3)| + |free_{M_3'}(V_4)| + ... + |free_{M_k'}(V_{k+1})| .$$

Let $V_2' = W_1$, $V_3 = W_2$, ..., $V_{k+1} = W_k$. We have

$$N_k' = |W_1| + |free_{L_1}(W_2)| + |free_{L_2'}(W_3)| + ... + |free_{L_{(k-1)}'}(W_k)| ,$$

where $L_1 = M_2'$ and $L_i' = M_{(i+1)}'$ ($i = 2, ..., k - 1$).

In terms of the induction hypothesis, N_k' is minimum. So $N_{k+1} = N_k'$ is minimum. This completes the proof. □

In the following, we analyze the computational complexities of the algorithm. The cost of the whole process can be divided into four parts:

- $cost_1$: the time spent on establishing virtual nodes and the corresponding new edges.
- $cost_2$: the time for edge inheritance.
- $cost_3$: the time for finding a maximum matching for every $G(V_{i+1}, V_i'; \boldsymbol{C}_i)$.
- $cost_4$: the time for resolving virtual nodes.

We claim that $cost_1$ is bounded by $O(n^2)$ since for each actual node v at most h virtual nodes will be constructed and the number of the new edges incident to a virtual node added to V_i is bounded by $|V_{i+1}|$. So the number of the new edges incident to these virtual nodes (related to v) is on the order of

$$O(\sum_{i=2}^{h-1} |V_i|) = O(n).$$

$cost_2$ is the time for edge inheritance, which is bounded by $O(\sum h) = O(nh)$.

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

$$O(\sqrt{|V_{i+1}| + |V_i|} \cdot |\boldsymbol{C}_i|). \qquad \text{(see [13])}$$

Therefore, $cost_3$ is bounded by

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

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

During the virtual-resolution process, the virtual nodes are resolved level by level. At each level, only $O(|\boldsymbol{C}_i'|)$ edges are visited. Therefore, $cost_4$ is bounded by

$$O(\sum_{i=1}^{h-1} |\boldsymbol{C}_i'|) = O(bn). \qquad \square$$

From the above analysis, we get the following proposition.

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

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 $G(V_{i+1}, V_i'; \boldsymbol{C}_i')$ is bounded by $O(b|V_{i+1}|)$. (After the edges incident to a virtual node v' are inherited to v'', the virtual node of v', they are not incident to v' any more.)

V. EXPERIMENTS

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 written in C++, running standalone.

A. On the Tested Methods

In the experiments, we have tested six methods:

- DAG decomposition - Jagadish's heuristic (*DD* for short) [14],
- Tree encoding by Chen (*TE* for short) [6],
- 2-hop labeling by Cohn et al. (*2-hop* for short) [9]
- Dual labeling by Wang et al. (*Dual-II* for short) [28],
- Matrix multiplication by Warren (*MM* for short) [27],
- ours (discussed in this paper).

The theoretical computational complexities of these meth-

ods, as well as the graph traversal are shown below.

	query time	labeling time	space overhead
graph-traversal	$O(e)$	0	0
DAG-decomposition	$O(\log b)$	$O(n^3)$	$O(bn)$
tree-labeling	$O(\log\beta)$	$O(\beta e)$	$O(\beta n)$
dual-II	$O(\log t)$	$O(n + m + t^3)$	$O(n + t^2)$
2-hop	$O(e^{1/2})$	$O(n^4)$	$O(ne^{1/2}\log n)$
matrix-multiplication	$O(1)$	$O(n^3)$	$O(n^2)$
ours	$O(\log b)$	$O(be)$	$O(bn)$

In this experiment, we use Jagadish's heuristic algorithm for tests since it needs much less time than $O(n^3)$, but with a commensurate sacrifice in space overhead. In addition, we implemented Dual-II, instead of Dual-I for tests. It is because for non-sparse graphs, Dual-I needs even more space than any traditional matrix-based method; no compression in any sense.

B. Test Results

The tests are organized into three groups. In the first group, we test large but sparse DAGs. In the second group, we test large and non-sparse DAGs. In the third group, we test very dense DAGs, but with relatively small number of nodes. In these tests, we measured the space overhead, the time spent on the generation of compressed transitive closures, as well as the time on checking reachability.

1) Tests on Sparse Graphs: In this group of experiments, we tested a series of graphs with 15000 nodes. The edges are randomly generated, ranging from 16000 edges to 20000 edges. For each generated graph, Tarjan's algorithm is used to find SCCs as a preprocessor. All SCCs are then removed.

In Table 1, we show the average size of the data structures generated by the different methods, and the average times spent on generating such data structures.

Table 1: Size of sparse graphs' TC and time for generating it

	size of data structures (16 bits)	time for generating TC (sec.)
ours	39126	15.764
DD	170786	67.683
TE	30357	12.025
Dual-II	36389	42.227
2-hop	801217	24145
MM	14063750	675.812

From this table, we can see that Chen's tree encoding method has the best performance both in space overhead and time for generating compressed transitive closures. It is because for this kind of graphs, the pair sequences associated with the nodes are quite short. Dual-II also has very good performance since the *TLC* search trees are very small, which are proportional to the number of non-tree edges. Our method is much better than Jagadish's heuristic method since the number of chains generated by Jagadish's is significant larger than the minimum number of chains. 2-hop can somehow reduce the size of the transitive closure. But it took too much time (more than 6 hours) for the task.

Fig. 10 shows the average query time over the tested graphs. Each query is a pair (x, y) to check whether node x is an ancestor of node y. For each graph, we have checked up to 100,000 queries randomly generated and recorded the accumulated time.

From this figure, we can see that Warren's method is best. (In our implementation, a boolean matrix is simply stored as bit strings.) The tree encoding method is slightly better than Dual-II since each time to check reachability the *TLC* search tree may be explored by Dual-II. But by the tree encoding method, a quite short pair sequence is visited in a binary searching way. Again, our method is better than Jagadish's heuristic method since the index sequences by ours (which is exactly the minimum number of disjoint chains) are averagely shorter than those generated by Jagadish's.

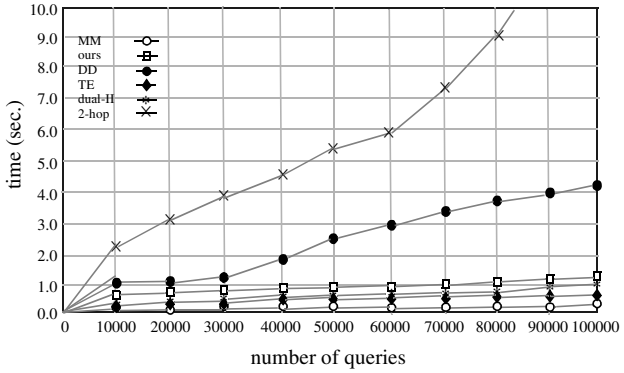

Fig. 10. Time for query evaluation - Group I

2) Test on Non-sparse Graphs: In the second group of experiments, we mainly tested two types of DAGs:

(1) *DAG systematically generated* (DSG)
A DAG of 640 roots with about four children per non-leaf; about three parents per non-root, eight levels, 31525 nodes and 71786 edges.

(2) *DAG semi-randomly generated* (DSRG)
Any graph of this type is generated as follows:
(i) construct a tree with each node having a random number of children from zero to six;
(ii) the tree contains a minimum of 20000 nodes; and
(iii) add randomly up to 10000 edges to the tree while ensuring that no cycle is formed.

The graph parameters are summarized in Table 2.

Table 2: Graph parameters for Group II

	number of nodes	number of arcs	average out-degree of internal nodes	average path length
DSG	31525	71786	3	8.0
DSRG	20004	30003	2.3	10.11

In Table 3, 2-hop is not included since it took too long to generate labels. We only report the results of the other five methods.

Table 3: Size of DSG's TC and time for generating it

	size of data structures (16 bits)	time for generating TC (sec.)
ours	169853	21.572
DD	307460	182.261
TE	267831	53.253
Dual-II	77182041	1269.359
MM	62114102	789.703

From this table, it can be seen that our method uniformly outperforms all the other methods. Especially, our method is better than Chen's tree encoding, which shows that the number of the leaf nodes of the found spanning tree can be much larger than the graph's width although a theoretical explanation can not be delivered. However, Chen's tree encoding is much better than Jagadish's heuristic method. On the one hand, the number of the chains found by Jagadish's is still large. On the other hand, find-

ing paths which can be connected to form a chain is very costly. Dual-II even needs more space and more time than Warren's. This shows that this method is totally not suitable for non-sparse graphs since the space complexity $O((e - n)^2)$ and the time complexity $O((e - n)^3)$ of this method become respectively $O(n^2)$ and $O(n^3)$ or more when a graph is not sparse. Although both Dual-II and Warren's are of the same theoretical space and time complexities, the boolean operations by Warren's make it more efficient than Dual-II.

In Fig. 11, we show the time spent on the query evaluation.

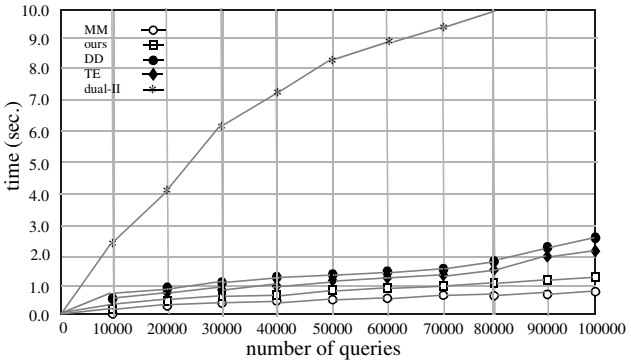

Fig. 11. Time for query evaluation - Group II: DSG

For the DSG, the query time of our method is not much worse than Warren's and better than all the other three methods. The reason for this is that the index sequence associated with a node by our method is shorter than both Jagadish's and Chen's. The query time of Dual-II is the worst among all the approaches. In fact, the claim that the query time is bounded by log$t$ [28] may not be true since there is no guarantee that a *TLC* search tree is balanced.

Table 4 shows the sizes of the data structures generated by the different methods for storing the compressed transitive closure of DSRG, and the times spent on generating such data structures.

Table 4: Size of DSRG's TC and time for generating it

	size of data structures (16 bits)	time for generating TC (sec.)
ours	68167	7.813
DD	356310	100.989
TE	200278	27.432
Dual-II	31613640	591.015
MM	25010001	286.235

Form this table, it can be observed that the time used by our method to generate a data structure for the DSRG's transitive closure is again much less than all the other graph labeling strategies, as well as Warren's. More importantly, the discrepancy of the space overhead between ours and all the other strategies is huge. It is just a little larger than the original graph while Jagadish's needs more than 8 times of space, Chen's about 5 times, and Warren's about 800 times. Dual-II even needs more space and time than Warren's.

We show the time for the query evaluation in Fig. 12. This figure demonstrates that our method needs slightly more time than Warren's for checking reachability, but better than all the other graph labeling approaches. Together with Table 4, this shows that trading time for space by our method pays off.

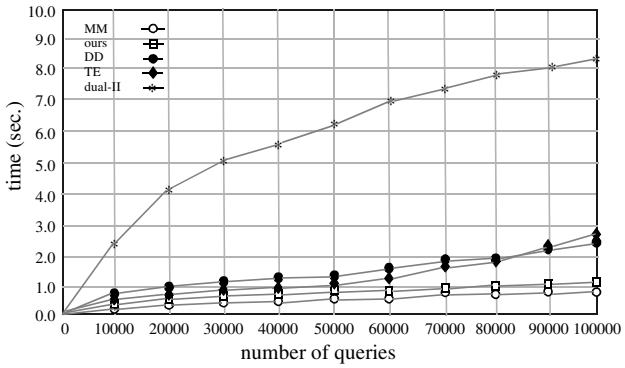

Fig. 12. Time for query evaluation - Group II: DSRG

3) Tests on Dense Graphs: In the third group of experiments, we have tested some DAGs with density near 0.25 (referred to as 0.25-DAG)

Any graph of this type contains 3000 nodes connected by 2230196 edges generated randomly. The density of the graph is

$$\frac{|E|}{|V|^2} = 2230196/9000000 = 0.247.$$

In Table 5, we show the sizes of the data structures generated by the different methods for storing the transitive closure of a 0.25-DAG, and the times spent on generating such data structures.

Table 5: Size of 0.25-DAG's TC and time for generating it

	size of data structures (16 bits)	time for generating TC (sec.)
ours	96000	23.000
DD	444420	235.354
TE	209784	101.000
Dual-II	1402622	2554.218
MM	562500	141.99

As we can see, even for very dense graphs our method works well and effectively compacts the transitive closures. The time for generating data structures is also very low. In fact, a dense graph tends to have a smaller width. This may explain why our method has an advantage over the others. We also notice that the space overhead of the tree-encoding method is not much worse than ours. The reason for this is that the more dense a graph is, the more reachability is "covered" by the spanning tree of that graph.

Fig. 13 shows the query time. Again, our method works much more efficiently than all the other graph labeling approaches although it is a little bit inferior to Warren's. For a dense graph, the average size of an index sequence by ours is smaller than the number of the leaf nodes of the spanning tree, as well as the number of chains found by Jagadish's heuristic method, and much smaller than the numbers of the nodes and edges of the graph.

VI. CONCLUSION

In this paper, a new algorithm for finding a chain decomposition of a DAG is proposed, which is useful for compressing transitive closures. The algorithm needs only $O(n^2 + bn\sqrt{b})$ time and $O(e + bn)$ space, where n and e are the number of the nodes and the edges of the DAG, respectively; and b is the DAG's width. The main idea of the algorithm is a DAG stratification that divides the DAG into a series of bipartite graphs.

Then, by using Hopcroft-Karp's algorithm for finding a maximum matching for each bipartite graph, a set of disjoint chains with virtual nodes involved can be produced in an efficient way. Finally, by resolving the virtual nodes on the chains, we will get the final result. Based on this algorithm, we can generate a compressed transitive closure in O(*be*) time and store it in O(*be*) space. The query time is bounded by O(log*b*). A wide range of graphs is tested, including sparse graphs, non-sparse graphs, and very dense graphs. This shows that our method significantly outperforms the existing graph labeling methods.

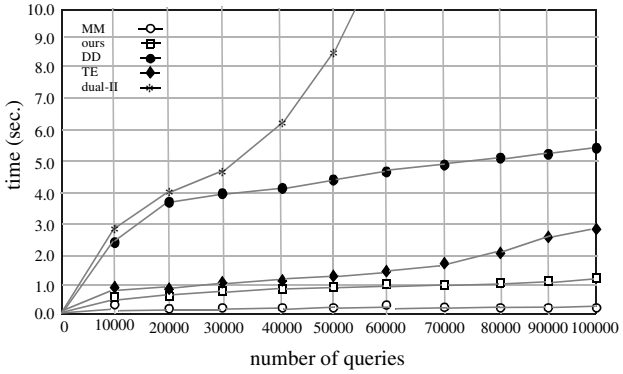

Fig. 13. Time for query evaluation - Group III

REFERENCES

[1] H. Alt, N. Blum, K. Mehlhorn, and M. Paul, Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{e/(\log n)})$, Information Processing Letters, 37(1991), 237 -240.

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

[3] J. Banerjee, W. Kim, S. Kim and J.F. Garza, "Clustering a DAG for CAD Databases," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 14, No. 11, Nov. 1988, pp. 1684-1699.

[4] 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.

[5] M. Carey et al., "An Incremental Join Attachment for Starburst," in: *Proc. 16th VLDB Conf.,* Brisbane, Australia, 1990, pp. 662-673.

[6] 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.

[7] Y. Chen, "On the Graph Traversal and Linear Binary-chain Programs," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15, No. 3, May 2003, pp. 573-596.

[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] 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.

[11] P. Dadam et al., "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies," *Proc. ACM SIGMOD Conf*., Washington D.C., 1986, pp. 356-367.

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

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

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

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

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

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

[18] 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.

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

[20] K. Mehlhorn, *Graph Algorithms and NP-Completeness*: *Data Structure and Algorithm 2*, Springer-Verlag, Berlin, 1984.

[21] M. Stonebraker, L. Rowe and M. Hirohama, "The Implementation of POSTGRES," *IEEE Trans. Knowledge and Data Eng*., vol. 2, no. 1, 1990, pp. 125-142.

[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] L.D. Shapiro, "Join Processing in Database Systems with Large Main Memories," *ACM Trans. Database Systems*, vol. 11, no. 3, 1986, pp. 239-264.

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

[26] 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.

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

[28] 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.

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

[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.