On the Decomposition of Posets into Minimum Set Node-Disjoint Chains

Yangjun Chen[1], Yibin Chen[2]

Dept. Applied Computer Science, University of Winnipeg, Canada
[1]y.chen@uwinnipeg.ca, [2]chenyibin@gmail.com

Abstract - One of the most famous results in the theory of partially ordered sets is due to Dilworth (1950) who showed that the size of a minimum decomposition (into chains) of a partially ordered set S is equal to the size of a maximum antichain, which is a subset of pairwise incomparable elements. However, up to now, the bestalgorithm to decompose S into a minimum set of chains needs $O(n^3)$ time, where n is the number of the elements in S. In this paper, we address this problem and propose an algorithm which produces a minimum decomposition in $O(□ \cdot n^2)$ time and $O(m + □ \cdot n)$space, where□ is the size of a maximum antichain and m is the number of relations between elements (i.e., the number of pairs (a, c) such that $a \succ c$).In general, is much smaller than n.

Keywords: Partially Ordered Sets,; Posets; Chains, Antichains.

1. Introduction

A partial order in a set S is a relation $\succeq$such that for each a, b, and c in S:

1. $a \succeq a$is true ($\succeq$ is *reflexive*),
2. $a \succeq b$ and $b \succeq c$ imply $a \succeq c$ ($\succeq$ is *transitive*), and
3. $a \succeq b$ and $b \succeq a$ imply $a = b$ ($\succeq$ is *antisymmetric*).

If we have a *partially ordered set* (*poset*for short) $\boldsymbol{S} = (S, \succeq)$, a *chain* in S is a non-empty subset $C = \{a_1, a_2, ..., a_k\} \subseteq□ S$ such that

$a_1 \succ a_2 \succ ... \succ a_k$.

Two elements of S are called *comparable* if they appear together in some chain in $\boldsymbol{S}$; elements which are not comparable are called *incomparable*. A non-empty set, in which every pair of elements is not comparable, is called an *antichain*.

Since each single element in S is itself a chain, it is always possible to partition the elements in S into disjoint chains. Such a partition is called a*decomposition* and a decomposition consisting of the smallest number of disjoint chains is called minimum. According to Dilworth [6], the size of a minimum decomposition equals the size of a maximum antichain.

Many proofs of Dilworth's Theorem are known [5, 9, 10, 14, 17, 18]. Among them, the argument provided by Fulkerson [10] is straightforward, by which a bipartite graph G_S with bipartite (V_1, V_2) for $S = \{a_1, a_2, ...,a_n\}$ is constructed, where $V_1 = \{x_1, x_2, ..., x_n\}$, $V_2 = \{y_1, y_2, ..., y_n\}$ and an edge joining $x_i \in V_1$ to $y_j \in V_2$ whenever $a_i \succ a_j$. Let Mbe a maximum matching of $G_S$and$D$a minimum decomposition of $\boldsymbol{S}$. Fulkerson proved that $|D| = n - |M|$. On the other hand, by the König's theorem ([2], page 180), we also have □ $= n - |M|$, where □ □ is the size of a maximum antichain of $\boldsymbol{S}$. So, $|D| =$ □ . Using the algorithm proposedby Hopcroft and Karp [11], M can be found in $O(m \cdot \sqrt{n})$ time, where m is the number of all pairs (a, c) such that $a \succ c$. Therefore, the maximum size of antichains can be determined in $O(m \cdot \sqrt{n})$ time. However, the method hinted by the Fulkerson's proof cannot be efficient since to construct G_S we have to first produce all the possible *transitive relations*. By a transitive relation, we mean a relation $a \succ c$iff there exists b such that $a \succ b$ and $b \succ c$.We need $O(n^3)$ time and $O(n^2)$ space to generate all these relations.

In [12], Jagadish discussed an algorithm for finding a minimum set of node-disjoint chains that cover a directed acyclic graph $G(V, E)$ (*DAG* for short, which contains no cycles and can be considered as a poset) by first creating the transitive closure TC_G of G and then finding a minimum set of node-disjoint paths of TC_G. TC_G itself is a directed graph $G^*(V, E^*)$ with $(v, u) \in□ E^*$ iff there is a path from v to u in G. Thus, the problem can be solved by transforming it to a *min* network flow [7, 13, 20]. The time complexity of finding a transitive closure is $O(n^□)$, and the *min* network flow also needs $O(n \cdot m)$ time.So the time complexity of the whole working process is bounded by $O(n^□)$.

In this paper, we propose an efficient algorithm to find a minimized set of disjoint chains for $\boldsymbol{S}$. For this purpose, we represent $\boldsymbol{S}$ as a DAG, in which we have an arc $u \to□ v$ if $u \succ v$. Removing any arc $u \to□ v$ if there is path of length ≥ 2 from u to v, we get another graph. A minimum set of node-disjoint chains that cover the graph G must be a decomposition of $\boldsymbol{S}$.

The algorithm runs in $O(□ \cdot n^2)$ time and in $O(m + □ \cdot n)$ space.

The remainder of the paper is organized as follows. In Section 2, we discuss an algorithm to stratify a DAG into different levels and review some concepts related to bipartite graphs, on which our method bases. Section 3 is devoted to the description of our algorithm to decompose a DAG into chains. Finally, a short conclusion is set forth in Section 5.

2. Graph stratification and bipartite graph

Our method 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.

© 2013. The authors - Published by Atlantis Press

2.1 Stratification of DAGs

Let $G(V, E)$ be a DAG with $|V| = n$ and $|E| = m$. We decompose V into subsets $V_0, V_1,...,V_h$ such that $V = V_0 \cup V_1 \cup ... \cup V_h$ and each node in V_i has its children appearing only in $V_{i-1}, ..., V_0$ ($i = 1, ..., h$), 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 all those children of v, which appear 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 use ($j < i$) $\boldsymbol{C}_j^i$ to represent $C_j(v_1) \cup ... \cup C_j(v_l)$.

Such a DAG decomposition can be done in O(m) time, by using an algorithm discussed in [4].

As an example, consider the graph shown in Fig. 1(a). In Fig. 1(b), the nodes of the DAG shown in Fig. 1(a) are divided into four levels: $V_0 = \{a_0, b_0, c_0, d_0, e_0\}$, $V_1 = \{b_1, c_1, d_1, e_1\}$, $V_2 = \{b_2, c_2, d_2, e_2\}$, and $V_3 = \{b_3, c_3, d_3\}$. Associated with each node at each level is a set of links pointing to its children at different levels.

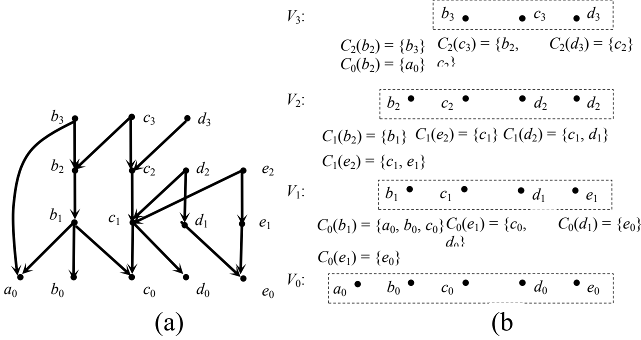

Fig. 1 A DAG and its stratification

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 $B(V_B, E_B)$ is bipartite if the node set V_B can be partitioned into two sets V_1 and V_2 in such a way that no two nodes from the same set are adjacent. We also denote such a graph as $B(V_2, V_1; E_B)$. □

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

Definition 2 (*matching* [2]) Let $B(V_2, V_1; E_B)$ be a bipartite graph. A subset of edges $E' \subseteq E_B$ is called a *matching* if no two edges in E' have a common end node. A matching with the largest possible number of edges is called a *maximal matching*. □

Let M be a matching of a bipartite graph $B(V_2, V_1; E_B)$. 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 $E_B \backslash M$ and M. A path is an *augmenting path* if it is an alternating path with free origin and terminus. Let $v_1 \to v_2 \to ... \to v_k$ be an alternating path with $(v_i, v_{i+1}) \in E_B \backslash M$ and $(v_{i+1}, v_{i+2}) \in M$ (i= 1, 3, ...). By transferring the edges on the path, we change it to another alternating path with $(v_i, v_{i+1}) \in M$ and $(v_{i+1}, v_{i+2}) \in E_B \backslash M$ (i= 1, 3, ...). In addition, we will use $free_M(V_1)$ and $free_M(V_2)$ to represent all the free nodes in V_1 and V_2, respectively. Finally, if $(u, v) \in M$, we say, u covers v with respect to M, and *vice versa*. Much research on finding a maximal matching in a bipartite graph has been done. The best algorithm for this task is due to Hopcroft and Karp [11] and runs in O($m \cdot \sqrt{n}$) time, where $n = |V_B|$ and $m = |E_B|$. The algorithm proposed by Alt, Blum, Melhorn and Paul [1] needs O($n^{1.5} \cdot \sqrt{m/(\log n)}$) time. In the case of large m, the latter is better than the former.

3. Algorithm description

In this section, we describe our algorithm for the DAG decomposition.

The main idea behind it is a kind of newly introduced arcs, called *virtual arcs*, used to transfer the reachability information from lower levels to higher levels. First, we discuss an example to motivate such a concept in 3.1. Then, in 3.2, we give a formal definition of virtual arcs and show how they can be used to create disjoint chains. In 3.3, we briefly discuss the removing of virtual arcs from created chains to get a final result.

3.1 Chain creation

The main idea of our algorithm is to construct a series of bipartite graphs for $G(V, E)$ and then find a maximal matching for each of such bipartite graphs using Hopcroft-Karp algorithm. However, by simply combining all the maximal matchings, we may get a set of chains, which is not minimal. The reason for this is that a free node relative to a certain maximal matching is not considered for a bipartite graph at a higher level. Therefore, it should be hoisted and involved in the computation fora next bipartite graph.Especially, some new arcs incident to itshould be created.

We start our discussion with the following specification:

$V_0' = V_0$.

$V_i' = V_i \cup$ {free nodes at lower levels, but hoisted to V_i} for $1 \leq i \leq h - 1$.}

$\boldsymbol{C}_i = \boldsymbol{C}_{i-1}^i \cup$ {all the new arcs from the nodes in V_i to the nodes hoisted to V_{i-1}'} for $1 \leq i \leq h - 1$.

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

M_i - a maximal matching of $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$.

N_i - a $|V_{i-1}'| \times |V_i|$ matrix, representing $G(V_i, V_{i-1}'; C_i)$.

L_i - a $|V_i| \times (n - (|V_i| + \ldots + |V_0|))$ matrix, representing all those arcs in E, which connect the node in $G \backslash (V_0 \cup \ldots \cup V_i)$ to the nodes in V_i.

In addition, we distinguish between two kinds of new arcs, as defined below.

Definition 3(*transitive arcs*)Let v be a node (in V_{i-1}') hoisted to V_i. If there exist $u \to w \in E$ and $w \to v \in \boldsymbol{C}_i$ with $u \in V_j$ (for some $j > i$) and $w \in V_i$, then, add $u \to v$ if it is not an arc in E. The new arc is referred to as a transitive arc.

Definition 4(*alternating arcs*)Let v be a node (in V_{i-1}') hoisted to V_i. If there exists $w \in V_{i-1}'$ such that v is connected to w through an alternating path $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$, and $u \in V_j$ (for some $j > i$) such that one of the two conditions holds:

- $u \to w \in E$, or
- there is a node $w' \in V_i$ such that $u \to w' \in E$ and $w' \to w \in \boldsymbol{C}_i$,

add $u \to v$ if it is not an arc in E or has not yet been created as a transitive arc. The new arc is referred to as an alternating arc. □

We further distinguish between two kinds of transitive arcs:

- $u \to v$ is an *actual* transitive arc if there is a path in E which connects u and v;
- $u \to v$ is a *virtual* transitive arc if any path connecting u and v contains at least one alternating arc.

Both virtual transitive and alternating arcs are called *virtual arcs*. The following example helps for illustration.

Example 1 Consider the graph shown in Fig. 1(a). This graph can be divided into four levels as shown in Fig. 1(b). The first bipartite graph $G(V_1, V_0; \boldsymbol{C}_1)$ is shown in Fig. 5(a). A possible maximal matching M_1 of it is shown in Fig. 2(b).

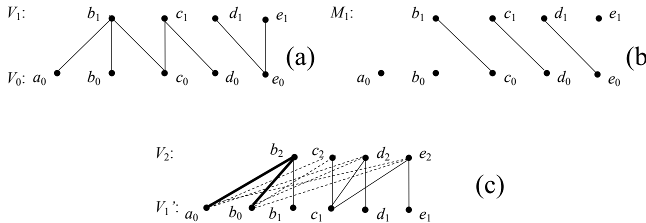

Fig. 2 Illustration for alternating arcs

Relative to M_1, a_0 and b_0 are two free nodes in V_0.

So they will be promoted to V_1. At the same time, two transitive arcs $b_2 \to a_0$ and $b_2 \to b_0$ (represented by two thick arrows in Fig. 2(c)) will be created for the following reasons:

- There exists $b_1 \in V_1$ such that $b_2 \to b_1$ and $b_1 \to a_0$.
- There exists $b_1 \in V_1$ such that $b_2 \to b_1$ and $b_1 \to b_0$.

In addition, six alternating arcs: $c_2 \to a_0$, $d_2 \to a_0$, $e_2 \to a_0$, $c_2 \to b_0$, $d_2 \to b_0$ and $e_2 \to b_0$, will also be created (see dashed arrows in Fig. 2(c)):

- $c_2 \to a_0$, $d_2 \to a_0$ and $e_2 \to a_0$ are created due to the alternating path $a_0 - b_1 - c_0$ in $G(V_1, V_0; \boldsymbol{C}_1)$ and the reachability of c_0 respectively from c_2, d_2 and e_2 through c_1 in V_1.
- $c_2 \to b_0$, $d_2 \to b_0$ and $e_2 \to b_0$ are created due to the alternating path $b_0 - b_1 - c_0$ and the reachability of c_0 respectively from c_2, d_2 and e_2 through c_1 in V_1.

We notice that $P_1 = a_0 - b_1 - c_0 - c_1 - d_0$ and $P_2 = b_0 - b_1 - c_0 - c_1 - d_0$ in G $(V_1, V_0; \boldsymbol{C}_1)$ are another two alternating paths starting from a_0 and b_0, respectively. P_1 and the reachability of d_0 from c_2, d_2 and e_2 through c_1 in V_1 will also lead to the creation of the first three alternating arcs; and P_2 and the reachability of d_0 from c_2, d_2 and e_2 through c_1 in V_1 will also lead to the creation of the remaining three alternating arcs. However, each new arc is recorded only once.

In order to create such new arcs efficiently, we can use the matrix multiplication. For example, by producing $NL_1 = N_1 \times L_1$, we will easily get all the transitive arcs incident to all the free nodes in V_0 when they are promoted to V_1. However, to create all the alternating arcs, more effort is needed with the following procedure being used, in which we denote by $L(u, *)$ and $L(*, v)$ a row corresponding to node u and a column corresponding to a node v in a matrix L, respectively. For two graphs G_1, G_2, we will also use $G_1 \backslash G_2$ to stand for a graph obtained by deleting the arcs of G_2 from G_1; and $G_1 \cup G_2$ for a graph obtained by adding the arcs of G_1 and G_2 together.

1. Let v be a free node in V_0. Figure out all those nodes $u_1, \ldots, u_k$ in V_0 such that each u_i ($i = 1, \ldots, k$) is connected to v through an alternating path relative to M_1.
2. Let $v_1, \ldots, v_j$ be all the nodes in V_1. Denote $L_0' = L_0 \backslash (L_0(*, v_1) \cup \ldots \cup L_0'(*, v_j))$. (Recall that L_0 is a matrix representing all those arcs in E, which connect the nodes in $G \backslash V_0$ to the nodes in V_0.)
3. Add (bit-wise OR) $NL_1(u_1, *)$, … $NL_1(u_k, *)$ to $NL_1(v)$. Add $L_0'(u_1, *)$, … $L_0'(u_k, *)$ to $NL_1(v, *)$.
4. Repeat (1) - (3) for each free node in V_0.

Example 2 Continued with Example 1. With respect to the first bipartite graph, we have bipartite graph, we have bipartite graph, we have

$$L_0 = \begin{array}{c} \\ a_0 \\ b_0 \\ c_0 \\ d_0 \\ e_0 \end{array} \begin{array}{c} \begin{array}{ccccccccccc} b_1 & c_1 & d_1 & e_1 & b_2 & c_2 & d_2 & e_2 & b_3 & c_3 & d_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$L_0' \quad \begin{array}{c|ccccccc} & b_2 & c_2 & d_2 & e_2 & b_3 & c_3 & d_3 \\ \hline a_0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ b_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ c_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ d_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ e_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$N_1 = \begin{array}{c|cccc} & b_1 & c_1 & d_1 & e_1 \\ \hline a_0 & 1 & 0 & 0 & 0 \\ b_0 & 1 & 0 & 0 & 0 \\ c_0 & 1 & 1 & 0 & 0 \\ d_0 & 0 & 1 & 0 & 0 \\ e_0 & 0 & 0 & 1 & 1 \end{array} \qquad L_1 \quad \begin{array}{c|ccccccc} & b_2 & c_2 & d_2 & e_2 & b_3 & c_3 & d_3 \\ \hline b_1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ c_1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ d_1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ e_1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}$$

$$NL_1 = N_1 \times L_1 = \begin{array}{c|ccccccc} & b_2 & c_2 & d_2 & e_2 & b_3 & c_3 & d_3 \\ \hline a_0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ b_0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ c_0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ d_0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ e_0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}$$

In NL_1, $NL_1(a_0, b_2) = 1$ and $NL_1(b_0, b_2) = 1$ represent the two transitive arcs newly created (see Example 1.)

In order to create all the alternating arcs, we will check, for each free node in V_0, all those nodesconnected toit through an alternating path.

For example, both c_0 andd_0 are reachable from a_0.So we need to add $L_0'(c_0, *)$and$L_0'(d_0, *)$ (both are (0, 0, 0, 0, 0, 0, 0)), as well as$NL_1(c_0, *) = (1, 1, 1, 1, 0, 0, 0)$ and $NL_1(d_0, *) = (0, 1, 1, 1, 0, 0, 0)$ to$NL_1(a_0, *)$ to get all the alternating arcsincident to a_0:

$$NL_1(a_0, *) = (1, 1, 1, 1, 0, 0, 0).$$

In addition, we canalso add$L_0'(a_0, *) = (0, 0, 0, 0, 1, 0, 0)$ to $NL_1(a_0, *)$. Then,all the arcs incident to a_0 can be represented by $NL_1(a_0, *)$:

$$NL_1(a_0, *) = (1, 1, 1, 1, 1, 0, 0).$$

In the same way, we will change $NL_1(b_0, *)$ to

$$NL_1(b_0, *) = (1, 1, 1, 1, 0, 0,0).$$

The problem of the above working process is that the difference between the virtual arcs and the actual arcs cannot be recognized; but they need to be handled differently in the subsequent computation. For this reason, we introduce a variant of the Boolean algebra, by which we have three values: 0, 1, and **1**, defined as follows:

- $N_i(v, u) = 0$ if u is not connected to v through any path.
- $N_i(v, u) = 1$ if $u \rightarrow v$ is an arc in E or an actual transitive arc.
- $N_i(v, u) = \mathbf{1}$ if $u \rightarrow v$ is an alternating arc or a virtual transitive arc.

In the subsequent discussion, we refer to**1** as a *marked* 1.Furthermore, the definitions of the $\wedge$ and$\vee$ operations in the classical Boolean algebra need to be slightly changed as follows:

$\wedge$	0	1	**1**
0	0	0	0
1	0	1	**1**
1	0	**1**	**1**

$\vee$	0	1	**1**
0	0	1	**1**
1	1	1	1
1	**1**	1	**1**

It is almost the same as the classical Boolean algebra. However, by the new Boolean algebra, the properties of new arc can be simply represented when they are created by the matrix multiplication or by the vector bit-wise OR operations.

Based on the new Boolean algebra, we design a general process for creating new arcs for the free nodes in V_{i-1}' ($i \geq 1$) relative to M_i when they are promoted to V_i. Initially, NL_0 is set to be $\emptyset$.

Algorithm*CNA*(*F*)
Input: F – a set of free nodes in V_{i-1}' relative to M_i.
Outout: a set of new arcs.

begin

1. Let$F = \{v_1, \ldots, v_k\}$.
2. ConstructL_i andN_i for $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$.
3. (*create transitive arcs*) Create $NL_i = N_i \times L_i$;
4. For each v_j ($j = 1, \ldots, k$),let $w_1, \ldots, w_p$ be all those nodes inV_{i-1}'each connected tov_j through an alternating path in $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$.
   i) For each w_q ($q = 1, \ldots, p$), make a copy x_q of $NL_i(w_q, *)$ and a copy y_q of $L_{i-1}'(w_q, *)$.
   ii) Change each 1 in $x_q$to**1**; change each 1 in y_q to **1**. (*Note that all **1**'s in x_q and y_q remain unchanged.*)
   iii) (*create alternating arcs*) For each q,add both $x_q$and $y_q$to $NL_i(v_j, *)$.
5. (add remaining arcs) For each v_j ($j = 1, \ldots, k$),add $L_{i-1}'(v_j, *)$ to $NL_i(v_j, *)$ if $v_j \in V_{i-1}$; otherwise (v_j is a node promoted to V_{i-1}' from a lower level), add $NL_{i-1}(v_j, *)$ to $NL_i(v_j, *)$.
6. Remove L_{i-1}, L_{i-1}', and NL_{i-1} (since they will not be used any more.)

end

In the above algorithm, the execution of line 3 will create all the actual and virtual transitive arcs while the execution of line 4 will create all new alternating arcs incident to the nodes hoisted to V_i. In line 5, all the free nodesin v_j(relative to M_i) in V_{i-1}' are divided into two groups. The first group contains all those free nodesbelonging to V_{i-1} and for each$v_j$in this groupwe add $L_{i-1}'(v_j, *)$ to $NL_i(v_j, *)$. In the second group, each free node$v_j$is a node promoted from a lower level to V_{i-1}', for whichwe add $NL_{i-1}(v_j, *)$ to $NL_i(v_j, *)$. Thus, any arc (in E or newly created) connectinga node $u \in G \backslash (V_0 \cup \ldots \cup V_i)$ to $v_j$will

be stored in $NL_i(v_j, u)$.In line 6, we remove L_{i-1}, L_{i-1}', and NL_{i-1} since in the subsequent computation they will not be used any more. However, N_{i-1} is kept since all N_i's will be used to remove alternating arcs after we have generated a minimum set of chains.

Example 3 As shown in Example 2, by applying the above algorithm to create the new arcs fora_0 andb_0, we will get

$NL_1(a_0, *)$ = (1, **1**,**1**,**1**,1, 0, 0), and

$NL_1(b_0, *)$ = (1, **1**, **1**, **1**, 0, 0,0).

Accordingly,$G(V_2, V_1; \boldsymbol{C}_1^2)$ is changed to$G(V_2, V_1'; \boldsymbol{C}_2)$, as shown in Fig. 2(c).

Assume that the maximal matching M_2 found for $G(V_2, V_1'; \boldsymbol{C}_2)$ is shown in Fig. 3.

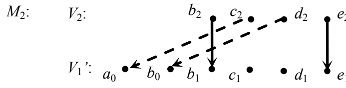

Fig. 3 Maximum matching for $G(V_2, V_1'; \boldsymbol{C}_2)$

Relative to M_2, c_1 andd_1 are two free nodes in V_1'. So they will be hoisted to V_2. Again, in order to determine the arcs incident to them, we will generate N_2, L_2, and

$NL_2 = N_2 \times L_2$as shown below:

$$N_2 = \begin{array}{c} \\ a_0 \\ b_0 \\ b_1 \\ c_1 \\ d_1 \\ e_1 \end{array} \begin{array}{c} b_2 \; c_2 \; d_2 \; e_2 \\ \begin{pmatrix} 1 & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ 1 & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \qquad L_2 = \begin{array}{c} \\ b_2 \\ c_2 \\ d_2 \\ e_2 \end{array} \begin{array}{c} b_3 \; c_3 \; d_3 \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$NL_2 = N_2 \times L_2 = \begin{array}{c} \\ a_0 \\ b_0 \\ b_1 \\ c_1 \\ d_1 \\ e_1 \end{array} \begin{array}{c} b_3 \; c_3 \; d_3 \\ \begin{pmatrix} 1 & 1 & \mathbf{1} \\ 1 & 1 & \mathbf{1} \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

In addition, L_1' can be easily constructed by removing all the columns corresponding to the nodes inV_2 from L_1. It is a matrix with each entry being 0.We notice that a_0 is connected toc_1 through an alternating path c_1□ □c_2□ □a_0 in $G(V_2, V_1'; \boldsymbol{C}_2)$ and b_0 is connected tod_1 through another alternating path d_1 □ □d_2 □ □d_1. Then, the following operations will be carried out by running Algorithm *CNA*( ):

1. Create a copy x of $NL_2(a_0, *)$ = (1, 1,**1**) and a copy $x'$of $NL_2(b_0, *)$ = (1, 1,**1**). Create a copy $y$of $L_1'(a_0, *)$ and a copy $y'$of $L_1'(b_0, *)$ (both x' and y' are (0, 0, 0).)
2. Change each 1 in x, x', $y$and y' to **1**.
3. Add x and$y$to $NL_2(c_1, *)$ = (0, 1, 1). Then, add $L_1'(c_1, *)$ to$NL_2(c_1, *)$.These operations will change $NL_2(c_1, *)$ to (**1**,1,1).
4. Add x', y' and $L_1'(d_1, *)$ to $NL_2(d_1, *)$. Thiswill change$NL_2(d_1, *)$ to (**1**,**1**,**1**).

Thus, $G(V_3, V_2'; \boldsymbol{C}_3)$ will be a bipartite graph as shown in Fig. 4(a).

Assume that the maximal matching M_3 found for $G(V_3, V_2'; \boldsymbol{C}_3)$ is a set of edges shown in Fig. 4(b). Then, by combining M_1, M_2, and M_3, we will get a set of chains as shown in Fig. 5.

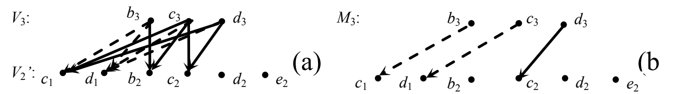

Fig. 4 A bipartite graph and a maximum matching

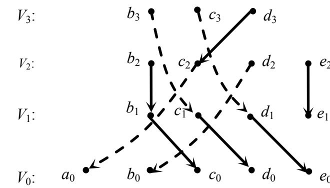

Fig. 5A set of chains containing alternating arc

This set contains 6 chains. It must be minimal since there exists a subset of 6 nodes {a_0, b_0, c_0, d_0, d_1, e_1} in the graph, in which each pair of nodes are not reachable from each other. However, some of chains contain alternating arcs which should be removed to get the final result.□

3.3 Removing alternating arcs

We call an arc on a chain a chain arc. Our purpose is to replace each alternating chain arc with an arc in E or a real transitive arc.

To this end, we use A_i (i = 1, …,h - 1) to represent a set containing all those alternating chain arcs at level i, i.e., all the alternating arcs e with □ (e) = i. Notice that anA_i may be ∅.

Then, we proceed to remove A_j'stop-down level by level in the descending order of level numbers. In general, we can remove an alternating chain arc in two possible ways as follows.

Let u_1→□ v_1, …,u_k→□ $v_k$be all the alternating chain arcs with each v_j(j =1, …, k) being in V_i'.Then, v_j(j =1, …,k) must be a node hoisted toV_i' from V_{i-1}'. For each u_j, we will search G from u_j to the nodes in V_i andconnect u_j to all those nodes in V_i, which are reachable from $u_j$through a path in E. In addition, we will descend any node$u$ at a level higher than level i+ 1 to i

+ 1if itis the ending node of some chain found up to now; and connect it to all those nodes in V_i', which are reachable from it through a path in E. We denote by U_{i+1}such a new graph. Removing all the alternating arcs from U_{i+1}, we will get another graph U_{i+1}'. Let M_{i+1}'be a set of edges obtained by removing all the alternating edges from M_{i+1}. The first way is to remove an alternating arc$u_j \rightarrow$□ $v_j$by finding an alternating path P relative to M_{i+1}'in U_{i+1}', satisfying one of the following conditions:

1. P starts at $v_j$and ends at a node descended to level i+ 1, or
2. P starts at a node u (in V_i), which is the starting node of a chain, and ends at u_j.

4. Conclusion

In this paper, a new algorithm for finding a minimal chain decomposition of a partially ordered set ***S*** is proposed. The algorithm needs O(□ $\cdot n^2$) time and O(m + □ $\cdot n$) space, where n is the number of the elements in ***S***,m is the number of relations between elements and □ is the size of a maximum antichain. The main idea of the algorithm is the concept of virtual arcs and the DAG stratification that generates a series of bipartite graphs which may contain virtual arcs. By theHopcropt-Karp's algorithm, we find a maximal matching for each of such bipartite graphs, which make up a set of node-disjoint chains. A next step is needed to replace all the virtual chains with the arcs in E or the actual transitive arcs to get the final result.

References

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

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

[3] C. Chekuri and M. Bender, An Efficient Approximation Algorithm for Minimizing Makespan on Uniformly Machines, *Journal of Algorithms* 41, 212-224(2001).

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

[5] G.B. Dantzig and A. Hoffman, On a theorem od Dilworth, *Linear Inequalities and related systems* (H.W. Kuhn and A.W. Tucker, eds.) Annals of Math. Studies 38(1966), 207-214.

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

[7] 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.

[8] S. Felsner, L. Wernisch, Maximum *k*-chains in planar point sets: combinatorial structure and algorithms, *SIAM J. Comp*. **28**, 1998, pp. 192-209.

[9] T. Gallai and A.N. Milgram, Verallgemeinerung eines Graphentheoretischen Satzes von Reedei. *Acta Sci. Math. Hung*., 21(1960), 429-440.

[10] D.R. Fulkerson, Note on Dilworth's embedding theorem for partially ordered sets, *Proc. Amer. Math. Soc*. 7(1956), 701-702.

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

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

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

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

[15] R.-D. Lou, M. Sarrafzadeh, An optimal algorithm for the maximum two-chain problem, *SIAM J. Disc. Math*.**5**(2), 1992, pp. 285-304.

[16] V.M. Malhotra, M.P. Kumar, and S.N. Maheshwari, An O($|V|^3$) Algorithm For Finding Maximum Flows in Networks, Computer Science Program, Indian Institute of Technology, Kanpur 208016, India, 1978.

[17] M.A. Perles, A proof of Dilworth's decomposition theorem for partially ordered sets, *Israel J. of Math*. 1(1963), 105-107.

[18] H. Tverberg, On Dilworth's decomposition theorem for partially ordered sets, *J. Comb. Th.* 3(1967), 305-306.

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

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

[21] L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Communication of the ACM* 21(7), July 1978, 95-114.

[22] H. Goeman, Time and Space Efficient Algorithms for Decomposing Certain Patially Ordered Sets, PhD thesis, Department of Mathematics-Science, Rheinischen Friedrich-WilhelmsUniversität Bonn, Germany, Dec. 1999.

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

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