Chapter 9
Print ISBN: 978-81-19217-73-1, eBook ISBN: 978-81-19217-81-6

On the DAG Decomposition into Minimum Number of Chains

Yangjun Chen [a*] and Yibin Chen [a]

DOI: 10.9734/bpi/ratmcs/v1/5816E

Abstract

By the DAG decomposition, we mean the decomposition of 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. In this paper, we discuss an efficient algorithm for this problem. Its time complexity is bounded by $O(max\{\kappa, \sqrt{n}\} \cdot n^2)$ while the best algorithm for this problem up to now needs $O(n^3)$ time, where n is the number of the nodes of G, and 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. κ is in general much smaller than n. In addition, by the existing algorithm, $\Theta(n^2)$ 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. This is particularly important for some nowadays applications with massive graphs including millions and even billions of nodes, like the *facebook*, *twitter*, and some other social networks.

Keywords: Reachability queries; directed graphs; transitive closure; graph decomposition; chains.

1 Introduction

Directed graph (or digraph) data arise in many fields, especially in contemporary research on structures of social relationships [1,2]. A graph can be analyzed using either combinatorial graph-theoretic methods or by matrix representations such as the adjacency matrix. In the latter case, algebraic methods for analysis are available. In particular, the spectrum of the matrix associated with an undirected graph can be related to its structural properties [3,4]. 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. We discuss an efficient approach for finding a minimized C for G in this study.

[a] *The University of Winnipeg, Canada.*
**Corresponding author: E-mail: y.chen@uwinnipeg.ca;*

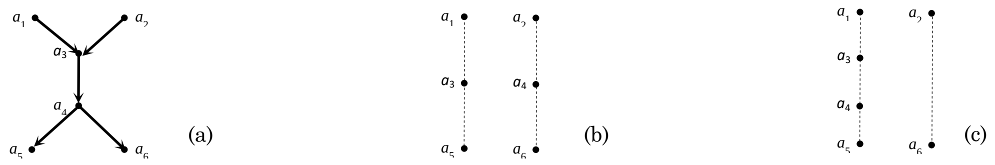
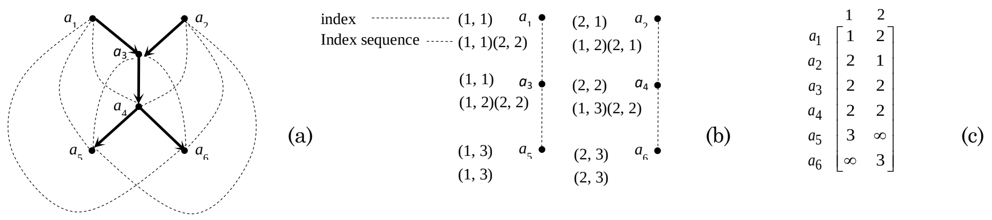


Fig. 1. Illustration for DAG decomposition



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

Fig. 2. Illustration for transitive closure and index sequences

As 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 networks, for which we may quite often ask whether a node v is reachable from another node u through a path in G [5,6,7].

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 $(v, u) \in E^*$ iff 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).)

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; otherwise, $\boldsymbol{M}[i, j] = 0$ [8]. 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 [9] 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 [10,11]. 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) … (i, j_i) … (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 iff 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 $v_1, ..., v_l$ 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 checkings:

- 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 $O(\sum_v d_v \cdot \kappa) = O(n \cdot m)$, 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 iff $\boldsymbol{M}_G(v, i) \leq j$. In practise, 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: 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 [12], 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 $a, c \in U$, $a \nsucc c$ and $c \nsucc a$. Furthermore, by using the Fulkerson's method [13], 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 [13], the number of chains constructed as described above is $n - |M|$. It must be minimum since in terms of König's theorem ([14], 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.

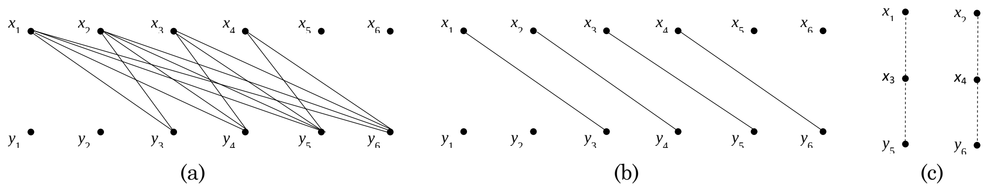

x_1 x_2 x_3 x_4 x_5 x_6
y_1 y_2 y_3 y_4 y_5 y_6
(a)
(b)
(c)

Fig. 3. Illustration for poset decomposition

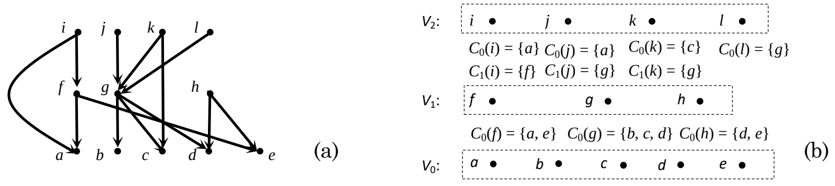

i j k l
f g h
a b c d e
(a)
V_2: i j k l
$C_0(i) = \{a\}$ $C_0(j) = \{a\}$ $C_0(k) = \{c\}$ $C_0(l) = \{g\}$
$C_1(i) = \{f\}$ $C_1(j) = \{g\}$ $C_1(k) = \{g\}$
V_1: f g h
$C_0(f) = \{a, e\}$ $C_0(g) = \{b, c, d\}$ $C_0(h) = \{d, e\}$
V_0: a b c d e
(b)

Fig. 4. Illustration for DAG stratification

In Fig. 3(a), we show the bipartite graph constructed for the transitive closure of the graph shown in Fig. 1(a). Fig. 3(b) shows one possible maximal matchings M of that bipartite graph, whose size is 4. From M we can establish a set of 2 chains by applying step (iii) shown above (see Fig. 3(c).) An interested reader could try to construct a bipartite graph over G, instead of G^*. Applying step (iii) above to the corresponding bipartite graph cannot generate a minimized set of chains.

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 [15]. However, using the algorithm proposed by Hopcroft and Karp [16], 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 [17], 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 [18-21,22,23]. 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. So, the number of paths found by the method discussed in [17] 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 [21] 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 [24] 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 a 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 [25] and [26] both with the same time complexity $O(\kappa\cdot n^2)$. However, in the method discussed in [25] each node is associated with a large data structure and requires $O(\kappa\cdot n^2)$ space in the worst case. By Chen and Chen [26], 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 Felsner and Wernisch [27] 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 $(i', j') \succ (i, j)$ iff $i' > i$ 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 Lou and Sarrafzadeh [28] and Goeman [29] are to find a maximum 2-chain and 1-chain in M, respectively. [28] needs $\Theta(n\cdot\log n)$ time while [29] 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 $O(n^2)$ 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, $d_{in}(v)$ and $d_{out}(v)$ represent v's indegree and v's outdegree, respectively.

ALGORITHM 1. *GraphStra*(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 $d_{out}(v) > k$ then remove $v$ from $W$;
8.          $G := G \backslash \{v \to v_1, ..., v \to v_k\}$;
9.          $d_{out}(v) := d_{out}(v) - k$;
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 $d_{out}(v)$ 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 $d_{out}(v) > k$ in line 7, where k is the number of v's 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 $d_{out}^{j}(v)$ 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 $v \in V_i$ ($i = 1, ..., j - 1$) $d_{out}^{i}(v) = 0$ while $d_{out}^{i-1}(v) \neq 0, ..., d_{out}^{0}(v) \neq 0$.

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

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

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 $d_{out}^{l+1}(v) = 0$ while $d_{out}^{0}(v) \neq 0, ..., d_{out}^{i}(v) \neq 0$. □

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 $d_{out}^{j}(v)$ 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).

In Fig. 4(b), the nodes of the DAG shown in Fig. 4(a) are divided into three levels: $V_0 = \{a, b, c, d, e\}$, $V_1 = \{f, g, h\}$, and $V_2 = \{i, j, k, l\}$. Associated with each node at each level is a set of links pointing to its children at different levels. For example, node g in V_1 is associated with three links respectively to nodes b, c, and d in V_0, denoted as $C_0(g) = \{b, c, d\}$. (For simplicity, we use $C_0(g) = \{b, c, d\}$ to represent three links from g to b, c, and d, 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* [14]) 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* [14]) 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.

In addition, we call an alternating path an α-segment if it starts and ends both at a covered edge, and a *β-segment* if it starts and ends both at a non-covered edge.

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 [16] and runs in O($\sqrt{n}\cdot m$) time, where $n = |V|$ and $m = |E|$. The algorithm proposed by Alt, Blum, Melhorn and Paul [30] 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 Description

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 [16]. 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 $E_2 = C_1^2$ is shown in Fig. 5(c) and a possible maximal matching M_2 of it is shown in Fig. 5(d).

Combining M_1 and M_2 by connecting their edges, we will get a set of seven chains, denoted by $M_1 \bar{\cup} M_2$ 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: $b — g — d — h — e$ (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 l (or k) to b as illustrated in Fig. 6(c), we will get a set of six chains as shown in Fig. 6(d). (Note that l and b are on a chain since there exists a path $l — g — b$, which connects l and b.) □

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

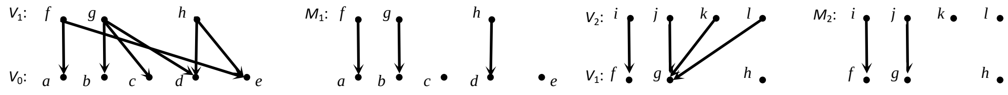


Fig. 5. Maximum matchings for bipartite graphs

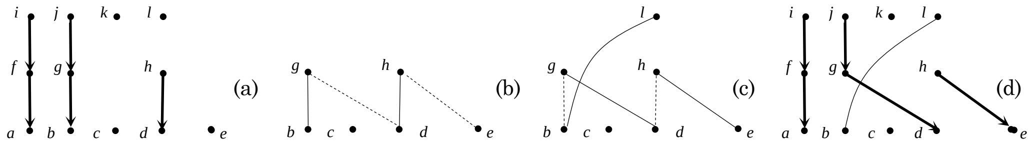


Fig. 6. Illustration for transferring edges on alternating paths

For this purpose, we introduce the concept of virtual nodes, in the next subsection, 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 Basic idea: 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 $i = 1, \dots, h$. 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 four kinds of virtual arcs, which are created in different ways:

alternating arc - If there exist a covered node $w \in V_{i-1}'$ (relative to M_i) such that one of v's parents is connected to w through an α-segment in $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$, 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 $u \rightarrow v'$ if it has not been created as an inherited or a transitive arc. It is referred to as an alternating arc of the first kind. 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), c and e are two free nodes. Then, two virtual nodes c' and e' (for c and e, respectively) will be created and added to V_1. Then, we have $V_1' = \{f, g, h, c', e'\}$. In addition, eight virtual arcs: $i \rightarrow c'$, $i \rightarrow e'$, $j \rightarrow c'$, $j \rightarrow e'$, $k \rightarrow c'$, $k \rightarrow e'$, $l \rightarrow c'$, and $l \rightarrow e'$ will be generated, shown as eight dashed arcs in Fig. 7(a).

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

Finally, $j \rightarrow e'$, $k \rightarrow c'$ and $l \rightarrow c'$ are three alternating arcs of the first kind, and $i \rightarrow c'$ is an alternating arcs of the second kind. We join j and e' since there is a node b that is connected to e's parent h through an α-segment: b — g — d — h (in $B(V_1, V_0, E_1)$, see Fig. 6(b)) and f is reachable from d in G through a node e (in V_1) (see Fig. 4(a).) For the same reason, we join k and c', and l and c'.

In Fig. 7(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. 7(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 reachability, 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.

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': $u \rightarrow v'$ with $u \in V_j$ ($j > i$) will be labeled as follows:

i) If $u \rightarrow v'$ is inherited or transitive, its label $label(u \rightarrow v')$ will be set to 0, indicating that $s(v')$ is reachable from u (through a path in G).
ii) If $u \rightarrow v'$ is an alternating arc, $label(u \rightarrow v')$ 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 $w_1 \to s(v')$, ..., $w_k \to s(v') \in \boldsymbol{C}_i$ 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 chain 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. 8(a). This graph can be divided into five levels as shown in Fig. 8(b).

In Fig. 9(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. 9(b). Relative to M_1, c, e and z are three free nodes in V_0. So three virtual nodes c', e' and z' will be created and added to V_1. At the same time, 15 arcs will be created. as shown in Fig. 9(c).

Among them, there are four transitive arcs: $t \to e'$, $t \to z'$, $i \to e'$, $i \to z'$; six alternating arcs of the first kind: $j \to c'$, $j \to e'$, $k \to c'$, $k \to e'$, $l \to c'$, $l \to e'$; and five alternating arcs of the second kind: $t \to c'$, $i \to c'$, $j \to z'$, $k \to z'$, $l \to z'$.

We have the transitive arc $t \to e'$ since e is reachable from t in G through a node f in V_1. The same claim applies to the other three transitive arcs.

The alternating arc of the first kind: $j \to c'$ is created since there is an alternating path $c - x - y - g - b$ in $B(V_1, V_0; \boldsymbol{C}_1)$ and b is reachable from j in G. In a similar way, we can analyze all the other five alternating arcs of the first kind. The alternating arc of the second kind: $t \to c'$ is due to the following facts:

- Free node c is on an alternating path $P_1 = c - x - y - g - b$ covered by another alternating path $P_2 = e - h - d - x - y - g - b$, which connects e and b, and
- e is reachable from t in G.

It is possible to transfer the edges on P_2 and then connect k (or l) to b to make e covered, which will, however, prevent c from being covered. But we can connect t to e and leave the chance for c to be covered along P_1. The virtual arc $t \to c'$ is created to represent this possibility.

Thus, $V_1' = \{c', e', z', g, x, h, f\}$. $B(V_2, V_1'; \boldsymbol{C}_2)$ is shown in Fig. 9(d). Assume that the maximal matching M_2 found for it is as shown in Fig. 9(e), and M_3 for $B(V_3, V_2'; \boldsymbol{C}_3)$ and M_4 for $B(V_4, V_3'; \boldsymbol{C}_4)$ are as shown in Fig. 9(f) and 9(g), respectively. By combining M_1, M_2, M_3 and M_4, we get $M_1 \ \bar{\cup}\ M_2 \ \bar{\cup}\ M_3 \ \bar{\cup}\ M_4$. This plus all the free nodes in V_4 make up a set of eight chains as shown in Fig. 10(a), and one of them contains only a single node.

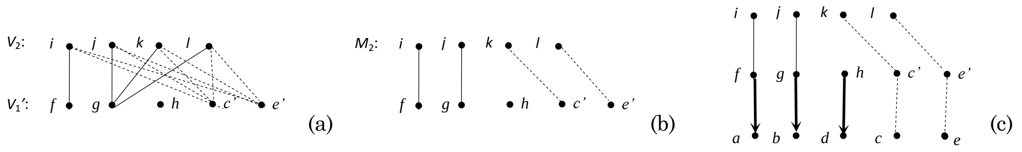


Fig. 7. Illustration for virtual nodes and chains containing virtual nodes

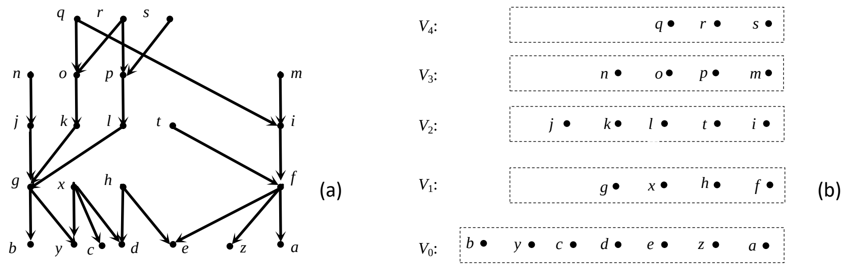
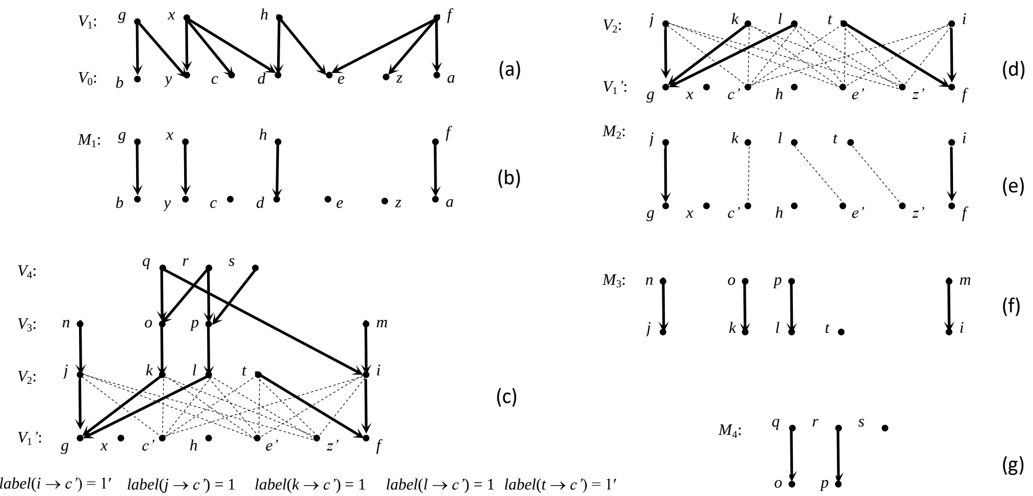


Fig. 8. A DAG and its stratification



$label(i \to c') = 1'$	$label(j \to c') = 1$	$label(k \to c') = 1$	$label(l \to c') = 1$	$label(t \to c') = 1'$
$label(i \to e') = 0$	$label(j \to e') = 1$	$label(k \to e') = 1$	$label(l \to e') = 1$	$label(t \to e') = 0$
$label(i \to z') = 0$	$label(j \to z') = 1'$	$label(k \to z') = 1'$	$label(l \to z') = 1'$	$label(t \to z') = 0$

Fig. 9. Illustration for generation of chains

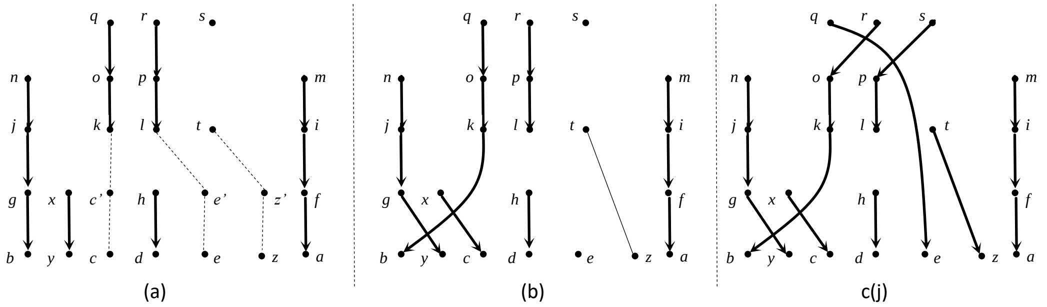


Fig. 10. Illustration for generation of chains

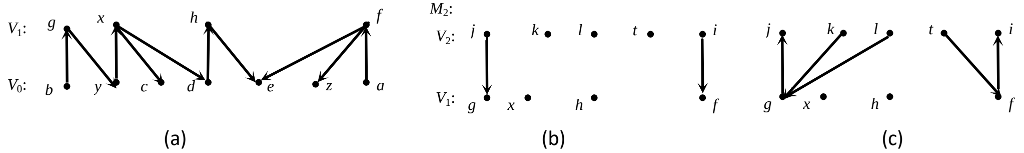


Fig. 11. Illustration for alternating graphs

We can simply connect t and z' since $t \rightarrow z'$ is a transitive arc. We can also transfer the edges on P_1 and then connect k and b as shown in Fig. 10(b). After that, removing e' will leave l and e disconnected, resulting in a set of nine chains. It is not minimum. In Fig. 10(c), we show a possible decomposition of eight chains. In the next subsection, we discuss how the problem can be figured out.

3.2.2 General algorithm for chain generation

To solve the above problem, we need to slightly modify the working process. First, we introduce a new concept.

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:

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

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

In Fig. 11(a), we show the alternating graph $\vec{B}_1$ with respect to M_1 for $B(V_1, V_0, \boldsymbol{C}_1)$ shown in Fig. 9(a). Assume that the maximum matching M_2 for $B(V_2, V_1, \boldsymbol{C}_2)$ is as shown in Fig. 11(b), the corresponding alternating graph is a graph shown in Fig. 11(c).

Next, we will combine two consecutive alternating graphs $\vec{B}_i = \vec{B}(V_i', V_{i-1}'; \boldsymbol{C}_i)$ and $\vec{B}_{i+1} = \vec{B}(V_{i+1}, V_i'; \boldsymbol{C}_{i+1})$, denoted as $\vec{B}_i \oplus \vec{B}_{i+1}$, by connecting each node in V_{i+1} to all its reachable nodes in V_{i-1}'. In Fig. 12(a), we show $\vec{B}_1 \oplus \vec{B}_2$ for the graph shown in Fig. 8(a).

What we want is to find a maximum set S of node-disjoint paths in $\vec{B}_i \oplus \vec{B}_{i+1}$, each starting from a free node u relative to M_{i+1} in V_{i+1}, and ending at a free node v relative to M_i in V_{i-1}', Let P be such a path which can always be divided into two parts: P' and P'' such that P' contains only the nodes in V_{i+1} while P'' contains only the nodes in V_{i-1}'. We will create a virtual node v' for v, connect it to the last node on P', and then transfer the edges on P'. However, for each free node (in V_{i-1}') not appearing on such a path, its virtual node will be added to V_{i+1}, for which only inherited and transitive arcs, as well as a new kind of virtual arcs, called supplementary arcs will be created.

supplementary arc – Let v' be a virtual node created for v in V_{i-1}' and added to V_{i+1}. If there exist a free node $w \in V_{i-1}'$ (relative to M_i) and a node $u \in V_j$ ($j > i$) such that one of

v's parents is connected to w through a β-segment in $B(V_i', V_{i-1}'; \boldsymbol{C}_i)$, satisfying one of the following two conditions:

- $u \rightarrow w \in E$, or
- there is an alternating path in $\vec{B}(V_i', V_{i-1}'; \boldsymbol{C}_i)$, which does not go through any node in S, but connects w to a node $w' \in V_{i-1}'$ such that w' is reachable from u,

add $u \rightarrow v'$ if it has not been created as an inherited or a transitive arc. $label(u \rightarrow v')$ is set to be i, same as an alternating arc incident to a virtual node added to V_i'. We create such an arc to resolve the conflict among free nodes in the case that they share a same alternating path P to a certain node. In this case, one free node, for example, node w can get covered by transferring the edges on P. But some other free node v which shares P with w may still get covered along P if it is possible to make w covered along a different alternating path, as demonstrated in the following example concerning $\vec{B}_1 \oplus \vec{B}_2$ shown in Fig. 11(a), in which we can find a maximum set of two paths: $P_1 = l \rightarrow b \rightarrow g \rightarrow y \rightarrow x \rightarrow d \rightarrow h \rightarrow e$ and $P_2 = t \rightarrow z$ as shown in Fig. 11(b). Then, we add two virtual node e' and z' to V_1 and a virtual node c' to V_2. Especially, P_1 can be divided into $P_1' = l$ and $P_1'' = b \rightarrow g \rightarrow y \rightarrow x \rightarrow d \rightarrow h \rightarrow e$; and P_2 into $P_2' = t$ and $P_2'' = z$. So e' will be connected to l according to P_1, and z' will be connected to t according to P_2. Furthermore, c' will be connected to m and q for the following reason:

- there is a free node e in V_1 which is connected to c's parent x through a β-segment: $x - d - h - e$, and
- e is reachable from both m and q in G.

See Fig. 13(a) for illustration.

In a next step, we will consider V_1', V_2', V_3 and determine new virtual nodes to be added to V_2' and V_3, by which any node in V_1', which does not have parents needn't be considered. Assume that the maximum matching M_2 found for $B(V_2, V_1; \boldsymbol{C}_2)$ is as shown in Fig. 10(b). With virtual nodes e' and z' added, M_2 is extended as illustrated in Fig. 13(b). There is no free node in V_1' relative to M_2, and thus no new virtual nodes will be added to V_2'. Finally, we will consider V_2', V_3', V_4. Assume that the maximum matching M_3 found for $B(V_3, V_2'; \boldsymbol{C}_3)$ is as shown in Fig. 13(c). Then, c' is a free node in V_2' relative to M_3. Further, assume that the maximum matching M_4 found for $B(V_4, V_3'; \boldsymbol{C}_4)$ is as shown in Fig. 13(d). A maximum set of node-disjoint paths in $\vec{B}_3 \oplus \vec{B}_4$ shown contains only one path: $P = s \rightarrow p \rightarrow r \rightarrow o \rightarrow q \rightarrow i \rightarrow m \rightarrow c'$, which can be divided into $P' = s \rightarrow p \rightarrow r \rightarrow o \rightarrow q$ and $P'' = i \rightarrow m \rightarrow c'$. So the virtual node c'' created for c' will be connected to q, as demonstrated in Fig. 13(e). (We notice that t does not have parents and therefore no virtual node for it will be generated.) Transferring edges on P', we will change M_4 to a matching as shown in Fig. 14(a), and the final chains $M_1 \bar{\cup} M_2 \bar{\cup} M_3 \bar{\cup} M_4$ is as shown in Fig. 14.

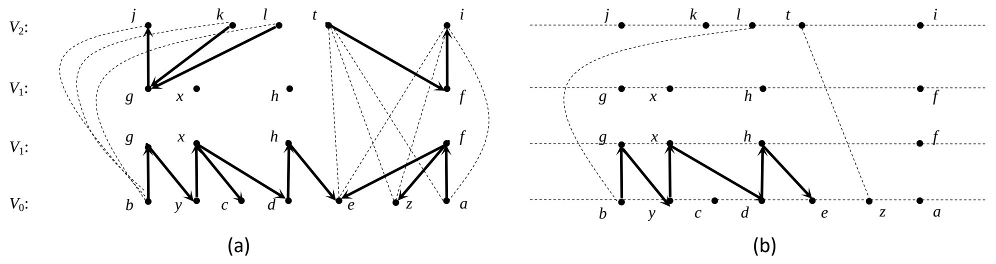


Fig. 12. Illustration for combined graphs and node-disjoint paths

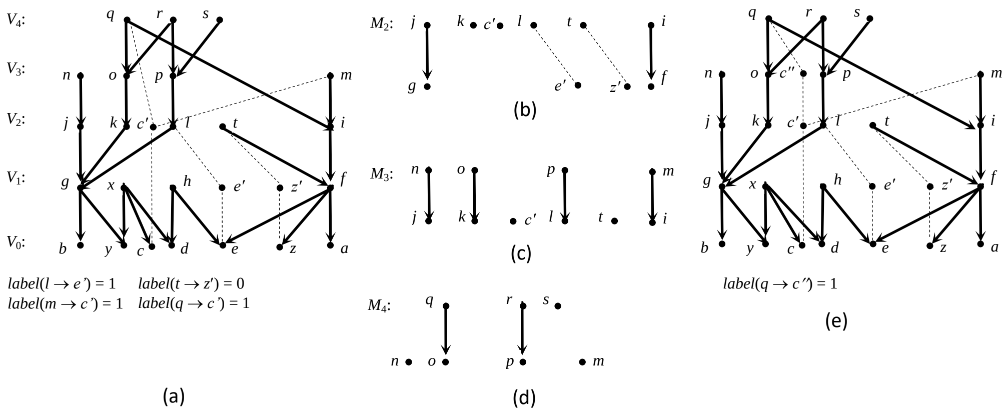

V_4:
V_3:
V_2:
V_1:
V_0:
q r s n o p m j k c′ l t i g x h e′ z′ f b y c d e z a
$label(l \rightarrow e') = 1$
$label(t \rightarrow z') = 0$
$label(m \rightarrow c') = 1$
$label(q \rightarrow c') = 1$
(a)
M_2:
(b)
M_3:
(c)
M_4:
(d)
c″
$label(q \rightarrow c'') = 1$
(e)

Fig. 13. Illustration for generation of virtual nodes

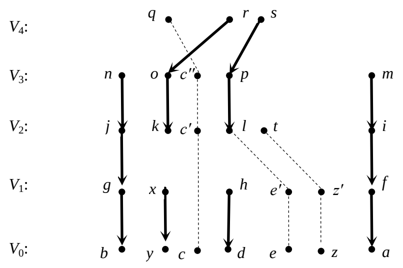


Fig. 14. Illustration for generation of virtual nodes

According to the above discussion, we design a process, denoted as *VirtualGen*(V_{i-1}', V_i', V_{i+1}, M_i), conducting the following task:

1. It takes V_{i-1}', V_i', V_{i+1}, M_i as the input.
2. Find M_{i+1} and form $\vec{B}_i \oplus \vec{B}_{i+1}$. Find a maximum set of node-disjoint paths in $\vec{B}_i \oplus \vec{B}_{i+1}$, each starting from a free node u relative to M_{i+1} in V_{i+1}, and ending at a free node v relative to M_i in V_{i-1}'. For each free node in V_{i-1}' appearing on a path in this set, the created virtual node v' is added to V_i'. For each free node in V_{i-1}' not appearing on a path in this set, the created virtual node v' is added to V_{i+1}. Create virtual arcs as described above.
3. M_{i+1} is used as the output of the process.

Based on this process, the general algorithm for the chain generation can be formally described as below.

ALGORITHM 2. *GenChain*(*stratification of G*)

```
input: a graph stratification.
output: a set of chains which may contain virtual nodes.
begin
1. $V_0' := V_0$; $V_1' := V_1$;
2. find $M_1$ for $B(V_1, V_0; \boldsymbol{C}_1)$;
3. for $i = 1$ to $h - 1$ do
4. {$M_{i+1}$ = VirtualGen($V_{i-1}'$, $V_i'$, $V_{i+1}$, $M_i$); }
5. $M := M_1 \cup \ldots \cup M_h$;
6. return $M$;
end
```

In the above algorithm, special attention should be paid to lines 1 - 2, by which the input for first call of *VirtualGen*( ) is prepared. In the main **for**-loop, the input for a next call of *VirtualGen*( ) is produced in the execution of the previous call of *VirtualGen*( ).

We also notice that at any point in time only the virtual nodes at the current level and the level just lower than the current are associated with supplementary arcs according to the following analysis.

Before the execution of *VirtualGen*(V_{i-1}', V_i', V_{i+1}, M_i), we may have some virtual nodes in V_i', which may be associated with supplementary arcs. During the execution of *VirtualGen*(V_{i-1}', V_i', V_{i+1}, M_i), we may continually add some more virtual nodes to both V_i' and V_{i+1}, and the virtual nodes added to V_{i+1} may be incident to supplementary arcs. However, when executing *VirtualGen*(V_i', V_{i+1}', V_{i+2}, M_{i+1}), any the virtual node in V_i' will become covered, or be promoted to V_{i+2}.

So, the number of virtual arcs maintained in the process is bounded by O($\kappa \cdot n$) since the number of supplementary arcs incident to a virtual node is bounded by O(n).

It remains to show how to find a maximal set of node-disjoint paths in $\vec{B}_i \oplus \vec{B}_{i+1}$. For this purpose, we define a maximum flow problem over $\vec{B}_i \oplus \vec{B}_{i+1}$, (with multiple sources and sinks) as follows:

- Each free node in V_{i+1} in B_{i+1} is designated as a *source*. Each free node in V_{i-1}' in B_i is designated as a *sink*.
- Each arc $u \rightarrow v$ 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.

3.2.3 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 $i \in h$ - 1} $|V_i'| \leq \kappa$.

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 $V_{(i+1)mod2}$ 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 a 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 $|\varphi_M(V_0) \cup M \cup \varphi_M(V_1))| = |A|$. 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 $1 \leq i \leq h - 1$. Then, for any $i \in h - 1$}, we have $|V_i'| \leq \kappa$.

Proof. First, we notice that $|V_1'| = |V_1| + |\varphi_{M_1}(V_0)| = |\varphi_{M_2}(V_1) \cup \varphi_{M_1}(V_1) \cup \xi_{M_1[1]}(V_0) \cup \xi_{M_1[2]}(V_1) \cup \xi_{M_1[3]}(V_0)| \leq k$. 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 do 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 the chain generation can be easily estimated.

First, using the Hopcroft and Karp algorithm [18], 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

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

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

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

Next, 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 $O(|V_{i-1}'| \cdot |V_i|) \leq O(\kappa \cdot |V_i|)$. 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 $w \rightarrow z \in \boldsymbol{C}_i$. Moreover, for each w, we have to further find all those nodes u such that $u \rightarrow w \in E$. So the cost for generating all the alternating arcs incident to v is bounded by $O(|V_{i-1}'| \cdot |V_i| + |V_i|^2 + |V_i| \cdot n)$. Therefore, the total cost for creating all the alternating arcs is bounded by

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

Finally, concerning the global alternating graphs, we have the following important lemma.

Lemma 5 For each arc e in $\vec{G}$ 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 V_i in $B(V_i, V_{i-1}; \boldsymbol{E}_i)$ ($i = 2, \ldots, h$) to a node in some V_j in $B(V_{j+1}, V_j; \boldsymbol{E}_{j+1})$ 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. □

Corollary 1 For each node v in $\vec{G}$, either there is only one arc emanating from it or only one arc entering it. □

By using Dinic's algorithm [18] for a maximum flow problem over a 0-1 network with the property described in Corollary 1, only $O(\sqrt{n}\cdot m)$ time is required, where n and m are the numbers of the nodes and arcs of the network, respectively. (See pp. 119 – 121 in [20]) Thus, cost$_{21}$ is bounded by

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

For estimating cost$_{22}$, we need to compute the costs for creating all the inherited, transitive and alternating arcs, as well as supplementary arcs. We remember that when executing *VirtualGen*(V_{i-1}', V_i', V_{i+1}, M_i) some virtual nodes will be added to V_i' and V_{i+1}. Each virtual node added to V_i' will be connected to a node in V_{i+1} (by an inherited, transitive, or alternating arc), but each virtual node added to V_{i+1} will be connected to some nodes at levels higher than $i + 1$ by supplementary arcs. In terms of the analysis in 3.2.2, cost$_{22}$ is bounded by $O(\kappa\cdot n)$.

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 $O(\kappa^2\cdot n + \kappa\cdot n^2)$.

The main space requirement of this process is bounded by $O(\kappa\cdot n)$ since at any time point, only for the virtual nodes at the current level and the level lower than the current some supplementary arcs have to be maintained, and it is obvious that the number of supplementary arcs incident to a virtual node is at most $O(n)$. Besides, for each virtual node not appearing at any of these two levels, only one virtual arc is produced. So for these virtual nodes the used space is also bounded by $O(\kappa\cdot n)$. In addition to 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 kept. Thus, the total space overhead is bounded by

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

3.3 Virtual node resolution

After the chain generation, the next step is to resolve (or say, to remove) virtual nodes from chains. In the following, we will first discuss the working process 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 top-down along the chains. Two steps will be carried out:

1. Remove virtual nodes, and at the same time connect some nodes according to the connectivity represented by them, and
2. Establish new connections between free nodes by transferring edges along alternating paths within a bipartite graph or cross more than one bipartite graph.

In the first step, we will check virtual nodes level by level, and change G_c to another graph G' containing part of G's transitive closure, which is necessary to find the final result. For doing this, the following rules should be followed.

i) Let v be a virtual node in V_i'. If v does not have a parent along the corresponding chain, it will be simply removed.
ii) If v has a parent u along a chain with $label(u \to v) = 0$, remove v and connect u to $s(v)$. $label(u \to s(v))$ is set to 0,
iii) If v has a parent u along a chain with $label(u \to v) = i$, connect u to each reachable node in $V_{i\text{-}1.}$

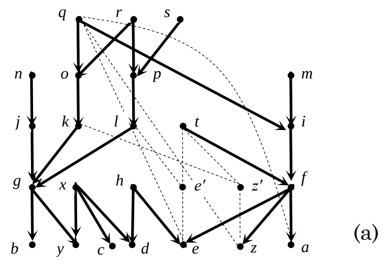
See Fig. 15 for illustration.

In Fig. 15(a), we show the resulting graph by removing c'', by which the parent q of c'' along the chain will be connected to all those nodes in V_0 that are reachable from q since $label(q \to c'') = 1$. After c'' is eliminated, c' becomes a node without a parent along a chain and is also removed. In Fig. 15(b), e' and z' are continually removed from the graph as shown in Fig. 15(a), by which

- l will be connected to b since $label(l \to e') = 1$ and b is the only node in V_0 reachable from l, and
- t will be connected to z since $label(t \to z') = 0$.

In the second step, we need to extend the concept of alternating graphs to include all the bipartite graphs, called a global alternating graph, and defined below.

Definition 6 (*global alternating graph*) Let $V = V_0 \cup ... \cup V_h$ be the stratification of $G(V, E)$. A global alternating graph of G, denoted as $\vec{G} = \vec{B}_1 \oplus \ldots \oplus \vec{B}_i \oplus \ldots \oplus \vec{B}_h$,is a directed graph obtained by separating G' into a set of bipartite graphs, by which each level, except for V_0 and V_h, will be stored two times (and will be considered to be different nodes.) In addition, for each arc $u \to v$ in G' with $u \in V_i$ and $v \in V_j$ ($j < i$), if (u, v) belongs to M_i, $v \to u$ is an arc in $\vec{G}$; otherwise, there is an arc from V_i in $B(V_i, V_{i\text{-}1}; \boldsymbol{E}_i)$ to V_j in $B(V_{j+1}, V_j; \boldsymbol{E}_{j+1})$. □



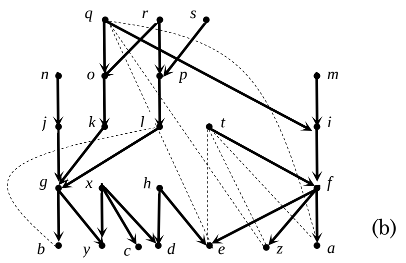


Fig. 15. Illustration for virtual node resolution in G_c

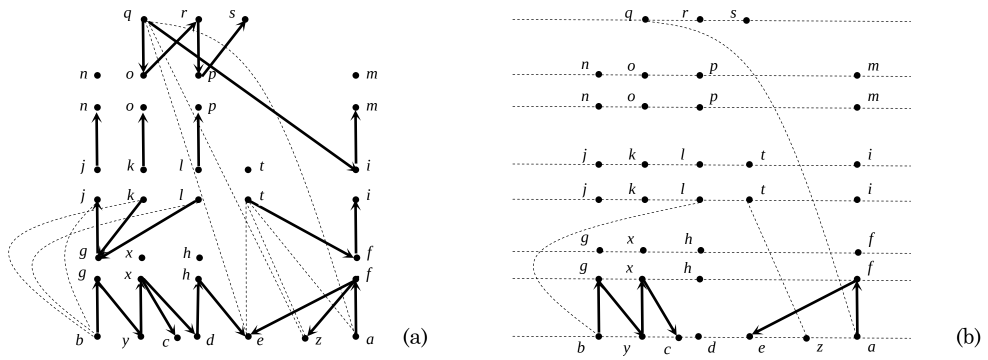


Fig. 16. Illustration for virtual node resolution in G_c

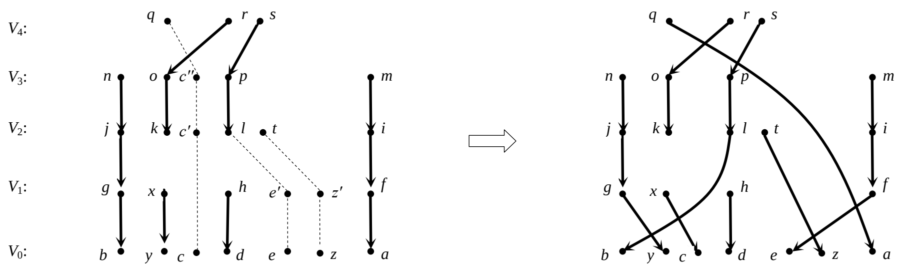


Fig. 17. Transform the chains generated by Algorithm *GenChain*( ) to the final result

In Fig. 16(a), we show such a global alternating graph, which is constructed by separating G' shown in Fig. 15(b) into four bipartite graphs.

We will then find a maximum set of node-disjoint paths with each starting from a node which is a parent of some virtual node along a chain, and ending at a node which is an actual free node relative to M_i in V_{i-1}' in $B(V_i, V_{i-1}'; \mathbf{C}_i)$ for $i = 1, \ldots, h - 1$. By transferring the arcs on these paths, we will get the final result. For example, in Fig. 16(b), we can see a possible maximum set of three paths in the graph shown in Fig. 16(a): $P_1 = q \to a \to f \to e$, $P_2 = l \to b \to g \to y \to x \to c$, and $P_3 = t \to z$. Transferring the arcs on each of these paths, we will transform the chains created by the algorithm *GenChain*( ) to a minimum set of chains containing no virtual nodes.

- Along P_1, we will connect node q to node a, cut off $a \to f$ on the corresponding chain, and then connect f to e.
- Along P_2, we connect node l to node b, cut off $b \to g$, connect g to y, cut off $y \to x$, and connect x to c.
- Along P_3, we connect node t to node z.

Fig. 17 demonstrates the final result.

Based on the above discussion, we give the following algorithm.

ALGORITHM 3. *RemoveVirtual* (G_c)

```
input: $G_c$.
output: a set of chains which contain no virtual nodes.
begin
1.  remove virtual nodes in $G_c$ top-down level by level to get $G'$;
2.  transform $G'$ to a global alternating graph $\vec{G}$;
3.  find a maximum set of node-disjoint paths in $\vec{G}$;
4.  let $P_1, \ldots, P_k$ be all the found node-disjoint paths;
5.  transfer the edges on each $P_i$ to change the chains obtained by
    Algorithm GenChain( );
6.  return the modified chains;
end
```

The dominant cost of the above algorithm mainly comprises two parts:

cost$_3$ – the cost for generating G' from G_c, and
cost$_4$ – the cost for finding a maximum set of node-disjoint paths in G_c'.

cost$_3$ is the same as the cost for generating chains since the construction of G' from G_c is just a reverse process of the chain generation.

cost$_4$ is the cost for finding a maximum flow in a special kind of 0-1 networks, where either there is only one arc emanating from it or only one arc entering it (see Corollary 1).

Using Dinic's algorithm [18] for this problem, only $O(\sqrt{n} \cdot e)$ time is required. (See pp. 119 – 121 in [20])

So we have the following proposition.

Proposition 3 The time complexity of Algorithm *RemoveVirtual*( ) is bounded by $O(max\{\kappa, \sqrt{n}\} \cdot n^2)$. □

The space overhead of this process is also $O(\kappa \cdot n)$ since no more space than the chain generation is used.

4 Correctness

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

First, we consider the transitive closure $G^*(V, E^*)$ of G, and 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 $u \to v \in E^*$.

Concerning B_G, we have the following lemma.

Lemma 6 Let $M = \{(x_{v_1}, y_{u_1}), (x_{v_2}, y_{u_2}), \ldots, (x_{v_k}, y_{u_k})\}$ be a maximum matching of B_G. By joining any two edge (x_{v_i}, y_{u_i}), (x_{v_j}, y_{u_j}) if y_{u_i} and x_{v_j} are identical, we form a set of chains. These chains plus all the free nodes relative to M make up a minimum decomposition of G. □

Proof. See [12,31,32]. (Also, see pp. 190-191 in [14].) □

Proposition 4 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 1 and 2.

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 F from $G(V, E)$ as below:

1. Stratify G, dividing V into $V_0, \ldots, V_h$ (i.e., $V = V_0 \cup \ldots \cup V_h$).
2. Find a maximum matching M_1 of $B(V_1, V_0; \boldsymbol{E}_1)$. 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 F is of height k. According to the induction hypothesis, our algorithm is able to find a minimum set of chains of F, which corresponds to a maximum matching M_F of B_F, a bipartite graph built over F^*.

Let $v_1, \ldots, v_l$ be all the free nodes in V_0, relative to M_1; and $v_1', \ldots, v_l'$ be their respective virtual nodes. According to our algorithm, G_c is constructed by adding $B(V_1, V_0; \boldsymbol{E}_1)$ to F_c, removing $v_1', \ldots, v_l'$, and connecting the parents of these virtual nodes to their respective reachable nodes in V_0. Then, in $\vec{G} = \vec{B}_1 \oplus \ldots \oplus \vec{B}_i \oplus \ldots \oplus \bar{B}_h$, relative to the matching $(M_F \backslash A) \cup M_1$ in B_G, where $A = \{e|\ e \in M_F$ and incident to some v_i', $i = 1, \ldots, l\}$, any augmenting path connects an end node u of some chain for F (after all v_i' have been removed) to a free node relative to $(M_F \backslash A) \cup M_1$. Transferring the edges on a maximum set of node-disjoint augmenting paths, we will get a maximum matching for B_G. According to Lemma 6, the number of final chains found by our algorithm is minimum. □

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 $O(\kappa \cdot n)$ 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] Holland PW, Leinhardt S. An exponential family of probability distributions for directed graphs. Journal of the American Statistical Association. 1981 Mar 1;76(373):33-50.

[2] Ore O. Studies on directed graphs, I. Annals of Mathematics. 1956 May 1;383-406.

[3] Mezić I, Fonoberov VA, Fonoberova M, Sahai T. Spectral complexity of directed graphs and application to structural decomposition. Complexity. 2019 Jan 1;2019.

[4] Chung FR. Spectral graph theory. American Mathematical Soc; 1997.

[5] Chen Y, Chen YB. Decomposing DAGs into Spanning Trees: A New Way to Compress Transitive Closures, in Proc. of Conf. on Data Engineering, IEEE, 2011, pp. 1007 – 1018.

[6] Chen Y, Chen YB, Zhang Y. Evaluation of reachability queries based on recursive DAG decomposition, accepted by IEEE Transactions on Knowledge and Data Engineering; Nov. 08, 2022.

[7] Su J, Zhu Q, Wei H, Jeffrey Xu Yu. Reachability querying: Can it be even faster? IEEE Transactions on Knowledge and Data Engineering. 2017;29(3):683-697.

[8] Coppersmith D, Winograd S. Matrix multiplication via arithmetic progression. Journal of Symbolic Computation. 1990;9:251-280.

[9] Tarjan R. Depth-first search and linear graph algorithms. SIAM J. Compt. June 1972;1(2):146 -140.

[10] Gallai T, Milgram AN. Verallgemeinerung eines Graphentheoretischen Satzes von Reedei. Acta Sci. Math. Hung. 1960;21:429-440.

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

[12] Dilworth RP. A decomposition theorem for partially ordered sets. Ann. Math. 1950;**51**:161-166.

[13] Fulkerson DR. Note on Dilworth’s embedding theorem for partially ordered sets. Proc. Amer. Math. Soc. 1956;7:701-702.

[14] Asratian AS, Denley T, Haggkvist R. Bipartite graphs and their applications. Cambridge University; 1998.

[15] Warren HS. A modification of Warshall’s algorithm for the transitive closure of binary relations. Commun. ACM. April 1975;18(4):218 - 220.

[16] Hopcroft JE, Karp RM. An $n^{2.5}$ algorithm for maximum matching in bipartite graphs SIAM J. Comput. 1973;2:225-231.

[17] Jagadish HV. A compression technique to materialize transitive closure. ACM Trans. Database Systems. 1990;15(4):558 - 598.

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

[19] Karzanov AV. Determining the maximal flow in a network by the method of preflow. Soviet Math. Dokl. 1974;15:434-437.

[20] Even S. Graph algorithms. Computer Science Press, Inc., Rockville, Maryland; 1979.

[21] Chekuri C, Bender M. An efficient approximation algorithm for minimizing makespan on uniformly related machines. Journal of Algorithms. 2001;41:212-224.

[22] Lawler EL. Combinatorial optimization: Neworks and Matroids. Holt, Rinehart, and Winston, New York; 1976.

[23] Malhotra VM, Kumar MP, Maheshwari SN. An $O(|V|^3)$ algorithm for finding maximum flows in networks, computer science program. Indian Institute of Technology, Kanpur 208016, India; 1978.

[24] Chen Y, Chen YB. An efficient algorithm for answering graph reachability queries, in Proc. 24th Int. Conf. on Data Engineering (ICDE 2008), IEEE. April 2008;892-901.

[25] Chen Y, Chen YB. On the decomposition of posets, in Proc. 2nd Int. Conf. on Computer Science and Service System (CSSS 2012), IEEE, Aug. 11-13, Nanjing, China. 2012; 1115 - 1119.

[26] Chen Y, Chen YB. On the decomposition of posets into minimized set of node-disjoint chains. 2013 Int. Conf. on Computer, Networks and Communication Engineering (ICCNCE 2013), Beijing, China. May 23-24, 2013;131-135.

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

[28] Lou RD, Sarrafzadeh M. An optimal algorithm for the maximum two-chain problem. SIAM J. Disc. Math. 1992;**5**(2):285-304.

[29] Goeman H. 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.

[30] Alt H, Blum N, Mehlhorn K, Paul M. Computing a maximum cardinality matching in a bipartite graph in time O(). Information Processing Letters. 1991;37:237 -240.

[31] Perles MA. A proof of Dilworth’s decomposition theorem for partially ordered sets. Israel J. of Math. 1963;1:105-107.

[32] Tverberg H. On Dilworth’s decomposition theorem for partially ordered sets. J. Comb. Th. 1967;3:305-306.

APPENDIX – Find node-disjoint paths alternating 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 $\boldsymbol{G}$. Its time complexity is bounded by $O(e\cdot\sqrt{n})$, where $n = V(\boldsymbol{G})$ and $e = E(\boldsymbol{G})$. It is in fact a modified version of Dinic's algorithm [18], adapted to combined graphs, in which each path from a virtual node to a free node relative to M_{i+1} or relative to M_i is an alternating path, and for each edge $(u, v) \in M_{i+1} \cup M_i$, we have $d_{out}(u) = d_{in}(v) = 1$. Therefore, for any three nodes v, v', and v'' on a path in $\boldsymbol{G}$, we have $d_{out}(v) = d_{in}(v') = 1$, or $d_{out}(v') = d_{in}(v'') = 1$. 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 $d_{out}(u) = d_{in}(v) = 1$ is called a *bridge*.

Our algorithm works in multiple phases. In each phase, the arcs in $\boldsymbol{G}$ will be marked or unmarked. We also call a virtual node in $\boldsymbol{G}$ 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 $\boldsymbol{G}$ 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 of A will be explored, which is defined as follows.

Let V_0 be the set of all the unsaturated origins (appearing in). Define V_j ($j > 0$) as follows [16]:

$E_{j-1} = \{u \to v \in E(A) \mid u \in V_{j-1}, v \notin V_0 \cup V_1 \cup ... \cup V_{j-1},$
$u \to v$ is unmarked$\} \cup \{v \to u \in E(A) \mid u \in V_{j-1}, v \notin V_0 \cup V_1 \cup ... \cup V_{j-1},$
$v \to u$ 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 $v_j \rightarrow v_{j+1}$ in , or a marked arc $v_{j+1} \rightarrow v_j$ in $A^{(k)}$.

Our algorithm will explore 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 there exists j such that $V_j = \Phi$ and $V_i \cap$ {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 5 Consider a A shown in Fig. 12(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. 12(b). With respect to these marked arcs, a second subgraph (in the second phase) will be constructed as shwon in Fig. 12(c). In this phase, $V_0 = \{b\}$ (since node a is saturated), $V_1 = \{d\}$, $V_2 = \{a\}$ (note that $a \rightarrow d$ is marked), $V_3 = \{c\}$, $V_4 = \{e\}$, $V_5 = \{g\}$, $V_6 = \{f\}$ (note that $f \rightarrow g$ is marked), $V_7 = \{h\}$; and $j^* = 7$. By exploring , 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 shown in Fig. 12(d). Fig. 12(e) shows all the marked arcs, which make up two node-disjoint paths: $a \rightarrow c \rightarrow e \rightarrow g$ and $b \rightarrow d \rightarrow f \rightarrow h$.

The following algorithm is devised to explore an $A^{(k)}$, in which a stack $\boldsymbol{H}$ is used to store complete sequences. In addition, for each v in , *neighbor*(v) represents a set of 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 $neighbor(v) \neq \Phi$ do {
6.               let $u$ be the first element in $neighbor(v)$;
7.               if $u$ is accessed then remove $u$ from $neighbor(v)$
8.               else {$push(u, \boldsymbol{H})$; mark $u$ 'accessed'; $v := u$;}
9.           }
10.          if $v$ is neither in $V_{j^*}$ nor in $V_0$ then $pop(\boldsymbol{H})$
11.          else {if $v$ is in $V_{j^*}$ then output all the elements in $\boldsymbol{H}$;
                 (*all the elements in $\boldsymbol{H}$ make up a complete sequence.*)
12.              remove all elements in $\boldsymbol{H}$;
13.              let $v$ be the next element in $V_0$;
14.              $push(v, \boldsymbol{H})$; mark $v$;
15.          }
`6.  }
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 $A^{(k)}$;
3.   while $A^{(k)} \neq \Phi$ do {
4.       call subgraph-exploring();
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, an $A^{(k)}$ 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 an $A^{(k)} = \Phi$, 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 2 Let , ..., be the levels constructed when establishing . 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 $v_1, ..., v_{j_{k+1}} = v$ such that there exist at least two consecutive nodes v_l, v_{l+1} ($1 < l < j_{k+1}$) with $v_{l+1} \to v_l$ 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 3 Let P be a maximal set of node-disjoint paths in A. Let $V_1^{(k)}$., ..., $V_{j_1}^{(k)}$. 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 4 In $B^{(k)}$, for each node v, we have either $d_{out}(v) \leq 1$ or $d_{in}(v) \leq 1$.

Proof. We prove the lemma by induction on k.
Basis step. When k = 1. The lemma trivially holds.

Induction step. Assume that for $k \leq h$, 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 $A^{(k)}$. The indegree and outdegree of v are the same as in $B^{(h)}$. If v appears in a complete sequence found in $A^{(h)}$, 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 $d_{out}(v) = d_{in}(v_{i+1}) = 1$, or $d_{out}(v_{i-1}) = d_{in}(v) = 1$ 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 , as shown by the dashed arrows in Fig. 13(a). Note that in $B^{(h+1)}$ $d_{out}(v)$

= $d_{in}(v_{i-1})$ = 1. In the latter case, $v_{i+1} \rightarrow v$ and $v \rightarrow u'$ (for some u'). In this case, $d_{out}(v_{i+1})$ = $d_{in}(v_i)$ = 1. See Fig. 13(b) for illustration. Thus, a bridge in is replaced with a different bridge in $B^{(h+1)}$. □

Proposition 3 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 $|P| \leq \sqrt{|V(A)|}$, 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 $V_1^{(k+1)}$, ..., $V_{j_k}^{(k+1)}$ be the levels constructed when establishing . In terms of Lemma 2 and 3, $j_k+1 \leq |V(A)|/\sqrt{|V(A)|} = \sqrt{|V(A)|}$.

In terms of Lemma 1, k must be less than j_k+1. Therefore, $k \leq \sqrt{|V(A)|}$. 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 5 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 $u \rightarrow v$ 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 $u' \rightarrow u$ is marked in $A^{(i)}$.
ii) If v is not a terminus, then there exists a node v' such that $v' \rightarrow v'$ 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 $i \leq j$. 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}) ($1 \leq g < l$) 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 5, we can also see that any two paths made up of marked arcs (from an origin to a terminus) are node-disjoint.

Proposition 4 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 $(R, \bar{R})$, where $V_0 \cup V_1 \cup ... \cup V_{j-1}$ and $\bar{R} = V(A)\backslash(V_0 \cup V_1 \cup ... \cup V_{j-1})$. Each arc $u \rightarrow v$ with $u \in R$ and $v \in \bar{R}$ must be marked. Otherwise, $V_j \neq \Phi$. In addition, each two of such arcs are not on a same path. According to Lemma 5, each of such arcs corresponds to a node-disjoint path and the number of such arcs is maximum. This completes the proof. □

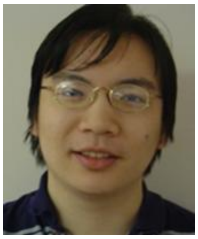
Biography of author(s)

Yangjun Chen
The University of Winnipeg, Canada.

Research and Academic Experience: He got his PhD in Computer Science from the University of Kaiserslautern, Germany, in 1995. He is now a professor in Dept. Applied Computer Science, University of Winnipeg, Canada.

Research Area: His area of research includes algorithm design and Databases.

Number of Published Papers: He has about 200 publications in Computer Science and Computer engineering.



Yibin Chen
The University of Winnipeg, Canada.

Research and Academic Experience: He has received the BS and master’s degree from the Department of Electrical and Computer Engineering, University of Waterloo, and the Department of Electrical and Computer Engineering, University of Toronto, Canada, respectively. Now he is a software engineer.

Research Area: His area of research mainly focused on Software engineering.

Number of Published Papers: He has published 13 research articles in several reputed journals.

DISCLAIMER
This chapter is an extended version of the article published by the same author(s) in the following journal.
British Journal of Mathematics & Computer Science, 10(6): 1-27, 2015.