Decomposing DAGs into Disjoint Chains

Conference Paper · September 2007

DOI: 10.1007/978-3-540-74469-6_25 · Source: DBLP

1 author:

Yangjun Chen
The University of Winnipeg
142 PUBLICATIONS **488** CITATIONS

Decomposing DAGs into Disjoint Chains

Yangjun Chen

Department of Applied Computer Science
University of Winnipeg
Winnipeg, Manitoba, Canada R3B 2E9
y.chen2@uwinnipeg.ca

Abstract. In this paper, we propose an efficient algorithm to decompose a directed acyclic graph (DAG) into chains, which has a lot of applications in computer science and engineering. Especially, it can be used to store transitive closures of directed graphs in an economical way. For a DAG G with n nodes, our algorithm needs $O(n^2 + bn\sqrt{b})$ time to find a minimized set of disjoint chains, where b is G's width, defined to be the 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. Accordingly, the transitive closure of G can be stored in $O(bn)$ space, and the reachability can be checked in $O(\log b)$ time. The method can also be extended to handle cyclic directed graphs.

1 Introduction

Let G be a DAG. A chain cover of G is a set S of disjoint chains such that it covers all the nodes of G, and for any two nodes a and b on a chain $p \in S$, if a is above b then there is a path from a to b in G. In this paper, we discuss an efficient algorithm to find a minimized S for G.

As an application of this problem, consider the transitive closure compression. Let $G(V, E)$ be a directed graph (digraph for short). 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. (See Fig. 1(a) and (b) for illustration.) Obviously, if a transitive closure (*TC* for short) is physically stored, the checking of the ancestor-descendant relationship (whether a node is reachable from another node through a path) 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.

The idea of using disjoint chains to compress a transitive closure was first suggested by Jagadish [6], who proposed an $O(n^3)$ algorithm to decompose a DAG into a minimized set of disjoint chains by reducing the problem to a network flow problem.

Once G is decomposed, its transitive closure can be represented as follow.

(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.

R. Wagner, N. Revell, and G. Pernul (Eds.): DEXA 2007, LNCS 4653, pp. 243–253, 2007.

In addition, each node v on the ith chain will be associated with a pair sequence of length k - 1: $(1, j_1)$ …$(i - 1, j_{i-1})$ $(i + 1, j_{i+1})$ … (k, j_k) such that for any index (x, y) $(x \neq i)$ if $y \leq j_x$ it is a descendant of v, where k is the number of the disjoint chains. Of course, any node below v on the ith chain is a descendant of v. In this way, the space overhead is decreased to $O(kn)$ (see Fig. 1(c) for illustration).

More importantly, it is very convenient to handle such a data structure in relational databases. We need only to establish a relational schema of the following form:

Node(Node_id, *label*, *label_sequence*, ...),

where *label* and *label_sequence* are utilized to accommodate the label pair and the label pair sequence associated with the nodes of G, respectively. Then, to retrieve the descendants of node v, we issue two queries as below.

Q_1:	SELECT	label	Q_2:	SELECT	*
	FROM	Node		FROM	Node
	WHERE	Node_id = v		WHERE	ϕ(label, label_sequence, y).

Let the label pair obtained by evaluating the first query Q_1 be y. Then, by evaluating the function ϕ(label, label_sequence, y) in the second query Q_2, we will get all those nodes, whose labels are subsumed by y or whose label sequences contain a label subsumed by y. A label pair (x, y) is said to be subsumed by another pair (x', y') if $x = x'$ and $y \leq y'$.

Since each label sequence is sorted according to the first element of each pair in the sequence, we need only $O(\log_2 k)$ time to check whether a node u is a descendant of v.

As demonstrated in [13], k is equal to b, the width of G, defined to be the 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.

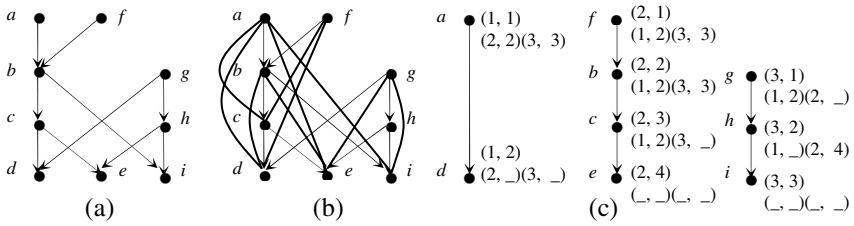

Fig. 1. DAG, transitive closure and graph encoding

Finally, we note that this technique can also be employed for cyclic graphs since we can collapse each *strongly connected component* (SCC) to a single node while maintaining all the reachability information (see 3.3 of [6]).

There are some other efforts to compress transitive closures. The method discussed in [4] is based on the so-called tree encoding. This method requires $O(\beta e)$ time to generate a compressed transitive closure, where β is the number of the leaf nodes of a spanning tree covering all the nodes of G. β is generally much larger than b. The time

and space complexities of this method are bounded by $O(\beta n)$ and $O(\log\beta)$, respectively. In [12], a quite different method is proposed, using the so-called *PQ*-Encoding, by which each node is associated with an interval and a bit-vector that is of the length equal to the number of slices. A slice is just a subset of the nodes and the size of a slice is bounded by a constant. So the number of the slices must be on $O(n)$. Therefore, this method needs $O(n^2)$ space in the worst case, not suitable for a database environment.

In this paper, we propose a new algorithm for decomposing DAGs into a minimized set of disjoint chains. Its time complexity is bounded by $O(n^2 + bn\sqrt{b})$, which enables us to generate a compressed transitive closure efficiently.

The remainder of the paper is organized as follows. In Section 2, we give some basic concepts and techniques related to our algorithm. Section 3 is devoted to the description of our algorithm to decompose a DAG into chains. In Section 4, we prove the correctness of the algorithm and analyze its computational complexities. Finally, a short conclusion is set forth in Section 5.

2 Graph Stratification and Bipartite Graphs

Our method for finding a minimized set of chains 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.

2.1 Stratification of DAGs

Let $G(V, E)$ be a DAG. We decompose V into subsets $V_1, V_2, ..., V_h$ such that $V = V_1 \cup V_2 \cup ... \cup V_h$ 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. $C_i(v) := \{v_1, ..., v_k\}$;
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 changed 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 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) once again. Applying the above algorithm to this graph, we will generate a stratification of the nodes as shown in Fig. 2.

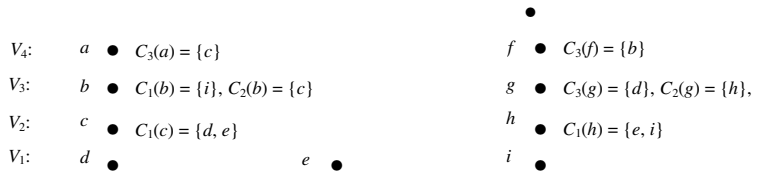

Fig. 2. Illustration for DAG stratification

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 below.

2.2 Concepts of Bipartite Graphs

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

Definition 1. (*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 2. (*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 with 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 conducted. The best algorithm for this task is due to Hopcroft and Karp [5] 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.

3 Algorithm Description

The main idea of our algorithm is to construct a series of bipartite graphs for $G(V, E)$ and then find a maximum matching for each bipartite graph by using Hopcroft-Karp's algorithm. All these matchings make up a set of disjoint chains and the size of this set is minimal.

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 obtain the final result.

In the following, we first show how a virtual node is constructed. Then, the algorithm will be 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 $\boldsymbol{C}_i = C_i(v_1) \cup ... C_i(v_k)$ and $v_l \in V_{i+1}$ ($l = 1, ..., k$).

M_i' - the found maximum matching of $G(V_{i+1}, V_i'; \boldsymbol{C}_i')$, where $V_i' = V_i \cup$ {all the virtual nodes added into V_i} $\cup$ {all the free nodes in $C_{i-1}(v) \cup ... C_1(v)$ for $v \in V_{i+1}$}. $\boldsymbol{C}_i' = \boldsymbol{C}_i \cup$ {all the new edges incident to the virtual nodes in V_i'} $\cup$ {all the edges incident to the free nodes in $C_{i-1}(v) \cup ... C_1(v)$ for $v \in V_{i+1}$}.

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

Definition 3. (*virtual nodes*) Let $G(V, E)$ be a DAG, divided into $V_1, ..., V_h$ (*i.e.*, $V = V_1 \cup ... \cup V_h$). The virtual nodes added into V_i ($i = 1, ..., h - 1$) are the new nodes constructed as below.

1. If $i = 1$, no virtual nodes are created for V_1. $V_1' := V_1$; $\boldsymbol{C}_1' := \boldsymbol{C}_1$; $M_1' := M_1$.
2. If $i > 1$, let M_{i-1}' be a maximum matching of $G(V_i, V_{i-1}'; \boldsymbol{C}_{i-1}')$. Let v be a free node in $free_{M_{i-1}'}(V_{i-1}')$. Let $u_1, ..., u_k$ be the covered nodes appearing in V_{i-1}' such that each

u_g (g = 1, ..., k) shares a covered parent node w_g (i.e., (w_g, u_g) ∈ $M_{i\text{-}1}$') with v. Let v_{g1}, ..., v_{gj_g} be all the parents of u_g in V_i. Construct a virtual node v' (to be added into V_i), labeled with v[(w_1, u_1, {v_{11}, ..., v_{1j_1}}), ..., (w_k, u_k, {v_{k1}, ..., v_{kj_k}})]. Let o_1, ..., o_l be the nodes in V_{i+1}, which have at least one child node appearing in {v_{11}, ..., v_{1j_1}} ∪ ... ∪ {v_{k1}, ..., v_{kj_k}}. Establish the edges (o_1, v'), ..., (o_l, v'). Establish a virtual edge from v' to v (so v is not considered as a free node any more). Denote V_i' = V_i ∪ {all the virtual nodes added into V_i} ∪ {all the free nodes in $C_{i\text{-}1}(v)$ ∪ ... $C_1(v)$ for v ∈ V_{i+1}}. Denote $\boldsymbol{C}_i$' = $\boldsymbol{C}_i$ ∪ {all the new edges} ∪ {all the edges incident to the free nodes in $C_{i\text{-}1}(v)$ ∪ ... $C_1(v)$ for v ∈ V_{i+1}}.

The following example helps for illustration.

Example 1. 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. 3(a) and a possible maximum matching M_1 of it is shown in Fig. 3(b).

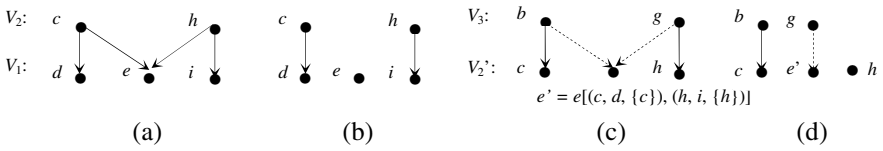

Fig. 3. Bipartite graphs and maximum matchings

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, d, {c}), (h, i, {h})], as shown in Fig. 3(a). In addition, two edges (b, e') and (g, e') are established according to Definition 3.'

The graph shown in Fig. 3(c) 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. 3(d).

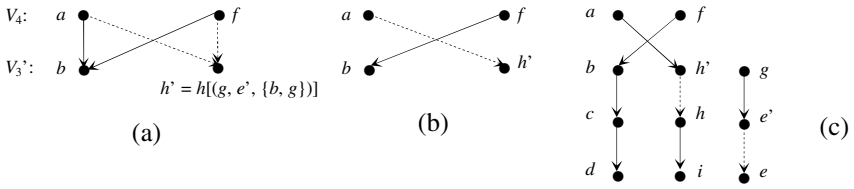

Fig. 4. Illustration for virtual node construction

Relative to M_2', h is a free node, for which a virtual node h' = h[(g, e', {b, g] will be constructed as illustrated in Fig. 4(a). This shows the third bipartite graph, $G(V_4, V_3'; \boldsymbol{C}_3')$, which has a unique bipartite graph M_3' shown in Fig. 4(b).

Now we consider $M_1 \cup M_2' \cup M_3'$. This is a set of three chains as illustrated in Fig. 4(c). In order to get the final result, all the virtual nodes appearing in those chains have to be resolved.

Therefore, we will have 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 $G(V_2, V_1; \boldsymbol{C}_1)$; $M_1' := M_1$; $V_1' := V_1$; $\boldsymbol{C}_1' := \boldsymbol{C}_1$; $i := 2$;
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 virtual nodes in V_i;
6. $V_i' := V_i \cup U$; $\boldsymbol{C}_i' := \boldsymbol{C}_i \cup W$;
7. find a maximum matching M_i' of $G(V_{i+1}, V_i'; \boldsymbol{C}_i')$;
8. }

end

The algorithm works in two steps: an initial step (line 1) and an iteration step (lines 2 - 8). In the initial step, we determine 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 involved in those chains. To do this, we search each chain top-down. Whenever we meet a virtual node v' along an edge (u, v'), we do the following:

1. Let $v[(w_1, u_1, \{v_{11}, ..., v_{1j_1}\}), ..., (w_k, u_k, \{v_{k1}, ..., v_{kj_k}\})]$ be the label of v'.
2. If there exists an i such that u is an ancestor of some node in $\{v_{i1}, ..., v_{ij_i}\}$, do the following operations:

   (i) Replace (w_i, u_i) with (w_i, v).
   (ii) Remove (u, v') and v'.
   (iii) Add (u, u_i).

See the following example for a better understanding.

Example 2. Searching the chains shown in Fig. 4(c), we will first meet h', whose label is $h[(g, e', \{b, g\})]$. Since b is a descendant of a, we will (i) replace (g, e') with (g, h), (ii) remove (a, h') and h', and (iii) add (a, e') (see Fig. 5(a) for illustration).

Next we will meet e', whose label is $e[(c, d, \{c\}), (h, i, \{h\})]$. Since c is a descendant of a, we will (i) replace (c, d) with (c, e), (ii) remove (a, e') and e', and (iii) add (a, d). The result is shown in Fig. 5(b).

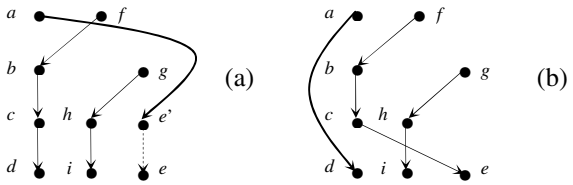

Fig. 5. Illustration for virtual node resolution

By this example, we should pay attention to the following properties:

(1) When we resolve $h' = h[(g, e', \{b, g\})]$, we will check whether b or g is a descendant of a. For this purpose, we need only to check a's child nodes since (a, h') is an edge on the chain.
(2) After h' is resolved, $e' = e[(c, d, \{c\}), (h, i, \{h\})]$ becomes connected to a and will be resolved next. Similarly, we will check whether c or d is a descendant of a. But we only need to check whether they are b's child nodes instead of searching G from a again. It is because e' appears in $h[(g, e', \{b, g\})]$ (showing that e' is a virtual child node of b) and b is the only child node of a.

In this way, we search the graph G only once for resolving the virtual nodes along a chain.

The following is the formal description of this process.

Algorithm. *virtual-resolution*(C) (*Phase 2*)
input: C - a chain set obtained by executing the algorithm *chain-generation*.
output: a set of chains containing no virtual nodes
begin
1. $C' := \Phi$;
2. **while** C not empty **do**
3. {choose a chain l from C such that the first virtual node on l appears at the highest level (the tie is resolved arbitrarily);
4. let l' be the chain containing no virtual node after resolving virtual nodes along l;
5. $C' := C' \cup \{l'\}$; $C := C/\{l'\}$;
6. }

end

4 Correctness and Computational Complexities

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

Proposition 1. The size of the chains generated by Algorithm *chain-generation*( ) is minimum.

Proof. Let $S = \{l_1, ..., l_g\}$ be the set of the chains generated by *chain-generation*( ). For any chain l_i and any two nodes v and u on l_i, if v is above u, there must be a path from v to u. 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 v' and u' on l_i', if v' is above u', we have a path from v' to u'.

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 $|free_{M_1}(V_1)| > 0$, some virtual nodes are added into V_2 and the corresponding edges are added into $\boldsymbol{C}_2$. Removing V_1, we get the stratification of another graph G' (with all the leaf nodes being in V_2'), which is of height k and has the same minimal decomposition as G. 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 $L_i' = M_{(i+1)}'$ ($i = 2, \ldots, k - 1$).

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

According to [6], the number of the chains in a minimized set is equal to b.

In the following, we analyze the computational complexities of the algorithm.

Lemma 1. The time complexity of the algorithm *chain-generation* is bounded by $O(n^2 + bn\sqrt{b})$.

Proof. The cost for generating a virtual node v' for node v can be divided into two parts: $cost_1$ and $cost_2$. $cost_1$ is the time spent on establishing new edges, which is bounded by $O(n^2)$ since for each actual node at most h virtual nodes will be constructed and the number of the new edges incident with all these virtual nodes is bounded by $O(n)$.

$cost_2$ is the time for the edge inheritance from node v. It is bounded by a constant.

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

$$O(\sqrt{|V_{i+1}| + |V_i'|} \cdot |\boldsymbol{C}_i'|). \quad \text{(see [5])}$$

Therefore, the total cost of this process is

$$O(n^2) + O(\sum_{i=1}^{h-1} \sqrt{|V_{i+1}| + |V_i'|} \cdot |\boldsymbol{C}_i'|) \le O(n^2 + (\sqrt{b} \sum_{i=1}^{h-1} b \cdot |V_i) = O(n^2 + bn\sqrt{b}).$$

Lemma 2. The time complexity of the algorithm *virtual-resolution* is bounded by $O(n^2)$.

Proof. During the process, the virtual nodes will be resolved level by level. At each level, only $O(|\boldsymbol{C}_i'|)$ edges will be visited. Therefore, the total number of the visited edges is bounded by

$$O(\sum_{i=1}^{h-1} |\boldsymbol{C}_i'|)) = O(n^2).$$

From Lemma 1 and Lemma 2, we have 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 process is bounded by $O(e + bn)$ since during the execution of the algorithm *chain-generation* at most bn new edges are added.

5 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 $O(n^2 + bn\sqrt{b})$ 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 generates a series of bipartite graphs. Then, by using Hopcropt-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 in the chains, we will get the final result.

References

[1] Alt, H., Blum, N., Mehlhorn, K., Paul, M.: Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{e/(\log n)})$

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

[3] Banerjee, J., Kim, W., Kim, S., Garza, J.F.: Clustering a DAG for CAD Databases. IEEE Trans. on Knowledge and Data Engineering 14(11), 1684–1699 (1988)

[4] Chen, Y., Cooke, D.: On the Transitive Closure Representation and Adjustable Compression. In: SAC'06, April 23-27, Dijon, France, pp. 450–455 (2006)

[5] Hopcroft, J.E., Karp, R.M.: An $n^{2.5}$ algorithm for maximum matching in bipartite graphs. SIAM J. Comput. 2, 225–231 (1973)

[6] Jagadish, H.V.: A Compression Technique to Materialize Transitive Closure. ACM Trans. Database Systems 15(4), 558–598 (1990)
[7] Keller, T., Graefe, G., Maier, D.: Efficient Assembly of Complex Objects. In: Proc. ACM SIGMOD conf. Denver, Colo., pp. 148–157 (1991)
[8] Kuno, H.A., Rundensteiner, E.A.: Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation. IEEE Transactions on Knowledge and Data Engineering 10(5), 768–792 (1998)
[9] Teuhola, J.: Path Signatures: A Way to Speed up Recursion in Relational Databases. IEEE Trans. on Knowledge and Data Engineering 8(3), 446–454 (1996)
[10] Wang, H., Meng, X.: On the Sequencing of Tree Structures for XML Indexing. In: Proc. Conf. Data Engineering, Tokyo, Japan, pp. 372–385 (April 2005)
[11] Zhang, C., Naughton, J., DeWitt, D., Luo, Q., Lohman, G.: On Supporting Containment Queries in Relational Database Management Systems. In: Proc. of ACM SIGMOD Intl. Conf. on Management of Data, California (2001)
[12] Zibin, Y., Gil, J.: Efficient Subtyping Tests with PQ-Encoding. In: Proc. of the 2001 ACM SIGPLAN conf. on Object-Oriented Programming Systems, Languages and Application, Florida, pp. 96–107 (October 14-18, 2001)
[13] Dilworth, R.P.: A decomposition theorem for partially ordered sets. Ann. Math. 51, 161–166 (1950)