On the DAG Decomposition

Yangjun Chen[1*] and Yibin Chen[1]

Please check corrsponding author name

[1]The University of Winnipeg, Canada.

Article Information

DOI: 10.9734/BJMCS/2015/19380

Editor(s):

(1)

Reviewers:

(1)

(2)

(3)

Complete Peer review History:

Original Research Article

Received: 06 June 2015

Accepted: …………….. 2015

Published: ……………. 2015

Abstract

In this paper, we propose an efficient algorithm to decompose a directed acyclic graph G into a minimized set of node-disjoint chains, which cover all the nodes of G. For any two nodes u and v on a chain, if u is above v then there is a path from u *to* v in G. The best algorithm for this problem up to now needs $O(n^3)$ time, where n is the number of the nodes of G. Our algorithm, however, needs only $O(\kappa \cdot n^2)$ time, where κ is G's width, defined to be the size of a largest node subset U of G such that for every pair of nodes $x, y \in U$, there does not exist a path from x to y or from y to x. More importantly, by the existing algorithm, $O(n^2)$ extra space (besides the space for G itself) is required to maintain the transitive closure of G to do the task while ours needs only $O(\kappa \cdot n)$ extra space.

Keywords: reachability queries; directed graphs; transitive closure; graph decomposition.

1 Introduction

Let G be a directed acyclic graph (a *DAG* for short). A *chain cover* of G is a set C of *node-disjoint chains* such that it covers all the nodes of G, and for any two nodes u and v on a chain $p \in C$, if u is above v then there is a path from u to v in G. In this paper, we discuss an efficient algorithm to find a *minimized* C for G.

As an example, consider the DAG shown in Fig. 1(a). We can decompose it into a set of two chains, as shown in Fig. 1(b), which covers all the nodes of G. Fig. 1(c) shows another possible minimized decomposition.

With the advent of the web technology, the efficient decomposition of a DAG G into a minimum set of chains becomes very important; especially, for the applications involving massive graphs such as social

**Corresponding author: Email: y.chen@uwinnipeg.ca;*

networks, for which we may quite often ask whether a node v is reachable from another node u through a path in G.

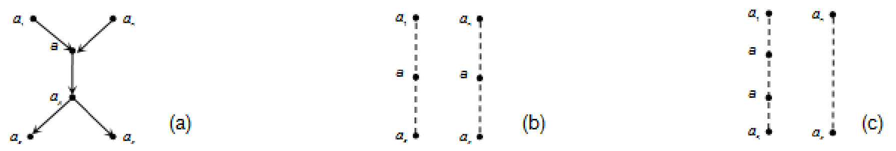
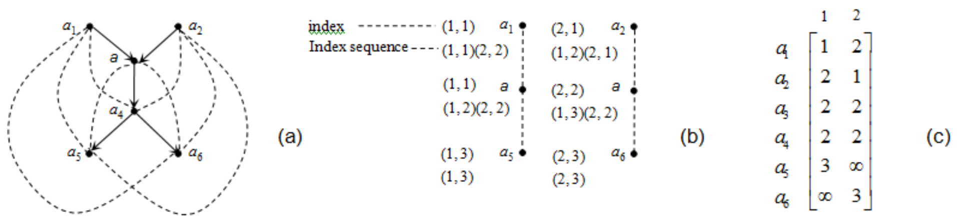


Fig. 1. Illustration for DAG decomposition

A naive method to answer such a query is to precompute the reachability between every pair of nodes in G (V, E) - in other words, to compute the transitive closure of G, which is also a directed graph $G^*(V, E^*)$ with $(v, u) \in E^*$ if and only if there is a path from v to u in G. (See Fig. 2(a) for illustration, in which we show the transitive closure of the graph shown in Fig. 1(a).).



$$\begin{array}{c|cc} & 1 & 2 \\ \hline a_1 & 1 & 2 \\ a_2 & 2 & 1 \\ a_3 & 2 & 2 \\ a_4 & 2 & 2 \\ a_5 & 3 & \infty \\ a_6 & \infty & 3 \end{array}$$

Fig. 2. Illustration for transitive closure and index sequences

As it is well known, the transitive closure of G can be stored as a boolean matrix $\boldsymbol{M}$ such that $\boldsymbol{M}[i, j] = 1$ if there is path from i to j [21]; otherwise, $\boldsymbol{M}[i, j] = 0$ [26]. Then, a reachability query can be answered in a constant time. However, this requires $O(n^2)$ space for storage, which makes it impractical for very large graphs, where $n = |V|$. Another method is to compute the shortest path from u to v over such a large graph on demand. Therefore, it needs only $O(m)$ space, but with high query processing cost - $O(m)$ time in the worst case, where $m = |E|$. However, if we are able to decompose a DAG into a minimum set of chains, we can effectively compress a transitive closure without increasing much query time, as described below.

Let G be a directed graph. If it is cyclic (i.e., it contains cycles), we can first 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* are equivalent to its representative as far as reachability is concerned since each pair of nodes in an SCC are reachable from each other. In this way, we transform G to a DAG. Next, we decompose the DAG into a minimum set C of node-disjoint chains. (Recall that if a node u appears above another node v on a chain, there is a path from u to v.) Denote $|C| = \kappa$. We will then

(1) Number each chain and number each node on a chain; and
(2) Use a pair (i, j) as an index for the jth node on the ith chain.

Besides, each node u on a chain will be associated with an index sequence of the form: $(r, j_r) \ldots (i, j_i) \ldots (k, j_k)$ $(1 \le r \le i \le k \le \kappa)$ such that any node v with index (x, y) is a descendant of u if and only if there exists (x, j_x) in the sequence with $y \ge j_x$. (See Fig. 2(b) for illustration.) Such index sequences can be created as follows.

First of all, we notice that we can associate each leaf node with an index sequence, which contains only one index, i.e., the index assigned to it. Clearly, such an index sequence is trivially sorted and its length is $1 \le \kappa$. Let v be a non-leaf node with children $v_1, \ldots, v_l$ each associated with an index sequence L_i $(1 \le i \le l)$. Assume that $|L_i| \le \kappa$ $(1 \le i \le l)$ and the indexes in each L_i are sorted according to the first element in each index. We will create an index sequence L for v, which initially contains only the index assigned to it. Then, we will merge all L_i's into L one by one. To merge an L_i into L, we will scan both L and L_i from left to right. Let (a_1, b_1) (from L) and (a_2, b_2) (from L_i) be the index pairs currently encountered. We will perform the following checking's:

- If $a_2 > a_1$, we go to the index next to (a_1, b_1) (in L) 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) (in L). Go to the index next to (a_2, b_2) (in L_i) 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 (in (a_1, b_1)) with b_2. In both cases, we will go to the indexes next to (a_1, b_1) (in L) and (a_2, b_2) (in L_i), respectively.
- We will repeat the above three steps until either L or L_i is exhausted. If when L is exhausted L_i still has some remaining elements, append them at the end of L.

Obviously, after all L_i's have been merged into L, the length of L is still bounded by the number κ. Denote by d_v the outdegree of v. The time spent on this process is then bounded by $O(\sum_v d_v \cdot \kappa) = O(\kappa \cdot m)$, but the space overhead is only $O(\kappa \cdot n)$. The query time remains O(1) if we store the index sequences as a matrix $\boldsymbol{M}_G$, as shown in Fig. 2(c), in which each entry $\boldsymbol{M}_G(v, j)$ is the jth element in the index sequence associated with node v. So, a node u with index (i, j) is a descendant of node v if and only if $\boldsymbol{M}_G(v, i) \leq j$. In practice, κ is in general much smaller than n. In this sense, G^* is effectively compressed based on a minimized decomposition of G.

The problem to decompose a DAG is also heavily related to another theoretical problem [7], [8], [11], [16], [19], [20]: the decomposition of *partially ordered sets* (or *posets* for short) $\boldsymbol{S} = (S, \succeq)$ into a minimum set of chains, where S is a set of elements and $\succeq$ is a *reflexive*, *transitive*, and *antisymmetric* relation over the elements. We can represent any poset $\boldsymbol{S}$ as a DAG G, where each node stands for an element in S and each arc $u \to v$ for a relation. Obviously, all the transitive relations in $\boldsymbol{S}$ can be represented by the transitive closure G^* of G. According to Dilworth [8], the size of a minimum decomposition equals the size of a maximum antichain U, which is a subset of elements such that for each two elements $a, c \in U$, $a \not\succ c$ and $c \not\succ a$. Furthermore, by using the Fulkerson's method [12], a minimum set of chains can be found in $O(n^3)$ time as follows:

i) Construct the transitive closure G^* of G representing $\boldsymbol{S} = (S, \succeq)$.
ii) Let $S = \{a_1, a_2, ..., a_n\}$. Construct a bipartite graph G_S with bipartite (V_1, V_2), where $V_1 = \{x_1, x_2, ..., x_n\}$, $V_2 = \{y_1, y_2, ..., y_n\}$ and an edge joins $x_i \in V_1$ to $y_j \in V_2$ whenever $a_i \to a_j \in G^*$.
iii) Find a maximal matching M of G_S. Then, for any two edges $e_1, e_2 \in M$, if $e_1 = (x_i, y_k)$ and $e_2 = (x_k, y_j)$, connect e_1 to e_2 (by identifying y_k with x_k.)

According to Fulkerson [12], the number of chains constructed as described above is $n - |M|$. It must be minimum since in terms of König's theorem ([2], page 180), the size of a maximum antichain U of $\boldsymbol{S}$ is also $n - |M|$ and we are not able to place any two elements in U on a same chain. Thus, we have $\kappa = |U|$, referred to as the width of G (or $\boldsymbol{S}$).

See Fig. 3 for illustration.

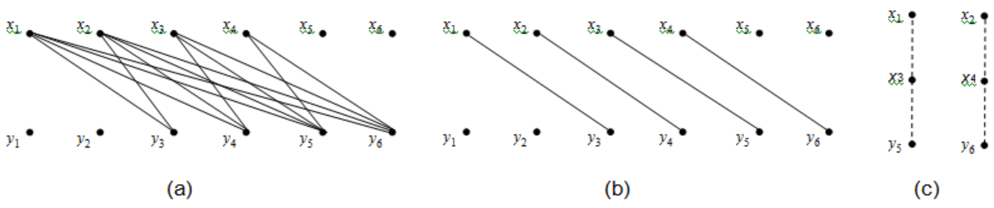


Fig. 3. Illustration for poset decomposition

The dominant cost of the above process is obviously the time and space for constructing G^*. They are bounded by O(n^3) and O(n^2), respectively [26]. However, using the algorithm proposed by Hopcroft and Karp [13], M can be found in O ($e\cdot\sqrt{n}$) time, where e is the number of the arcs in G^*, bounded by O(n^2).

In [14], Jagadish discussed an algorithm for finding a minimum set of node-disjoint paths that cover a directed acyclic graph G by transforming the problem to a *min* network flow [9], [15], [18], [22]. Its time complexity is bounded by O($n\cdot m$). However, a chain is in general not a path. For any pair of nodes u and v on a chain, we only require that if u appears above v, there is a path from u to v [23]. So, the number of paths found by the method discussed in [14] is generally much larger than the minimal number of node-disjoint chains. However, if we apply the Jagadish's method to G^*, we can get a minimized set of chains of G. But again, O(n^3) time and O(n^2) space are required to construct G^*.

The method discussed in [3] is also to decompose a DAG into node-disjoint chains. It runs in O($n^{2.5}$) time. However, the decomposition found is not minimum. Our earlier algorithm [4] works for the same purpose. Its time complexity is bounded by O($k^{1.5}n$), where k is the number of the chains, into which the DAG is decomposed. But in some cases it fails to find a minimum set of chains since when generating chains, only part of reachability information is considered. This problem is removed by [5] and [6] both with the same time complexity O($\kappa\cdot n^2$). However, in the method discussed in [5] each node is associated with a large data structure and requires O($\kappa\cdot n^2$) space in the worst case. By [6], the generated chains may contain some newly created nodes, but how to remove such nodes are not discussed at all.

Different from the above strategies, the algorithm discussed in [10] is to find a maximum k-chain in a planar point set $M \subseteq N \times N$, where $N = \{0, 1, ..., n - 1\}$ and is defined by establishing $(i, j) \prec (i', j')$ if and only if $i' > i$ and $j' > j$. So M is a special kind of posets. A k-chain is a subset of M that can be covered by k chains. The time complexity of this algorithm is bounded by O((n^2/k)/log$n$). The algorithms discussed in [17] and [24] are to find a maximum 2-chain and 1-chain in M, respectively. [17] needs $\Theta(n\cdot\log n)$ time while [24] needs only O($p\cdot n$) time, where p is the length of the longest chain.

In this paper, we propose an efficient algorithm to find a minimum set of chains for G. It runs in O($\kappa\cdot n^2$) time and in O($\kappa\cdot n$) space while the best algorithm for this problem needs O(n^3) time and in O(n^2) 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 is based. Section 3 is devoted to the description of our algorithm to decompose a DAG into chains, as well as the analysis of its computational complexities. In Section 4, we prove the correctness of the algorithm. Finally, a short conclusion is set forth in Section 5.

2 Graph Stratification and Bipartite Graphs

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

2.1 Stratification of DAGs

We first discuss the DAG stratification.

Definition 1 Let $G(V, E)$ be a DAG. We decompose V into subsets $V_0, V_1, ..., V_h$ such that $V = V_0 \cup V_1 \cup ... \cup V_h$ and each node in V_i has its children appearing only in $V_{i-1}, ..., V_0$ ($i = 1, ..., h$), 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 $level(v) = i$. We also use $C_j(v)$ ($j < i$) to represent a set of links which start from v to all those v's children, 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 $\boldsymbol{C}_j^i$ ($j < i$) 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 the following algorithm, in which we use $G_1 \backslash G_2$ to stand for a graph obtained by deleting the arcs of G_2 from G_1; and $G_1 \cup G_2$ for a graph obtained by adding the arcs of G_1 and G_2 together. In addition, $d^-(v)$ and $d^+(v)$ represent v's indegree and v's outdegree, respectively.

Algorithm *graph-stratification*(G)
Begin

```
1.  $V_0$ := all the nodes with no outgoing arcs; $i$ := 0;
2.  $W$ := all the nodes that have at least one child in $V_0$;
3.  **while** $W \neq \emptyset$ **do**
4.  {**for** each node $v$ in $W$ **do**
5.  {let $v_1, ..., v_k$ be $v$'s children appearing in $V_i$;
6.  $C_i(v) := \{v_1, ..., v_k\}$; (*Here, for simplicity, we use $v_j$ to represent a link from $v$ to $v_j$.*)
7.  **if** $d^+(v) > k$ **then** remove $v$ from $W$;
8.  $G := G\backslash\{v \rightarrow v_1, ..., v \rightarrow v_k\}$;
9.  $d^+(v) := d^+(v) - k$;
10. }
11. $V_{i+1} := W$; $i := i + 1$;
12. $W$ := all the nodes that have at least one child in $V_i$;
13. }
```

end

In the above algorithm, we first determine V_0, which contains all those nodes having no outgoing arcs (see line 1). In the subsequent computation, we determine $V_1, ..., V_h$. In this process, G is reduced step by step (see line 8), so is $d^+(v)$ for any $v \in G$ (see line 9). In order to determine V_i ($i > 0$), we will first find all those nodes that have at least one child in V_{i-1}, which are stored in a temporary variable W. For each node v in W (see line 3), we will then check whether it also has some other children not appearing in V_{i-1}, which is done by checking whether $d^+(v) > k$ in line 7, where k is the number of v'children in V_{i-1}. If it is the case, it will be removed from W since it cannot belong to V_i. Concerning the correctness of the algorithm, we have the following proposition.

Proposition 1 Let $G_0 = G$. Denote by G_j the reduced graph after the jth iteration of the out-most **for**-loop. Denote by (v) the outdegree of v in G_j. Then, any node v in G_j does not have children appearing in $V_0 \cup ... \cup V_{j-1}$, where V_0 contains all those nodes having no outgoing arcs, and for any $v \in V_i$ ($i = 1, ..., j - 1$) (v) = 0 while (v) $\neq$ 0, ..., (v) $\neq$ 0.

Proof. We prove the proposition by induction on j.

Basic step. When $j = 1$, the proposition trivially holds.

Induction hypothesis. Assume that when $j = l$, the proposition holds. Then, we have

(1) $G_l = G\backslash(\bigcup_{i=0}^{l-1}\left(\bigcup_{v \in G} C_i(v)\right))$, and

(2) $d_l^+(v) = d^+(v) - \sum_{i=0}^{l-1} |C_i(v)|$.

Now we consider the case of $j = l + 1$. From lines 8 and 9, as well as the induction hypothesis, we immediately get

(3) $G_{l+1} = G \backslash (\bigcup_{i=0}^{l} \left(\bigcup_{v \in G} C_i(v) \right))$, and

(4) $d_{l+1}^+(v) = d^+(v) - \sum_{i=0}^{l} |C_i(v)|$.

From (3) and (4), and also from lines 7 and 11, we can see that for any node $v \in V_{l+1}$ we have $d_{l+1}^+(v) = 0$ while $d_0^+(v) \neq 0, ..., d_l^+(v) \neq 0$. □

From the proof of Proposition 1, we can see that

- to check whether a node v in G_j belongs to V_{j+1}, we need only to check whether $d_l^+(v)$ is strictly larger than $|C_j(v)|$ (see line 7), which requires a constant time; and
- G is correctly stratified.

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

As an example, consider the graph shown in Fig. 4(a). Applying the above algorithm to this graph, we will generate a stratification of the nodes as shown in Fig. 4(b).

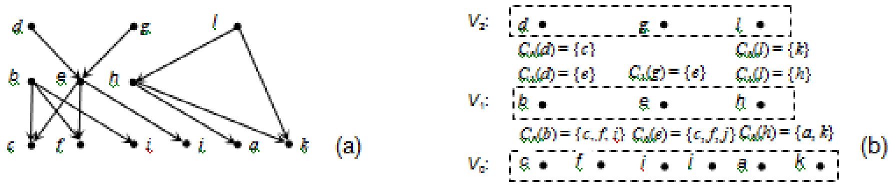


Fig. 4. Illustration for DAG stratification

In Fig. 4(b), the nodes of the DAG shown in Fig. 4(a) are divided into three levels: $V_0 = \{c, f, i, j, a, k\}$, $V_1 = \{b, e, h\}$, and $V_2 = \{d, g, l\}$. Associated with each node at each level is a set of links pointing to its children at different levels. For example, node b in V_1 is associated with three links respectively to nodes c, f, and i in V_0, denoted as $C_0(b) = \{c, f, i\}$. (For simplicity, we use $C_0(b) = \{c, f, i\}$ to represent three links from b to c, f, and i, respectively.)

2.2 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 $B(V, E)$ 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 $B(T, S; E)$. □

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

Definition 3 (*matching* [2]) Let $B(V, E)$ be a bipartite graph. A subset of edges $E' \subseteq E$ 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*, denoted as M_B (or simply M if B is clear from context.) □

Let M be a matching of a bipartite graph $B(T, S; E)$. 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 $E \backslash M$ and M. A path is an *augmenting path* if it is an alternating path with free origin and terminus. Let $v_1 - v_2 - ... - v_k$ be an alternating path with $(v_i, v_{i+1}) \in E \backslash M$ 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 $(v_{i+1}, v_{i+2}) \in E \backslash M$ $(i = 1, 3, ...)$. In addition, we will use $\varphi_M(T)$ and $\varphi_M(S)$ to represent all the free nodes in T and S, respectively; and use $\xi_M(T)$ for all the covered nodes in T and $\xi_M(S)$ for all the covered nodes in S. Finally, if $(u, v) \in M$, we say, u covers v with respect to M, and *vice versa*, denoted as $M(u) = v$, and $M(v) = u$, respectively.

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 [13] and runs in $O(m \cdot \sqrt{n})$ time, where $n = |V|$ and $m = |E|$. The algorithm proposed by Alt, Blum, Melhorn and Paul [1] needs $O(n^{1.5}\sqrt{m/(\log n)})$ time. In the case of large m, the latter is better than the former.

3 Algorithm Descriptions

In this section, we describe our algorithm for the DAG decomposition. The main idea behind it is to construct a series of bipartite graphs for $G(V, E)$ based on the graph stratification and then find a maximum matching for each of such bipartite graphs using the Hopcroft-Karp algorithm [13]. All these matchings make up a set of node-disjoint chains, which, however, may not be minimal. In the following, we first discuss an example to illustrate this idea in 3.1. Then, in 3.2, we define the so-called *virtual nodes*, and show how they can be used to efficiently and effectively reduce the number of node-disjoint chains. Next, in 3.3, we discuss how the virtual nodes can be resolved (removed) from created chains to get the final result.

3.1 An Example

Example 1 Consider the graph and the corresponding stratification shown in Fig. 4. A bipartite graph made up of V_0 and V_1: $B(V_1, V_0; E_1)$ with $E_1 = \boldsymbol{C}_0^1$ is shown in Fig. 5(a) and a possible maximal matching M_1 of it is shown in Fig. 5(b).

Another bipartite graph made up of V_1 and V_2: $B(V_2, V_1; E_2)$ with $E_2 =$ is shown in Fig. 5(c) and a possible maximal matching M_2 of it is shown in Fig. 5(d).

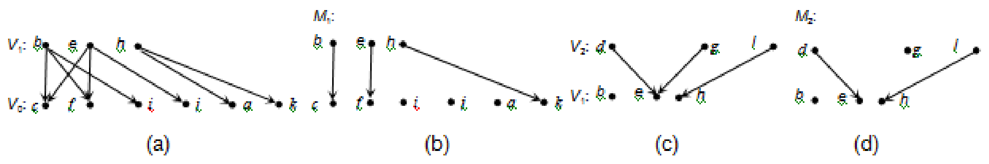


Fig. 5. Maximum matchings for bipartite graphs

Combining M_1 and M_2 by connecting their edges, we will get a set of seven chains, denoted by $M_1 \overline{\cup} M_2$ and shown in Fig. 6(a). (Note that four of these chains each contain only a single node.)

However, if we transfer the edges on an alternating path relative to M_1: $f - e - c - b - i$ (see Fig. 6(b), where a solid edge represents an edge belonging to M_1 while a dashed edge to $E_1 \backslash M_1$); and connect g to f as illustrated in Fig. 6(c), we will get a set of six chains as shown in Fig. 6(d). (Note that g and f are on a chain since there exists a path $g - e - f$, which connects g and f.)

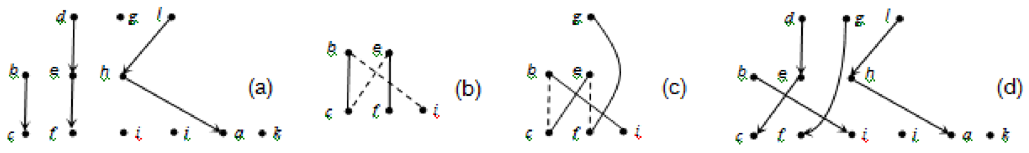


Fig. 6. Illustration for transferring edges on alternating paths

The question is how to efficiently find such a possible transformation to reduce the number of chains.

For this purpose, we introduce the concept of virtual nodes to transfer the reachability information and at the same time to maintain the information on how a transformation can be conducted.

3.2 Chain Generation

From the above example, we can see that by simply combining maximal matchings of bipartite graphs, the number of formed chains may be larger than the minimized number of chains. To solve this problem, we need to introduce some virtual nodes into the original graph, which are used to transfer the reachability information from lower levels to higher levels.

3.2.1 Virtual Nodes

We will work bottom-up. During the process, some virtual nodes may be added to V_i ($i = 1, ..., h - 1$) level by level. However, such virtual nodes will be eventually resolved to obtain the final result.

In the following, we first give a formal definition of virtual nodes. Then, we describe how a virtual node is established. We start our discussion with the following specification:

$V_0' = V_0$.
$V_i' = V_i \cup$ {virtual nodes added to V_i} for $1 \le i \le h - 1$.
$\boldsymbol{C}_i = \boldsymbol{C}_{i-1}^{i} \cup$ {all the new arcs from the nodes in V_i to the virtual nodes added to V_{i-1}'} for $1 \le i \le h - 1$.
$B(V_i, V_{i-1}'; \boldsymbol{C}_i)$ - the bipartite graph containing V_i and V_{i-1}'.
M_i - a maximal matching of $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$.

Definition 4 (*virtual nodes*) Let $G(V, E)$ be a DAG, divided into $V_0, ..., V_h$ (i.e., $V = V_0 \cup ... \cup V_h$). Let M_i be a maximal matching of $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$ for $i = 1, …, h$. For each free node v in V_{i-1}' with respect to M_i, a virtual node v' created for v is a new node added to V_i ($1 \le i \le h - 1$), denoted as $v = s(v')$.

The goal of virtual nodes is to establish the connection between the free nodes (with respect to a certain maximum matching of a bipartite graph) and the nodes that may be several levels apart. Therefore, for each virtual node v' (created for v in V_{i-1}' and added to V_i), a bunch of virtual arcs incident to it should be created. Especially, we distinguish among three kinds of virtual arcs: inherited arcs, transitive arcs and alternating arcs, which are created as follows.

inherited arcs - If there is $u \in V_j$ ($j > i$) such that $u \to v \in E$, add $u \to v'$, referred to as an inherited arc.

transitive arcs - If there exist $u \in V_j$ ($j > i$) and $w \in V_i$ such that $u \to w \in E$ and $w \to v \in \boldsymbol{C}_i$, add $u \to v'$ if it has not been created as an inherited arc, referred to as a transitive arc.

In order to create the *alternating* arcs, we need first to find a maximum set of node-disjoint alternating paths such that

1. Each of them starts at a free node and ends at a covered node in V_{i-1}'; and
2. They cover all the nodes in $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$, except several (possibly 0) free nodes in V_{i-1}' and V_i.

For example, relative to the maximum matching shown in Fig. 5(b) for the bipartite graph shown in Fig. 5(a), we can find three such paths as shown in Fig. 7.

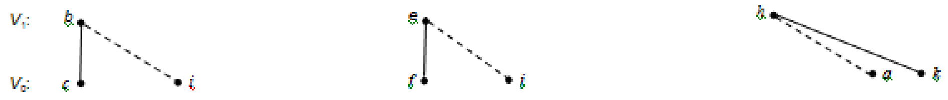


Fig. 7. A maximum set of alternating paths

Denote the path starting at free node v by P_v. The alternating arcs can be described as below.
alternating arc - If there exists w ($\neq v$) on P_v and $u \in V_j$ ($j > i$) 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 has not been created as an inherited or a transitive arc. It is referred to as an alternating arc. We create such an arc to indicate a possibility to make v covered by transferring the edges on the corresponding alternating path from v to w, and then connect u and w (see Fig. 6(b) and (c) for illustration.)

In addition, a virtual arc from v' to $s(v')$ is generated to record the relationship between v' and $s(v')$.

Example 2 Continued with Example 1. Relative to M_1 of $B(V_1, V_0; E_1)$ shown in Fig. 5(b), i, j and k are three free nodes. Then, three virtual nodes i', j' and k' (for i, j and k, respectively) will be created and added to V_1. Then, we have $V_1' = \{b, e, h, i', j', k'\}$. In addition, five virtual arcs: $d \to i'$, $d \to j'$, $g \to i'$, $g \to j'$, and $l \to k'$ will be generated, shown as five dashed arcs in Fig. 8(a).

Among these virtual arcs, $l \to k'$ is an inherited arc since in the original graph we have $l \to k$ (see Fig. 4(a)). But $d \to j'$ and $g \to j'$ are two transitive arcs since j is reachable respectively from d and g through e in V_1 (see Fig. 4(a)).

Finally, $d \to i'$ and $g \to i'$ are two alternating arcs. We join d and i' since there is a node f that is connected to i through an alternating path: $f - e - c - b - i$ (see Fig. 6(b)) and f is reachable from d through a node e in V_1 (see Fig. 4(a).) (We also note that c is another node connected to i through an alternating path: $c - b - i$, and $d \to c \in E$. However, only one alternating arc is created no matter how many possibilities we have to generate such an arc.) For the same reason, we join g and i'.

In Fig. 8(b), we show a possible maximum matching M_2 of $B(V_2, V_1'; \boldsymbol{C}_2)$. Combining M_2 and M_1, we get a set of six chains as shown in Fig. 8(c).

On these chains, the virtual nodes j' and k' can be simply removed since they do not have a parent along the corresponding chains. In order to remove i', however, we have to transfer the edges on the alternating path: $f - e - c - b - i$ and then connect g and f, obtaining the final chains shown in Fig. 6(d). We can also transfer the edges on $c - b - i$ and then connect g and c to get a different set of six chains.

We will call an arc along a chain a *chain arc*. From the above example, we can see that how a virtual node is resolved depends on how it is connected to its parent through a chain arc. Especially, an alternating arc in fact does not represent a reach ability, but indicates a possibility to connect two nodes by transferring edges along some alternating path. Thus, we need to label virtual arcs to represent their properties, and at the same time indicate at what level a virtual node is added. Let v' be a virtual node. Depending on whether its source $s(v')$ is an actual node or a virtual node itself, we label the virtual arcs incident to v' in two different ways.

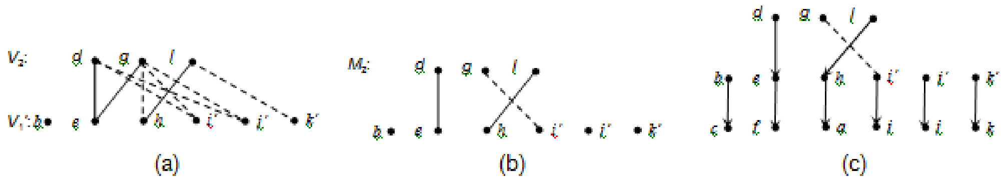


Fig. 8. Illustration for virtual nodes and chains

Assume that $s(v')$ is an actual node in V_{i-1}. Then, v' is a virtual node added to V_i and an virtual arc incident to v': $u \to v'$ with $u \in V_j$ $(j > i)$ will be labeled as follows:

i) If $u \to v'$ is inherited or transitive, its label $label(u \to v')$ will be set to 0, indicating that $s(v')$ is reachable from u (through a path).

ii) If $u \to v'$ is an alternating arc, $label(u \to v')$ will be set to i, indicating that to resolve v' we need to transfer edges along an alternating path in $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$.

If $s(v')$ itself is a virtual node, we need to label $u \to v'$ a little bit differently:

iii) If $u \to v'$ is inherited, the label for it is set to be the same as $label(u \to s(v'))$.

iv) If $u \to v'$ is transitive, there must exist $w_1, \ldots w_k$ $(k \geq 1)$ in V_i such that $w_1 \to s(v'), \ldots, w_k \to s(v') \in \boldsymbol{C}_i$ and $u \to w_1, \ldots, u \to w_k \in E$. We will label $u \to v'$ with min $\{l_1, \ldots, l_k\}$, where $l_j = label(w_j \to s(v'))$ $(j = 1, \ldots, k)$.

v) If $u \to v'$ is an alternating arc, $label(u \to v')$ is set to i (in the same way as (ii)).

In addition, for convenience, all the original arcs in G are considered to be labeled with 0.

In the whole process, we will not only create a set of chains which may contain virtual nodes, but also a new graph by adding virtual nodes and virtual arcs to G, called a *companion graph* of G, denoted as G_c, which will be used for resolution of virtual nodes.

Example 3 Consider the graph shown in Fig. 9(a). This graph can be divided into four levels as shown in Fig. 9(b).

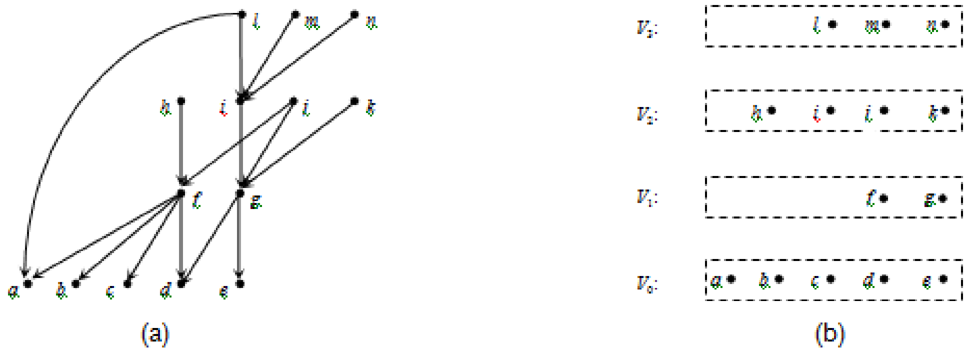


Fig. 9. A DGA and its stratification

In Fig. 10(a), we show the bipartite graph $B(V_1, V_0; \boldsymbol{C}_1)$ made up of the first two levels. A possible maximal matching M_1 of it is shown in Fig. 10 (b). Relative to M_1, a, b and c are three free nodes in V_0. So three virtual nodes a', b' and c' will be created and added to V_1 as shown in Fig. 10(c). At the same time, 13 arcs will be created. Among them, only one arc connects a node in V_3 to one of these virtual nodes while all the remaining 12 arcs connect some nodes in V_2 to them, as described below.

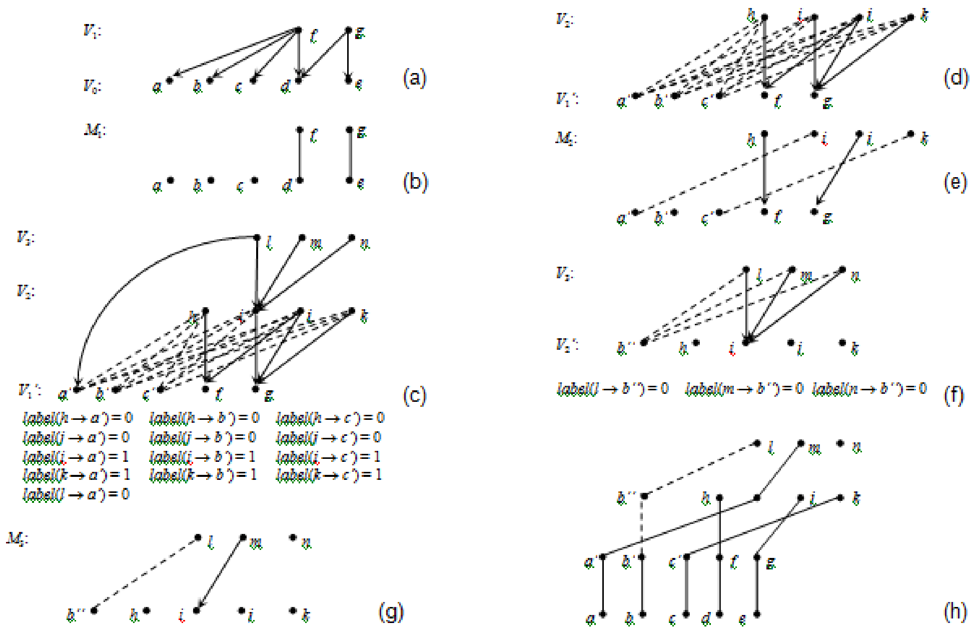


Fig. 10. Illustration for generation of paths

- The only new arc connecting a node l in V_3 to a virtual node a': $l \to a'$ is an inherited arc, labeled with 0 according to (i). (There is neither alternating nor transitive arc connecting any node in V_3 to any of these virtual nodes.)

The 12 new arcs connecting the nodes in V_2 to some virtual nodes can be divided into two groups.

Group 1 contains 6 transitive arcs, labeled with 0 according to (i):

$h \rightarrow a'$, $h \rightarrow b'$ and $h \rightarrow c'$.
$j \rightarrow a'$, $j \rightarrow b'$ and $j \rightarrow c'$.

Group 2 contains 6 alternating arcs, labeled with 1 according to (ii):

$i \rightarrow a'$, $i \rightarrow b'$ and $i \rightarrow c'$.
$k \rightarrow a'$, $k \rightarrow b'$ and $k \rightarrow c'$

Thus, $V_1' = \{a', b', c', f, g\}$. $B(V_2, V_1'; \boldsymbol{C}_2)$ is shown in Fig. 10(d). Assume that the maximal matching M_2 found for it is as shown in Fig. 10(e). Relative to M_2, b' is a free nodes in V_1' and then a virtual nodes b'' will be created and added to V_2 (see Fig. 10(f).) In addition, three transitive arcs: $l \rightarrow b''$, $m \rightarrow b''$ and $n \rightarrow b''$ will be created. All of them are labeled with 1.

$l \rightarrow b''$ is added because b' is reachable from l through i in V_1'. *Label* $(l \rightarrow b'')$ is set to be 1 according to (iv) since $label(i \rightarrow b') = 1$. The same analysis applies to the other two arcs.
$B(V_3, V_2'; \boldsymbol{C}_3)$ is shown in Fig. 9(f). Fig. 10(g) shows a possible maximal matching M_3 of this bipartite graph. Combining M_1, M_2, and M_3, we get $M_1 \uplus M_2 \uplus M_3$. This plus all the free nodes in V_3 make up a set of six chains as shown in Fig. 10(h), and one of them contains only a single node. This must be a minimum set since the original graph contains an antichain of 6 nodes: $\{a, b, c, k, m, n\}$.

In terms of the above discussion, we give the following algorithm.

Algorithm *GenChain*(*stratification of G*)
input: a graph stratification.
output: a set of chains which may contain virtual nodes.

Begin

```
1.   $V_0' := V_0$;
2.   for $i$ = 1 to $h$ do
3.   {  find $M_i$ for $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$
4.   let $v_1, ..., v_k$ be all the free nodes in $V_{i-1}'$, relative to $M_i$;
5.   $V_i' := V_i \cup \{v_1', ..., v_k'\}$; (*$v_1', ..., v_k'$ are the virtual nodes created for $v_1, ..., v_k$, respectively.*)
6.   let $u_1, ..., u_j$ be all the covered virtual nodes in $V_{i-1}'$;
7.   for $l$ = 1 to $j$ do
8.   {  remove all the virtual arcs incident to $u_l$, except the arc belonging to $M_i$; }
9.   for $l$ = 1 to $k$ do
10.  {  create all the virtual arcs incident to $v_l'$; }
11.  }
12.  $M := M_1 \uplus ... \uplus M_h$; return $M$;
```

End

In the above algorithm, special attention should be paid to lines 6 - 8, by which all the virtual arcs incident to a covered virtual node in V_{i-1}' (i.e., a virtual node in V_{i-1}', which is covered relative to M_i), except the corresponding arcs belonging to M_i, will be removed since they will not be used any more in the subsequent computation for the chain generation. Therefore, at any point in time, the number of virtual arcs maintained in the process is bounded by $O(\kappa \cdot n)$.

3.2.2 Computational complexity of chain generation

We now analyze the time complexity of the chain generation.

In general, the cost of this process can be divided into two parts:

- cost$_1$: The time for finding a maximal matching of every $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$ ($i = 1, ..., h$; $V_0' = V_0$); and
- cost$_2$: The time for generating virtual arcs.

We first prove three lemmas to show that for any $i \in \{1, \ldots, h - 1\}$ $|V_i'| \le \kappa$.

Let M be a maximal matching of a bipartite graph $B(V_1, V_0; E)$. Let Y_i ($i = 0, 1$) be a subset of V_i. We denote by $M(Y_i)$ ($i = 0, 1$) a subset of $V_{(i+1)\text{mod}2}$ such that for each $v \in M(Y_i)$ there exists a node u in Y_i with $(u, v) \in M$. We further divide M into three (possibly empty) groups: $M[1]$, $M[2]$, and $M[3] = M\backslash(M[1] \cup M[2])$ such that

- for each $v \in \xi_{M[1](}(V_1)$, there exists at least one alternating path connecting a free node in $\varphi_M(V_0)$ to v; (remember that $\varphi_M(V_i)$ stands for all the free nodes in V_i relative to M while $\xi_M(V_i)$ for all the covered nodes in V_i by M. So $\xi_{M[1](}(V_i)$ represents all those nodes in V_i covered by $M[1]$.)
- for each $v \in \xi_{M[2]}(V_0)$, there exists at least one alternating path connecting a free node in $\varphi_M(V_1)$ to v;
- for each $v \in \xi_{M[3]}(V_0)$, there exists no alternating path connecting it to any node in $\varphi_M(V_1) \cup \xi_{M[2]}(V_1)$; and for each $u \in \xi_{M[3]}(V_1)$, there exists no alternating path connecting it to any node in $\varphi_M(V_0) \cup \xi_{M[1]}(V_0)$.

Concerning this partition of M, we have the following lemma.

Lemma 1 Any node in $\xi_{M[1]}(V_0)$ does not connect to any node in $\varphi_M(V_1) \cup \xi_{M[2]}(V_1)$ through an alternating path relative to M. Also, any node in $\xi_{M[2]}(V_1)$ does not connect to any node in $\varphi_M(V_0) \cup \xi_{M[1]}(V_0)$ through an alternating path relative to M.

Proof. Let v be a node in $\xi_{M[1]}(V_0)$. Assume that there is an alternating path P connecting v to a free node u in $\varphi_M(V_1)$. Then, the path from a free node w in $\varphi_M(V_0)$ to $M(v)$, the edge $(M(v), v)$, and P together make up an augmenting path connecting w and u, contradicting the fact that M is a maximal matching. Therefore, any node in $\xi_{M[1]}(V_0)$ does not connect to any node in $\varphi_M(V_1)$. In the same way, we can prove the rest part of the lemma. □

From this lemma, the following lemma can be immediately derived, by which we show how to find an antichain for bipartite graph.

Lemma 2 Let M be a maximal matching of a bipartite graph $B(V_1, V_0; E)$. Then, $\varphi_M(V_0) \cup M \cup \varphi_M(V_1)$ make up a minimized set of chains of $B(V_1, V_0; E)$; and $\varphi_M(V_0) \cup \varphi_M(V_1) \cup \xi_{M[1]}(V_0) \cup \xi_{M[2]}(V_1) \cup \xi_{M[3]}(V_0)$ is one of its antichains.

Proof. It is easy to see that $\varphi_M(V_0) \cup M \cup \varphi_M(V_1)$ is a minimized set of chains of $B(V_1, V_0; E)$. It is also easy to see that any node in $\xi_{M[3]}(V_0)$ is not reachable from any node in $\varphi_M(V_1) \cup \xi_{M[2]}(V_1)$. Then, from Lemma 1, we can see that any node in $\xi_{M[1]}(V_0)$ is not reachable from $\varphi_M(V_1) \cup \xi_{M[2]}(V_1)$, and thus any pair of nodes in $A = \varphi_M(V_0) \cup \varphi_M(V_1) \cup \xi_{M[1]}(V_0) \cup \xi_{M[2]}(V_1) \cup \xi_{M[3]}(V_0)$ are not reachable from each other. Therefore, A is an antichain. In addition, we have $|\varphi_M(V_0) \cup \mathrm{M} \cup \varphi_M(V_1))| = |A|$. So A is a maximum antichain. □

Lemma 3 Let $G(V, E)$ be a DAG, divided into V_0, ..., V_h (i.e., $V = V_0 \cup ... \cup V_h$). Let M_i be a maxima matching of the bipartite graph $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$. $V_i' = V_i \cup$ {virtual nodes added to V_i} for $1 \le i \le h - 1$. Then, for any $i \in \{1, \ldots, h - 1\}$, we have $|V_i'| \le \kappa$.

Proof. First, we notice that $|V_1'| = |V_1| + |\varphi_{M_1}(V_0)| = |\varphi_{M_2}(V_1) \cup \varphi_{M_1}(V_1) \cup \xi_{M_1[1]}(V_0) \cup \xi_{M_1[2]}(V_1) \cup \xi_{M_1[3]}(V_0)| \leq \kappa$. Then, we analyze the size of V_2'. Obviously, $|V_2'| = |V_2| + |\varphi_{M_2}(V_1')| = |A|$, where $A = \varphi_{M_2}(V_1') \cup \varphi_{M_2}(V_2) \cup \xi_{M_2[1]}(V_1') \cup \xi_{M_2[2]}(V_2) \cup \xi_{M_2[3]}(V_1')$. According to Lemma 2, A is an antichain of $G(V_2, V_1'; \boldsymbol{C}_2)$. Let $R = \varphi_{M_2}(V_1') \cup \xi_{M_2[1]}(V_1') \cup \xi_{M_2[3]}(V_1')$. We will distinguish between two cases:

1. R does not contain virtual nodes. Then, A is a set such that any two nods in it are not connected through a path. Thus, $|A| \leq \kappa$.
2. R contains at least a virtual node. In this case, we will replace each virtual node v in R with $s(v)$ (a free node in V_0) and each of those nodes u in R with $M_1(u)$ (a node in V_0 such that $(v, u) \in M_1$) which belong to $\xi_{M_1[1]}(V_1)$ and connected to a free node of V_0 relative to M_1 (through an alternating path), resulting in a new set R' with the following properties:

   i) Any two nodes in R' are not connected through a path according to Lemma 1.

   ii) Any node w in R' is not connected to a node in $D = \varphi_{M_2}(V_2) \cup \xi_{M_2[2]}(V_2)$. Otherwise, if w is a free node of V_0 relative to M_1 in R', then its virtual node must be connected to that node in D. Contradiction. If w is a node in V_0 such that $w = M_1(u)$ for some $u \in \xi_{M_1[1]}(V_1)$, there must be a free node of V_0 relative to M_1 in R', which is connected to a node in D. Contradiction.

Let $R'' = D \cup R'$. It can be seen that R'' is a set in which any two nodes are connected through a path. Therefore, $|A| = |R''| \leq \kappa$.

Repeating the above argument to all V_i' with $i \geq 3$, we can prove the lemma. □

Based on Lemma 3, the time complexity of chain generation can be easily estimated.

First, using the Hopcroft and Karp algorithm [13], the time for finding a maximal matching of $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$ is bounded by

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

Therefore, cost$_1$ is bounded by

$$O(\sum_{i=1}^{k} (\sqrt{|V_i| + |V_{i-1}'|} \cdot |C_i|))$$
$$\leq O(\sqrt{k} \sum_{i=1}^{h} k|V_i|) = O(\sqrt{\kappa} \cdot \kappa \cdot n)$$

For estimating cost$_2$, we need to compute the costs for creating all the inherited, transitive and alternating arcs. First, for a virtual node, the cost for generating inherited arcs is a constant since we can simply promote the corresponding free node from its level to the level above it and handle it as virtual. (For example, to create a virtual node for a node v at level i, we can add v to level $i + 1$. Then create a node containing only a link to v and leave it at level i, used as a representative of v.)

Secondly, the cost for creating the transitive arcs for all the virtual nodes added to V_i is obviously bounded by $O(|V_{i-1}'| \cdot |V_i| \cdot n)$. It is because at most $|V_{i-1}'|$ virtual nodes can be added to V_i and the number of all the transitive arcs incident to each of these virtual nodes is bounded by $O(|V_i| \cdot n)$.

So the total cost for creating the transitive arcs is bounded by

$$\sum_{i=1}^{h} |V_{i-1}'| \cdot |V_i| \cdot n \leq \kappa \cdot n \cdot \sum_{i=1}^{k} |V_i| = \mathrm{O}(\kappa \cdot n^2).$$

Finally, we notice that for generating the alternating arcs incident to the virtual nodes added to V_i, we need to search $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$ once for each of them, and the number of the arcs in $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$ is bounded by $\mathrm{O}(|V_{i-1}'|\cdot|V_i|) \leq \mathrm{O}(\kappa\cdot|V_i|)$. In addition, for each free node v (in V_{i-1}'), the number of all those nodes z (in V_{i-1}') which are connected to v through an alternating path is bounded by $|V_i|$ since each of such nodes must be covered relative to M_i. For each z, we need to search at most $|V_i|$ arcs to find all those w in V_i such that $w \to z \in \boldsymbol{C}_i$. Moreover, for each w, we have to further find all those nodes u such that $u \to w \in E$. So the cost for generating all the alternating arcs incident to v is bounded by $\mathrm{O}(|V_{i-1}'|\cdot|V_i| + |V_i|^2 + |V_i|\cdot n)$. Therefore, the total cost for creating all the alternating arcs is bounded by

$$\kappa \cdot \sum_{i=1}^{h} |V_{i-1}'| \cdot |V_i| + |V_i|^2 + |V_i| \cdot n = \mathrm{O}(\kappa^2 \cdot n + \kappa \cdot n^2).$$

In terms of the above analysis, we have the following proposition.

Proposition 2 The time for creating a minimized set of disjoint chains, which may contain virtual nodes, is bounded by $\mathrm{O}(\kappa^2\cdot n + \kappa\cdot n^2)$.
The analysis of the space requirement of this process is quite simple. As mentioned in 3.2.1, at any time point, we maintain at most $\mathrm{O}(\kappa\cdot n)$ virtual arcs. Besides this, all the bipartite graphs: $B(V_1, V_0; \boldsymbol{C}_1)$, $B(V_2, V_1'; \boldsymbol{C}_2)$, ..., $B(V_h, V_{h-1}'; \boldsymbol{C}_h)$ have to be maintained.
Thus, the space overhead is bounded by

$$\mathrm{O}(\kappa\cdot n + \sum_{i=1}^{h} |V_{i-1}'| \cdot |V_i|) = \mathrm{O}(\kappa\cdot n).$$

3.3 Virtual Node Resolution

After the chain generation, the next step is to resolve virtual nodes on chains. In the following, we will first discuss the working process to remove virtual nodes in 3.3.1. Then, in 3.3.2, we analyze its computational complexities.

3.3.1 Removing virtual nodes

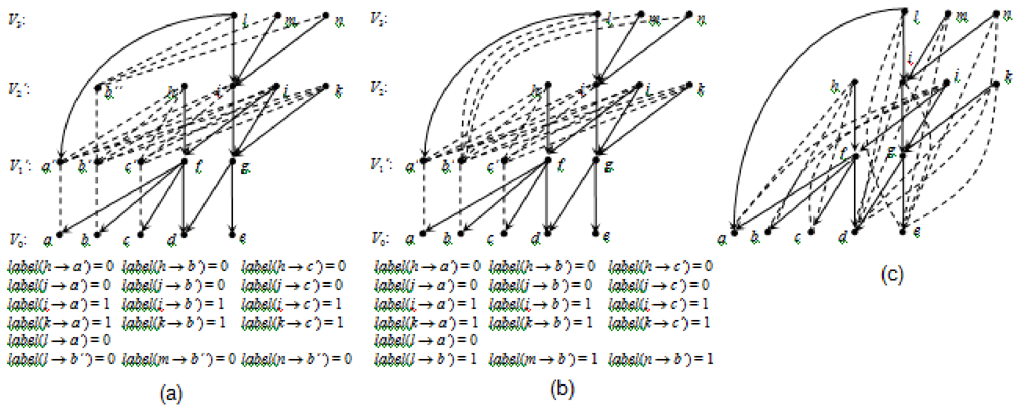
To remove virtual nodes from chains, we will work with the companion graph G_c. Two steps will be carried out:

1. Connect some nodes according to the connectivity represented by the virtual nodes and then remove them.
2. Establish new connections between free nodes by transferring edges along alternating within a bipartite graph or cross more than one bipartite graph.

In the first step, we will check G_c top-down and remove virtual nodes level by level. Let v be a virtual node in V_i'. Let $u_1, \ldots, u_k$ be its parents in G_c. Wi will distinguish between two cases:

i) $s(v)$ is an actual node. If $label(u_j \to v) = 0$ ($j = 1, \ldots, k$), connect u_j to $s(v)$. Otherwise, $label(u_j \to v)$ must be equal to i, and we will connect u_j to each node w which is connected to $s(v)$ through an alternating path in $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$ and w is reachable from u_j.
ii) $s(v)$ itself is a virtual node. If $label(u_j \to v) > i$, connect u_j to $s(v)$ and $label(u_j \to s(v))$ is set to be the same as $label(u_j \to v)$. Otherwise, $label(u_j \to v) = i$. In this case, we will connect u_j to some nodes in in the same way as (i).

See Fig. 11 for illustration



V_3:
V_2':
V_1':
V_0:
$label(h \to a') = 0$ $label(h \to b') = 0$ $label(h \to c') = 0$
$label(j \to a') = 0$ $label(j \to b') = 0$ $label(j \to c') = 0$
$label(i \to a') = 1$ $label(i \to b') = 1$ $label(i \to c') = 1$
$label(k \to a') = 1$ $label(k \to b') = 1$ $label(k \to c') = 1$
$label(l \to a') = 0$
$label(l \to b'') = 0$ $label(m \to b'') = 0$ $label(n \to b'') = 0$
(a)
$label(h \to a') = 0$ $label(h \to b') = 0$ $label(h \to c') = 0$
$label(j \to a') = 0$ $label(j \to b') = 0$ $label(j \to c') = 0$
$label(i \to a') = 1$ $label(i \to b') = 1$ $label(i \to c') = 1$
$label(k \to a') = 1$ $label(k \to b') = 1$ $label(k \to c') = 1$
$label(l \to a') = 0$
$label(l \to b') = 1$ $label(m \to b') = 1$ $label(n \to b') = 1$
(b)
(c)

Fig. 11. Illustration for virtual node resolution

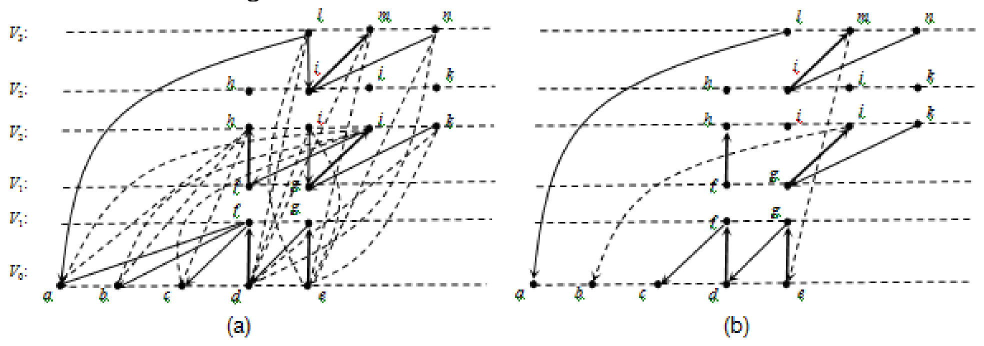


Fig. 12. Illustration for transferring edges along alternating paths

Let v be a virtual node and u be its parent along a chain. We call $u \to v$ is a chain arc. To remove all the virtual nodes, we will go through three steps.

In the first step, all those virtual nodes that do not have a parent along a chain we will be simply eliminated.

In the second step, we will check all those chain arcs with $label(u \to v) < level(v)$. For such an arc, we will directly connect u to $s(v)$ and then remove v. $label(u \to s(v))$ is set to be the same as $label(u \to v)$. If $label(u \to s(v)) < level(s(v))$, we will connect u to $s^2(v) = s(s(v))$ and remove $s(v)$. We repeat this operation until we meet $s^i(v)$ for some i such that $label(u \to s^i(v)) = level(s^i(v))$.

After these two steps, we have only those virtual nodes v with $label(u \to v) = level(v)$ left on the chains, where u is the parent of v along the corresponding chain. Then, in the third step, we will remove all such virtual nodes by using alternating paths.

For this purpose, we first construct a combined graph, denoted as $\Im_G$, over all the bipartite graphs of G, based on G_c. We notice that in G_c all the generated virtual arcs are maintained.

1. A node in V_i ($i = 1, \ldots h - 1$) will appear two times: one is in $\vec{B}$ (V_{i+1}, V_i'; $\mathbf{C}_{i+1}$), and one is in $\vec{B}$ (V_i, V_{i-1}'; $\mathbf{C}_i$), but they are considered to be different nodes.

2. For each ending node of a chain $v \in V_i$ in $\vec{B}$ (V_i, V_{i-1}'; $\boldsymbol{C}_i$) (i = 2, …, h), we will search G starting from each node $u \in V_i$ which is connected to v through an alternating path in $\vec{B}$ (V_i, V_{i-1}'; $\boldsymbol{C}_i$), and connect u to each reachable node (in this way, part of the transitive closure is established.)

3. Remove all the remaining virtual nodes level by level. Let v be a virtual node in V_i' (in $\vec{B}$ (V_{i+1}, V_i'; $\boldsymbol{C}_{i+1}$)), connect each of its parent u to some nodes w in V_{i-1}' (in $\vec{B}$ (V_i, V_{i-1}'; $\boldsymbol{C}_i$)) as below.

   i) If $label(u \to v) = i$, w should be a node connected to $s(v)$ through an alternating path. W is actual node, and $label(u \to w)$ is implicitly set to be 0.

   ii) If $label(u \to v) < i$, $w = s(v)$ and $label(u \to w)$ is set to be the same as $label(u \to v)$.

In Fig. 12, we show the construction of a combined graph over the bipartite graphs shown in Fig. 11.

Next, we need to define an important concept, the so-called alternating graphs.

Definition 5 (*alternating graph*) Let $B(T, S; E)$ be a bipartite graph. Let M be a matching of $B(T, S; E)$. The alternating graph $\vec{B}$ with respect to M is a directed graph with the following sets of nodes and arcs:

$$\vec{V} = V(\vec{B}) = T \cup S, \text{ and}$$

$$\vec{E} = E(\vec{B}) = \{u \to v \mid u \in S, v \in T, \text{ and } (u, v) \in M\} \cup \{v \to u \mid u \in S, v \in T, \text{ and } (u, v) \in E \backslash M\}. \ \square$$

By this definition, an alternating graph of a bipartite graph is almost the same as the original bipartite graph, but only with the covered and uncovered edges being set with reverse directions.

Now what we need to do is to find a maximum set of node-disjoint paths in the alternating version of the combined graph, each starting from an end node of a chain or a node which is a parent of some virtual node along a chain, and ending at a starting node of a chain.

This must be a minimum set since the original graph contains an antichain of size 6: {a, b, c, k, m, n}. □

It remains to show how to find a maximal set of node-disjoint paths in $\Im_G$. For this purpose, we define a maximum flow problem over $\Im_G$ (with multiple sources and sinks) as follows:

- Each free node $v \in V_i$ in $\vec{B}$ (V_i, V_{i-1}; $\vec{E}_i$) (with respect to M_i, i = 2, …, h) is designated as a *source*. Each free node $u \in V_{j-1}$ in $\vec{B}$ (V_j, V_{j-1}; $\vec{E}_j$) (with respect to M_j, j = 1, …, h -1) is designated as a *sink*.
- Each arc $u \to v$ is associated with a capacity $c(u, v) = 1$. (If nodes u, v are not connected, $c(u, v)$ is considered to be 0.)

It is a typical 0-1 network. Finding a maximum flow corresponds to finding a maximum set of node-disjoint paths.

See Fig. 13(a), (b) and (c) for illustration. In Fig. 13(d), we show the final result obtained by transferring edges on the alternating paths.

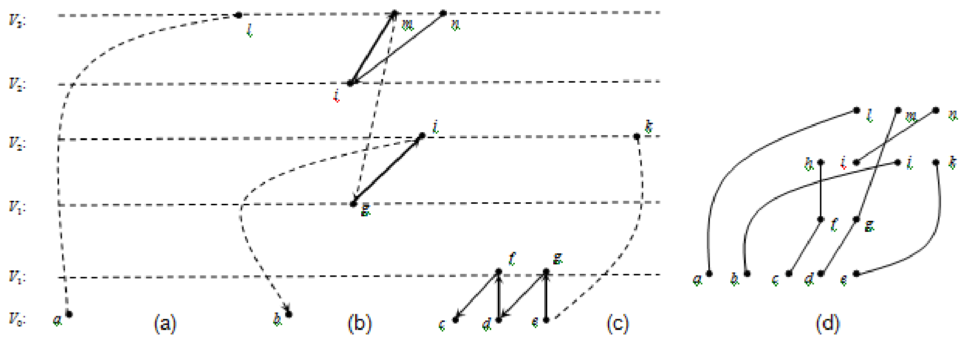


Fig. 14. A set of node-disjoint paths and the final result

3.3.2 Computational complexities of virtual node resolution

In this subsection, we analyze the time complexity of the virtual node resolution. The dominant cost of this process consists of two parts:

- cost$_3$: the time for constructing combined graphs $\Im_G$; and
- cost$_4$: the time for finding a maximum set of bode-disjoint paths in $\bar{\Im}_G$.

Let $u_1, \ldots, u_k$ ($k < \kappa$) be all the end nodes of chains appearing in V_2 or higher. We will estimate the number n_i of the nodes connected to each u_i ($i = 1, \ldots, k$) through an alternating path within the corresponding bipartite graph.

To this end, we consider the transitive closure $G^*(V, E^*)$ of G. We make a bipartite graph B_G as follows.

1. For each $v \in V$, we produce two nodes x_v and y_v.
2. For any two nodes $u, v \in V$, we will connect x_u and y_v if $u \to v \in E^*$.

Let M be a maximum matching of B_G. We have $|M| \leq n$. Notice that the chains generated by Algorithm *GenChain*(*stratification of G*) corresponds a matching M' of B_G. Since on the chains almost each node is connected to another node and only some nodes are connected to virtual nodes, it is reasonable to assume that $|M'| = \alpha\cdot|M|$, where α is a constant with $0 < \alpha \leq 1$. The following proposition can be found in [2].

Proposition 3 $M \Delta M'$ contains at least $|M|$ - $|M'|$ node-disjoint augmenting paths relative to M', where $M \Delta M' = (M \cup M')\backslash(M \cap M')$, referred to as the symmetric difference of M and M'. □

From this proposition, the following lemma can be easily derived.

Lemma 4 The average length of an augmenting path relative to M' is $2\lfloor |M'|/(|M|-|M'|)\rfloor + 1$. □

We have Lemma 4 since on average each augmenting path contains $\lfloor |M'|/(|M|-|M'|)\rfloor$ edges from M', and so $2\lfloor |M'|/(|M|-|M'|)\rfloor + 1$ edges altogether.

Since $|M'| = \alpha\cdot|M|$, we have

$$n_i = 2\lfloor |M'|/(|M|-|M'|)\rfloor + 1 = 2\lfloor \alpha/(1-\alpha)\rfloor + 1.$$

In addition, each v_i (i = 1, … k) can be at most on one of such node-disjoint augmenting paths, and k is bounded by κ. Therefore, the average value of cost_3 is bounded by

$$\text{O}(\sum_{i=1}^{k} n_i \cdot |E|) = \text{O}(\kappa \cdot m).$$

To estimate cost_4, we need first to establish a lemma.

Lemma 5 For each arc e in $\Im_G$ corresponding to an edge in some maximum matching found for a bipartite graph, if is followed by another arc e', then e' must be an arc which corresponds to an edge not belonging to any maximum matching. Similarly, any arc corresponding to an edge not covered by any maximum matching must be followed by an arc corresponding to a covered edge if any.

Proof. Obviously, for any arc corresponding to an edge within a bipartite graph, the lemma holds. For any arc which crosses bipartite graphs, it always goes from some V_i in $\vec{B}(V_i, V_{i-1}; \vec{E}_i)$ (i = 2, …, h) to a node in some V_j in $\vec{B}(V_{j+1}, V_j; \vec{E}_{j+1})$ with $j < i - 1$. So it cannot be an arc corresponding to a covered edge, but must be followed by an arc corresponding to a covered edge. □

From this lemma, the following corollary can be immediately derived.
Corollary 1 For each node v in $\Im_G$, either there is only one arc emanating from it or only one arc entering it. □

According to Corollary 1, we know that by using Dinic's algorithm [9] the time required to find a maximum flow and then a maximum set of node-disjoint paths is bounded by $\text{O}(|V(\Im_G)|^{0.5} \cdot |E(\Im_G)|)$ (see pp. 119 – 121 in [2] for a detailed analysis.) For self-explanation, we also give a modified version of Dinic's algorithm adapted to our problem in Appendix.

Proposition 4 The time complexity of the whole process to resolve virtual nodes is bounded by $\text{O}(\kappa \cdot n^2 + \kappa \cdot m)$.

4 Correctness

In this subsection, we prove the correctness of our algorithm.

First, for any graph which can be divided into two levels the algorithm obviously produce a correct answer.

For any graph which can divided into three levels, the result created by the algorithm is also correct, as stated in the following lemma.

Lemma 6 For any three-level graph $G(V, E)$, the number of chains generated by the algorithm is minimum.

Proof. Let $V = V_0 \cup V_1 \cup V_2$. If no virtual node is created, the lemma obviously holds. So we assume that some virtual nodes for the free nodes in V_0 relative to M_1. In this case, all the virtual nodes v will be removed by constructing $\Im_G$, in which each node on an alternating path starting from a free node in V_2 relative to M_2 will be connected to all reachable nodes in V_0. Also, v's parent (in V_2) will be connected to all the reachable nodes in V_0. By transferring edges on each path of a maximum set of node-disjoint paths each starting from a parent of a virtual node or a free node in V_2, and terminating at a free node in V_1 or V_0, we will get the result. It must be a minimum set of chains since the number of the free nodes which become covered is maximized. □

Proposition 5 The number of the chains generated for a DGA by our algorithm is minimum.

Proof. We prove the proposition by induction on h.

Initial step. When $h = 1, 2$, the proposition holds according to Lemma 6.

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

1. Stratify G, dividing V into $V_0, ..., V_h$ (i.e., $V = V_0 \cup ... \cup V_h$).

2. Find a maximum matching M_1 of $G(V_1, V_0; \boldsymbol{C}_1)$. Construct virtual nodes for all the free nodes in V_0, and add them into V_1. Add all the virtual arcs as described in 3.2. Then, remove V_0.

So G' is of height k. According to the induction hypothesis, a minimal set Q of disjoint chains can be found.

Let $u_1, \dots, u_k$ ($k < \kappa$) be all the end nodes of chains appearing in V_2 or higher. We connect each node on an alternating path starting from a u_i ($1 \le i \le k$) to a reachable node in V_0; and then connect the parents of the virtual nodes to all the respective reachable nodes in V_0. Removing the virtual nodes. In a next step, we will find a maximum set of node-disjoint paths each starting from a u_i or a parent of some virtual node along a chain, and ending at a free node in V_0. Assume that there are still l free nodes not on any of such node-disjoint paths. Thus, Q and these l free nodes make up a set of chain. It must be minimum since Q is minimum and l is also minimum. It should be the same as the result by applying the algorithm to G up to the edge transferring, as illustrated in Fig. 14. □

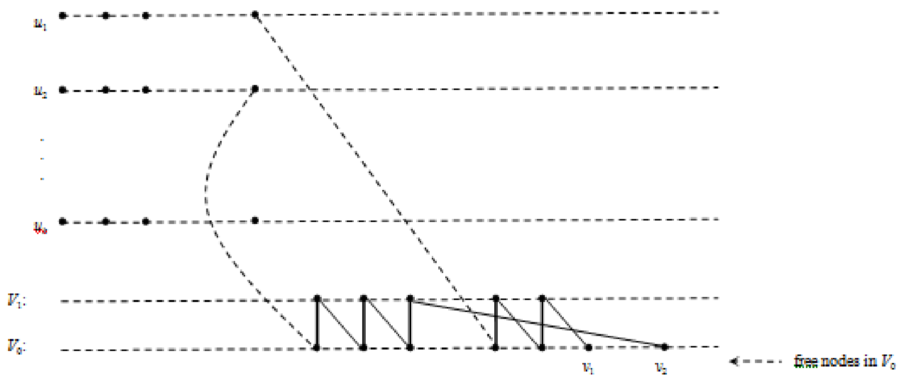


Fig. 14. A set of node-disjoint paths; each of them is an alternating path

5 Conclusion

In this paper, a new algorithm for finding a minimal decomposition of DAGs is proposed. The algorithm needs $O(\kappa \cdot n^2)$ time and $O(\kappa \cdot n + m)$ space, where n and m are the number of the nodes and the arcs in a DAG G, respectively; and κ is the width of G. The main idea of the algorithm is the concept of virtual nodes and the DAG stratification that generates a series of bipartite graphs which may contain virtual nodes. By executing Hopcropt-Karp's algorithm, we find a maximum matching for each of such bipartite graphs, which make up a set of node-disjoint chains. A next step is needed to resolve all the virtual nodes appearing on the chains to get the final result.

We also point out that our algorithm can be easily modified to a 0-1 network flow algorithm by defining a chain to be a path and accordingly changing the conditions for creating transitive and alternating arcs.

Competing Interests

Authors have declared that no competing interests exist.

References

[1] H. Alt, N. Blum, K. Mehlhorn, and M. Paul, Computing a maximum cardinality matching in a bipartite graph in time O(), *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] C. Chekuri and M. Bender, An Efficient Approximation Algorithm for Minimizing Makespan on Uniformly Related 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] Y. Chen and Y.B. Chen, On the Decomposition of Posets, in *Proc. 2nd Int. Conf. on Computer Science and Service System (CSSS 2012)*, IEEE, Aug. 11-13, Nanjing, China, pp. 1115 - 1119.

[6] Y. Chen and Y.B. Chen, On the Graph Decomposition, *Int. Conf. on Big Data and Cloud Computing,* IEEE, Dec. 3 - 5, 2014, Sydney, Australia, pp. 777-784.

[7] 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.

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

[9] 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.

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

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

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

[13] 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.

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

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

[18] 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.

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

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

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

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

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

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

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

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

APPENDIX

Find node-disjoint paths in Combined graphs

A.1 Algorithm

In the appendix, we discuss an algorithm for finding a maximal set of node-disjoint paths in a combined graph $\Im_G$. Its time complexity is bounded by O ($e\cdot\sqrt{n}$), where $n = V(\Im_G)$ and $e = E(\Im_G)$. It is in fact a modified version of Dinic's algorithm [6], adapted to combined graphs, in which each path from a virtual node to a free node relative to *a* maximum matching M_i for some bipartite graph is an alternating path, and for each edge $(u, v) \in M_i$, we have $d^+(u) = d^-(v) = 1$. Therefore, for any three nodes v, v', and v'' on a path in $\Im_G$, we have $d^+(v) = d^-(v') = 1$, or $d^+(v') = d^-(v'') = 1$. We call this property the *alternating property*, which enables us to do the task efficiently by using a dynamical arc-marking mechanism. An arc $u \to v$ with $d^+(u) = d^-(v) = 1$ is called a *bridge*.

Our algorithm works in multiple phases. In each phase, the arcs in $\Im_G$ will be marked or unmarked. We also call a virtual node in $\Im_G$ an origin and a free node a terminus. An origin is said to be saturated if one of its outgoing arcs is marked; and a terminus is saturated if one of its incoming arcs is marked.

In the following discussion, we denote $\Im_G$ by A.

At the very beginning of the first phase, all the arcs in A are unmarked.

In the kth phase ($k \geq 1$), a subgraph $A^{(k)}$ of A will be explored, which is defined as follows.

Let V_0 be the set of all the unsaturated origins. Define V_j ($j > 0$) (note that not to confuse this with the graph stratification) as follows:

$$E_{j-1} = \{u \to v \in E(A) \mid u \in V_{j-1}, v \notin V_0 \cup V_1 \cup ... \cup V_{j-1}, u \to v \text{ is unmarked}\} \cup$$
$$v \to u \in E(A) \mid u \in V_{j-1}, v \notin V_0 \cup V_1 \cup ... \cup V_{j-1}, v \to u \text{ is marked}\},$$

$V_j =$ $\{v \in V(A) \mid$ for some u, $u \to v$ is unmarked and $u \to v \in E_{j-1}\} \cup$
$v \in V(A) \mid$ for some u, $v \to u$ is marked and $v \to u \in E_{j-1}\}$.

Define $j^* = \min\{j \mid V_j \cap \{\text{unsaturated terminus}\} \neq \Phi\}$.

$A^{(k)}$ is formed with $V(A^{(k)})$ and $E(A^{(k)})$ defined below.

If $j^* = 1$, then
$V(A^{(k)}) = V_0 \cup (V_j^* \cap \{\text{unsaturated terminus}\})$,
$E(A^{(k)}) = \{u \to v \mid u \in V_{j^*-1}$, and $v \in \{\text{unsaturated terminus}\}\}$.

If $j^* > 1$, then

$V(A^{(k)}) = V_0 \cup V_1 \cup ... \cup V_{j^*-1} \cup (V_{j^*} \cap \{\text{unsaturated terminus}\}\}$,
$E(A^{(k)}) = E_0 \cup E_1 \cup ... \cup E_{j^*-2} \cup \{u \to v \mid u \in E_{j^*-1}$, and $v \in \{\text{unsaturated terminus}\}\}$.

The sets V_j are called *levels*.

In $A^{(k)}$, a node sequence $v_1, ..., v_j, v_{j+1}, ..., v_l$ is called a complete sequence if the following conditions are satisfied.

(1) v_1 is an origin and v_l is a terminus.
(2) For each two consecutive nodes v_j, v_{j+1} ($j = 1, ..., l - 1$), we have an unmaked arc $v_j \to v_{j+1}$ in $A^{(k)}$, or a marked arc $v_{j+1} \to v_j$ in $A^{(k)}$.

Our algorithm will explore $A^{(k)}$ to find a set of node-disjoint complete sequences (i.e., no two of them share any nodes.) Then, we mark and unmark the arcs along each complete sequence as follows.

(i) If (v_j, v_{j+1}) corresponds to an arc in $A^{(k)}$, mark that arc.
(ii) If (v_{j+1}, v_j) corresponds to an arc in $A^{(k)}$, unmark that arc.

Obviously, if for an $A^{(k)}$ there exists j such that $V_j = \Phi$ and $V_i \cap \{\text{unsaturated terminus}\} = \Phi$ for $i < j$, we cannot find a complete sequence in it. In this case, we set $A^{(k)}$ to Φ and then the kth phase is the last phase.

Example 4 Consider a graph A shown in Fig. 15(a), in which nodes a and b are two origins; and nodes g and h are two terminus. Initially, all the arcs are not marked. Thus, $V_0 = \{a, b\}$, $V_1 = \{c, d\}$, $V_2 = \{e, f\}$, $V_3 = \{g, h\}$; and $j^* = 3$. $A^{(1)}$ is the same as A. (Normally, $A^{(1)}$ has fewer nodes than A.)

Assume that by exploring $A^{(1)}$ (using the algorithm given below), we find a complete sequence: a, d, f, g. Then, we will mark three arcs (a, d), (d, f), and (f, g), as shown by the thick arrows in Fig. 15(b). With respect to these marked arcs, a second subgraph $A^{(2)}$ (in the second phase) will be constructed as shown in Fig. 15(c). In this phase, $V_0 = \{b\}$ (since node a is saturated), $V_1 = \{d\}$, $V_2 = \{a\}$ (note that $a \to d$ is marked), $V_3 = \{c\}$, $V_4 = \{e\}$, $V_5 = \{g\}$, $V_6 = \{f\}$ (note that $f \to g$ is marked), $V_7 = \{h\}$; and $j^* = 7$. By exploring $A^{(2)}$, we will find another complete sequence: b, d, a, c, e, g, f, h, on which all the unmarked arcs will be marked while all the marked arcs will be unmarked, as demonstrated in Fig. 15(d). Fig. 16(e) shows all the marked arcs, which make up two node-disjoint paths: $a \to c \to e \to g$ and $b \to d \to f \to h$.

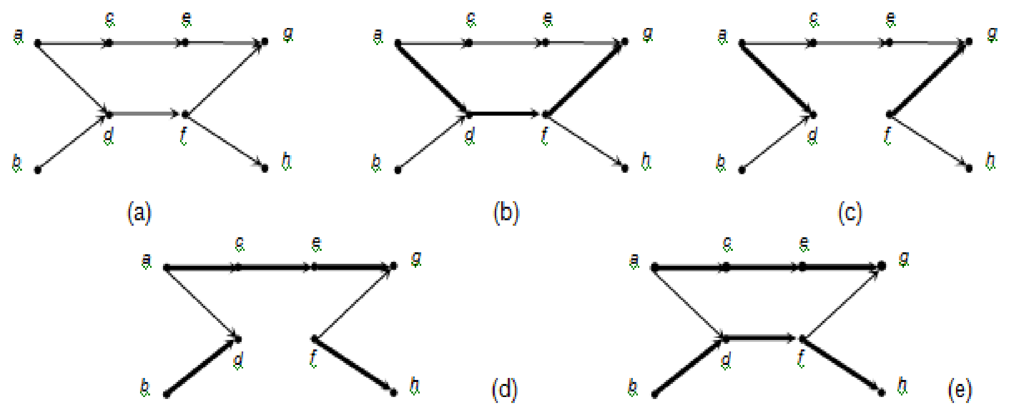


Fig. 15. Illustration for graph searching

The following algorithm is devised to explore an $A^{(k)}$, in which a stack ***H*** is used to store complete sequences. In addition, for each v in $A^{(k)}$, *neighbor*(v) represents a set of nodes: $v_1, .., v_m$ such that for each $j \in \{1, ..., m\}$ $v \rightarrow v_j \in E(A^{(k)})$ if $v \rightarrow v_j$ is unmarked, or $v_j \rightarrow v \in E(A^{(k)})$ if $v_j \rightarrow v$ is marked.

Algorithm *subgraph-exploring* ()

Begin

```
1.  let $v$ be the first element in $V_0$;
2.  push($v$, H); mark $v$ 'accessed';
3.  while H is not empty do {
4.  $v$ := top(H); (*the top element of H is assigned to $v$.*)
5.  while neighbor($v$) $\neq \emptyset$ do {
6.  let $u$ be the first element in neighbor($v$);
7.  if $u$ is accessed then remove $u$ from neighbor($v$)
8.  else {push($u$, H); mark $u$ 'accessed'; $v$ := $u$;}
9.  }
10. if $v$ is neither in $V_{j*}$ nor in $V_0$ then pop(H)
11. Else {if $v$ is in $V_{j*}$ then output all the elements in H; (*all the elements in H make up a
    complete sequence.*)
12. remove all elements in H;
13. let $v$ be the next element in $V_0$;
14. push($v$, H); mark $v$;
15. }
```

End

The above algorithm works top-down, searching level by level. In each iteration of the outer **while**-loop, a complete sequence is explored (by executing the inner **while**-loop, lines 5 - 9) and stored in the stack ***H***. All the found complete sequences are node-disjoint since any repeated access of a node is blocked by using the mark 'accessed' (see lines 2, 7, and 8.)

Based on the above algorithm, the whole process to find a maximal set of node-disjoint paths is given below.

Algorithm *node-disjoint-paths*(A)

Begin

```
1.  $k := 1$;
2.  construct $A^{(1)}$;
3.  **while** $A^{(k)} \neq \Phi$ **do** {
4.  call *subgraph-exploring*($A^{(k)}$);
5.  let $P_1, \ldots P_l$ be all the found complete sequences;
6.  **for** $j = 1$ to $l$ **do**
7.  {let $P_j = v_1, v_2, \ldots, v_m$;
8.  mark $v_i \rightarrow v_{i+1}$ or unmark $v_{i+1} \rightarrow v_i$ ($i = 1, \ldots, m - 1$) according to (i) and (ii) above;
9.  }
10. $k := k + 1$; construct $A^{(k)}$;
11. }
```

end

The above algorithm runs in several phases. In each phase, a $A^{(k)}$ is constructed (see line 2, and 10). Then, *subgraph-exploring* () is invoked to find all node-disjoint complete sequences. Also, the arcs along each complete sequence will be marked or unmarked (see line 8). When we meet a $A^{(k)} = \Phi$, the algorithm terminates; and all the marked arcs in A make up a maximal set of node-disjoint paths.

A.2 Time complexity and correctness

In this subsection, we analyze the time complexity of *node-disjoint-paths*(A) and prove its correctness.

A.2.1 Time complexity

Lemma 7 Let $V_1^{(k)}, \ldots, V_{j_k}^{(k)}$ be the levels constructed when establishing $A^{(k)}$. Then, we have $j_k < j_{k+1}$ if (k + 1)th phase is not the last.

Proof. Let v be a node in $V_{j_{k+1}}^{(k)}$. If $v \notin V_{j_k}^{(k)}$, then, according to the definition of j^*, we must go along a longer path from a node in $V_1^{(k+1)}$ to v than any path from a node in $V_1^{(k)}$ to v. If $v \in V_{j_k}^{(k)}$, then, to reach v, we must go along a node sequence $v_1, \ldots, v_{j_{k+1}} = v$ such that there exist at least two consecutive nodes v_l, v_{l+1} ($1 < l < j_{k+1}$) with $v_{l+1} \rightarrow v_l$ being marked in the kth phase due to the alternating property. Going from v_l to v_{l+1} means a detour around. In terms of the definition of j^*, $j_{k+1} > j_k$. □

Lemma 8 Let P be a maximal set of node-disjoint paths in A. Let $V_1^{(k)}, \ldots, V_{j_1}^{(k)}$ be the levels when establishing $A^{(1)}$. Then, $j_1 \leq |V(A)|/|P|$.

Proof. Let α be the length of the shortest path in P. Then, we have

$$\alpha \cdot |P| \leq |V(A)|.$$

Therefore, $\alpha \leq |V(A)|/|P|$. However, $j_1 \leq \alpha$. Thus, the lemma follows. □

Foe each $A^{(k)}$, we define $B^{(k)}$ as follows.

(1)If $u \to v \in E(A^{(k)})$ is an unmarked arc, then $u \to v \in E(B^{(k)})$.
(2)If $u \to v \in E(A^{(k)})$ is a marked arc, then $v \to u \in E(B^{(k)})$.

We will prove that $B^{(k)}$ also has the alternating property.

Lemma 9 In $B^{(k)}$, for each node v, we have either $d^+(v) \le 1$ or $d^-(v) \le 1$.

Proof. We prove the lemma by induction on k.

Basis step. When $k = 1$. The lemma trivially holds.

Induction step. Assume that for $k \le h$, the lemma holds. We consider $B^{(h+1)}$. Let v be a node in $B^{(h+1)}$. If v does not appear in any complete sequences found in $A^{(h)}$. The indegree and outdegree of v are the same as in $B^{(h)}$. If v appears in a complete sequence found in , we do the following analysis. Let $v_1, ..., v_{i-1}, v, v_{i+1}, ..., v_l$ be a complete sequence found in $A^{(h)}$, on which v appears. Consider v_{i-1}, v, v_{i+1}, we have $d^+(v) = d^-(v_{i+1}) = 1$, or $d^+(v_{i-1}) = d^-(v) = 1$ according to the induction hypothesis. In both cases, neither of $v_{i-1} \to v$ and $v \to v_{i+1}$ appear in $B^{(h+1)}$. But in the former case, $u \to v$ (for some u) and $v \to v_{i-1}$ will be added into $B^{(h+1)}$, as shown by the dashed arrows in Fig. 16(a). Note that in $B^{(h+1)}$ $d^+(v) = d^-(v_{i-1}) = 1$. In the latter case, $v_{i+1} \to v$ and $v \to u'$ (for some u'). In this case, $d^+(v_{i+1}) = d^-(v_i) = 1$. See Fig. 16(b) for illustration. Thus, a bridge in is replaced with a different bridge in $B^{(h+1)}$. □

Proposition 6 The time complexity of the algorithm *node-disjoint-paths*(A) is bounded by O($\sqrt{|V(A)|}$ $|E(A)|$).

Proof. Let P be a maximal set of node-disjoint paths in A. If $|P| \le \sqrt{|V(A)|}$, then the number of phases is bounded by $\sqrt{|V(A)|}$, and the result follows. If $|P| > \sqrt{|V(A)|}$, we consider k such that in the kth phase the number of node-disjoint paths is $|P| - \sqrt{|V(A)|}$. Then, by constructing $A^{(k+1)}$, ..., $A^{(m)}$ (assume that the mth phase is the last), and then exploring them, we will find the rest node-disjoint paths. Assume that $V_1^{(k+1)}$, ..., $V_{j_{k+1}}^{(k+1)}$ be the levels constructed when establishing $A^{(k+1)}$. In terms of Lemma 7 and 8, $j_{k+1} \le |V(A)|/\sqrt{|V(A)|} = \sqrt{|V(A)|}$.

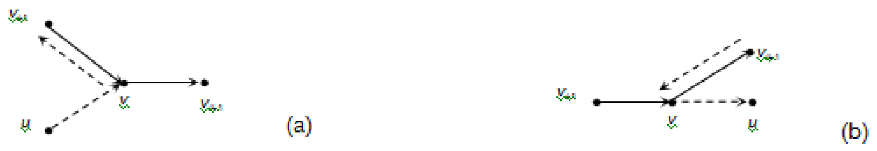


Fig. 16. Illustration for Lemma 9

In terms of Lemma 7, k must be less than j_{k+1}. Therefore, $k \le \sqrt{|V(A)|}$. Thus, the time spent for the first k phase is bounded by O($\sqrt{|V(A)|}$ $|E(A)|$*).* Also, the time for finding the rest node-disjoint paths is bounded by O($\sqrt{|V(A)|}$ $|E(A)|$). So the total cost is O($\sqrt{|V(A)|}$ $|E(A)|$) □.

A.2.2 Correctness

Lemma 10 Let $A^{(0)} = A, A^{(1)}, ..., A^{(k)} = \Phi$ be the graphs generated in the different phases during the execution of Algorithm *node-disjoint-paths*(A). Then, for each marked arc $u \to v$ in $A^{(i)}$ ($i = 1, ..., k - 1$), the following conditions are satisfied.

i) If u is not an origin, then there exists a node u' such that $u' \to u$ is marked in $A^{(i)}$.
ii) If v is not a terminus, then there exists a node v' such that $v' \to v$ is marked in $A^{(i)}$.

Proof. We prove condition (1) by induction on phases i.

Basic step. When $i = 1$. The proof is trivial.

Induction step. Assume that the lemma holds for $i \le j$. Consider phase $j + 1$. Let $v_1, ..., v_l$ be a complete sequence found in $A^{(j+1)}$. Without loss of generosity, assume that (v_g, v_{g+1}) $(1 \le g < l)$ corresponds to a marked arc. If (v_{g-1}, v_g) corresponds to a marked arc, condition (i) is proved. Otherwise, there exists a subsequence $v_h, v_{h+1}, ..., v_g$ such that each pair (v_{r+1}, v_r) corresponds to a marked arc in $A^{(j)}$. According to the induction hypothesis, if v_g is not an origin, there must exist a node v such that (v, v_g) corresponds to a marked arc in $A^{(j)}$. According to the algorithm, (v_{r+1}, v_r) will be unmarked, but (v, v_g) remains marked.

In the same way, we can prove condition (ii). □

From the proof of Lemma 10, we can also see that any two paths made up of marked arcs (from an origin to a terminus) are node-disjoint.

Proposition 7 The number of the node-disjoint paths in A found by *node-disjoint-paths*(A) is maximum.

Proof. Let $A^{(0)} = A, A^{(1)}, ..., A^{(k)} = \Phi$ be the graphs generated in the different phases during the execution of the algorithm. Let $V_0, V_1, ..., V_j$ be the levels when creating $A^{(}{}_{k)}$. Then, $V_j = \Phi$ and $V_i \cap$ {unsaturated terminus} $= \Phi$ for $i < j$. Consider the cut $(R, \overline{R})$, where $R = V_0 \cup V_1 \cup ... \cup V_{j-1}$ and $\overline{R} = V(A) \backslash (V_0 \cup V_1 \cup ... \cup V_{j-1})$. Each arc $u \to v$ with $u \in R$ and $v \in \overline{R}$ must be marked. Otherwise, $V_j \ne \Phi$. In addition, each two of such arcs are not on a same path. According to Lemma 10, each of such arcs corresponds to a node-disjoint path and the number of such arcs is maximum. This completes the proof □.