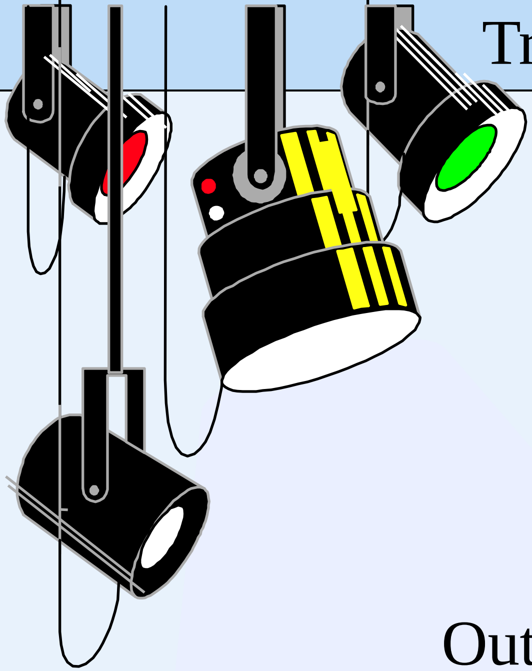
Transitive Closure Compression



Outline: Transitive Closure Compression

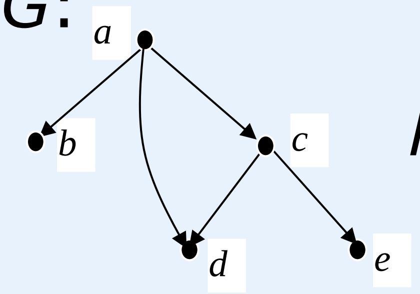
- Motivation
- DAG decomposition into node-disjoint chains
    - Graph stratification
    - Virtual nodes
    - Maximum set of node-disjoint paths

Motivation

- **A simple method**

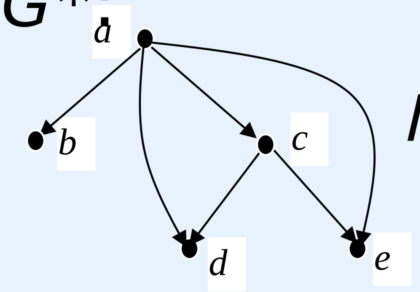
  - *store a transitive closure as a matrix*

G:



$$M = \begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 1 & 0 \\ b & 0 & 0 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 1 & 1 \\ d & 0 & 0 & 0 & 0 & 0 \\ e & 0 & 0 & 0 & 0 & 0 \end{array}$$

G^*:



$$M^* = \begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 1 & 1 \\ b & 0 & 0 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 1 & 1 \\ d & 0 & 0 & 0 & 0 & 0 \\ e & 0 & 0 & 0 & 0 & 0 \end{array}$$

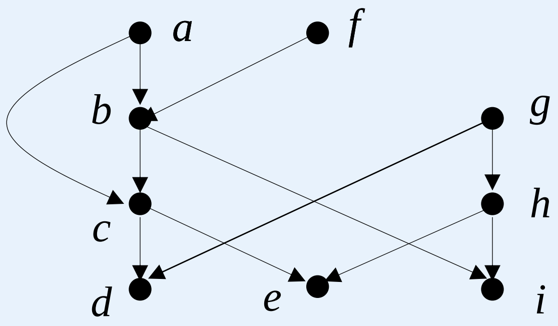
Space overhead: $O(n^2)$

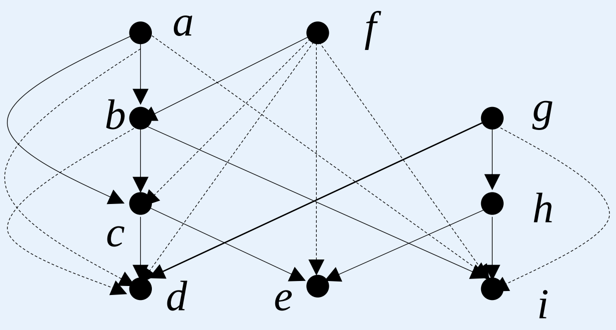
Query time: $O(1)$

DAG Decomposition into Node-Disjoint Chains

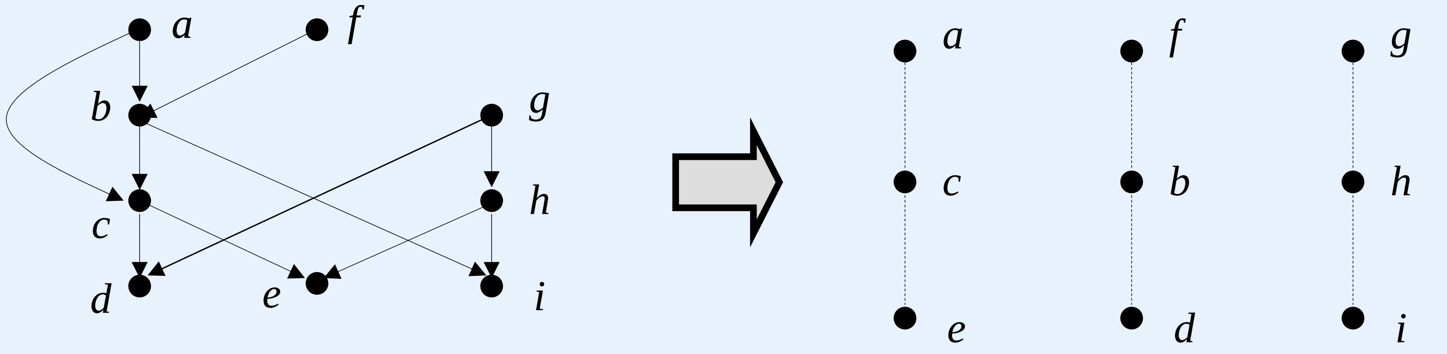
A DAG is a directed acyclic graph (a graph containing no cycles).

On a chain, if node *v* appears above node *u*, there is a path from *v* to *u* in *G*.





Decomposition of a DAG into a set of node-disjoint chains

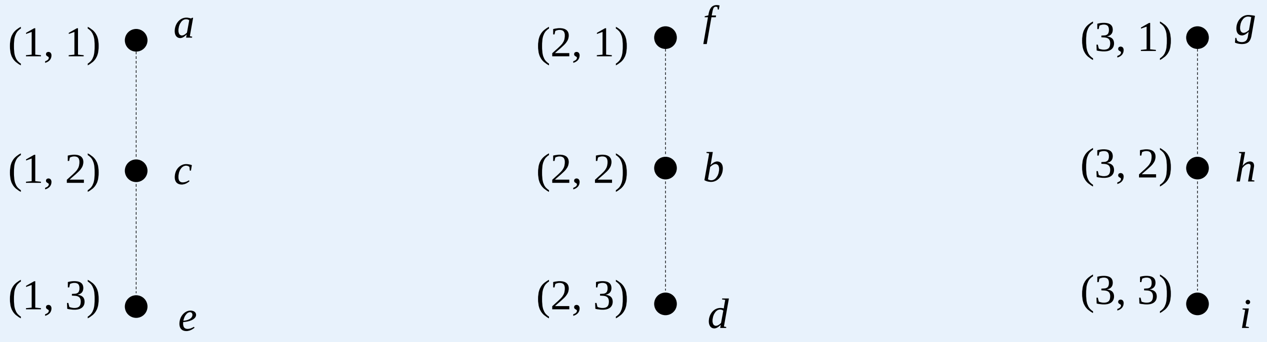


Transitive Closure Compression

Based on such a chain decomposition, we can assign each node an index as follows:

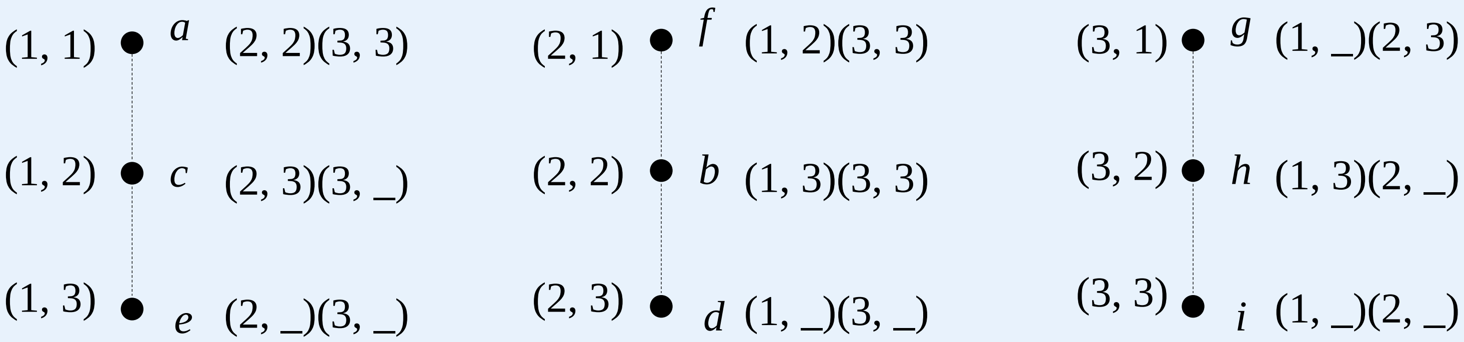
(1) Number each chain and number each node on a chain.

(2) The *j*th node on the *i*th chain will be assigned a pair (*i*, *j*) as its index.



Transitive Closure Compression

Each node v on the ith chain will also be associated with an index sequence of length k : $(1, j_1) \ldots (i - 1, j_{i-1})\ (i + 1, j_{i+1}) \ldots (k, j_k)$ such that any node with index (x, y) is a descendant of v if $x = i$ and $y < j$ or $x \neq i$ but $y \leq j_x$, where k is the number of the disjoint chains.



(1, 1) a (2, 2)(3, 3)
(1, 2) c (2, 3)(3, _)
(1, 3) e (2, _)(3, _)

(2, 1) f (1, 2)(3, 3)
(2, 2) b (1, 3)(3, 3)
(2, 3) d (1, _)(3, _)

(3, 1) g (1, _)(2, 3)
(3, 2) h (1, 3)(2, _)
(3, 3) i (1, _)(2, _)

The space complexity is bounded by $O(kn)$.

Construction of Index Sequences

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

Transitive Closure Compression

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

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

$$O(\sum_v d_v k) = O(ke),$$

where e is the number of edges of G.

Graph Stratification

Definition (*DAG stratification*) Let $G(V, E)$ be a DAG. The stratification of G is a decomposition of V into subsets $V_1, V_2, \ldots, V_h$ such that $V = V_1 \cup V_2 \cup \ldots V_h$ and each node in V_i has its children appearing only in $V_{i-1}, \ldots, V_1$ ($i = 2, \ldots, h$), where h is the height of G, i.e., the length of the longest path in G.

For each node v in V_i, its level is said to be i, denoted $l(v) = i$. In addition, $C_j(v)$ ($j < i$) represents a set of links with each pointing to one of v's children, which appears in V_j. Therefore, for each v in V_i, there exist $i_1, \ldots, i_k$ ($i_l < i$, $l = 1, \ldots, k$) such that the set of its children equals $C_{i_1}(v) \cup \ldots \cup C_{i_k}(v)$. Assume that $V_i = \{v_1, v_2, \ldots, v_k\}$. We use $\mathbf{C}_j^i$ ($j < i$) to represent $C_j(v_1) \cup \ldots \cup C_j(v_l)$.

Such a DAG decomposition can be done in O(*e*) time, by using the following algorithm.

Transitive Closure Compression

G_1/G_2 - a graph obtained by deleting the edges of G_2 from G_1.

$G_1 \cup G_2$ - a graph obtained by adding the edges of G_1 and G_2 together.

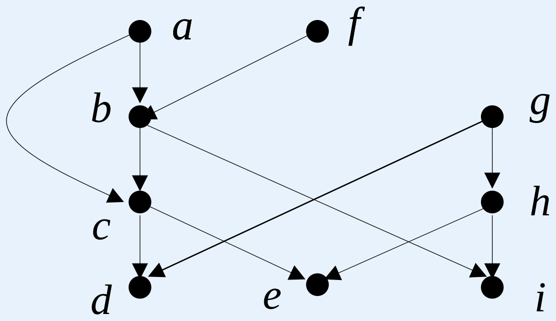
(v, u) - an edge from v to u. $d(v)$ - v's outdegree.

```
Algorithm graph-stratification($G$)
begin
  $V_1$ := all the nodes with no outgoing edges;
  for $i = 1$ to $h - 1$ do
    { $W$ := all the nodes that have at least one child in $V_i$;
      for each node $v$ in $W$ do
        { let $v_1, ..., v_k$ be $v$'s children appearing in $V_i$;
          $C_i(v)$ := {links to $v_1, ..., v_k$};
          if $d(v) > k$ then remove $v$ from $W$;
          $G := G/\{(v, v_1), ..., (v, v_k)\}$;
          $d(v) := d(v) - k$;}
      $V_{i+1} := W$;
    }
end
```

Transitive Closure Compression

- In the above algorithm, we first determine V_1, which contains all those nodes having no outgoing edges (see line 1).
- In the subsequent computation, we determine V_2, ..., V_h. In order to determine V_i ($i > 1$), we will first find all those nodes that have at least one child in V_{i-1} (see line 3), which are stored in a temporary variable W. For each node v in W, we will then check whether it also has some children not appearing in V_{i-1}, which can be done in a constant time as demonstrated below. During the process, the graph G is reduced step by step, and so does $d(v)$ for each v (see lines 8 and 9).
- First, we notice that after the *j*th iteration of the out-most **for**-loop, V_1 , ..., V_{j+1} are determined. Denote $G_j(V, E_j)$ the reduced graph after the *j*th iteration of the out-most **for**-loop. Then, any node v in G_j, except those in $V_1 \cup ... \cup V_{j+1}$, does not have children appearing in $V_1 \cup ... \cup V_j$. Denote $d_j(v)$ the outdegree of v in G_j. Thus, in order to check whether v appearing in G_{i-1} has some children not appearing in V_i, we need only to check whether $d_{i-1}(v)$ is strictly larger than k, the number of the child nodes of v appearing in V_i (see line 7).

Transitive Closure Compression



V_4:	a ●	$C_3(a) = \{c\}$		f ●	$C_3(f) = \{b\}$
V_3:	b ●	$C_2(b) = \{c\}$ $C_1(b) = \{i\}$		g ●	$C_2(g) = \{h\}$ $C_1(g) = \{d\}$
V_2:	c ●	$C_1(c) = \{d, e\}$		h ●	$C_1(h) = \{e, i\}$
V_1:	d ●		e ●	i ●	

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

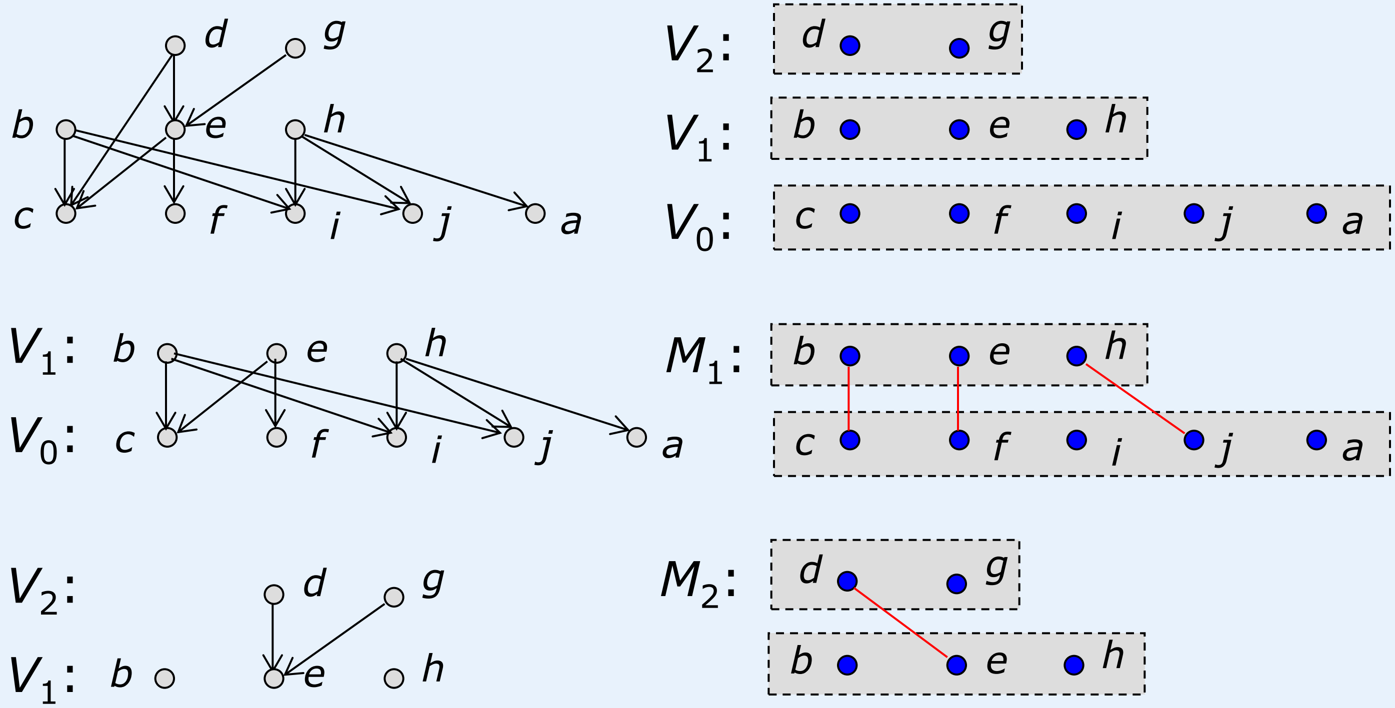
Transitive Closure Compression

Find a minimum set of node disjoint chains for a given DAG *G* such that on each chain if node *v* is above node *u*, then there is a path from *v* to *u* in *G*.

Step 1: Stratify *G* into a series of bipartite graphs.
Step 2: Find a maximum matching for each bipartite graph (which may contain the so-called virtual nodes.) All the matchings make up a set of node-disjoint chains.
Step 3: resolve all the virtual nodes.

Transitive Closure Compression

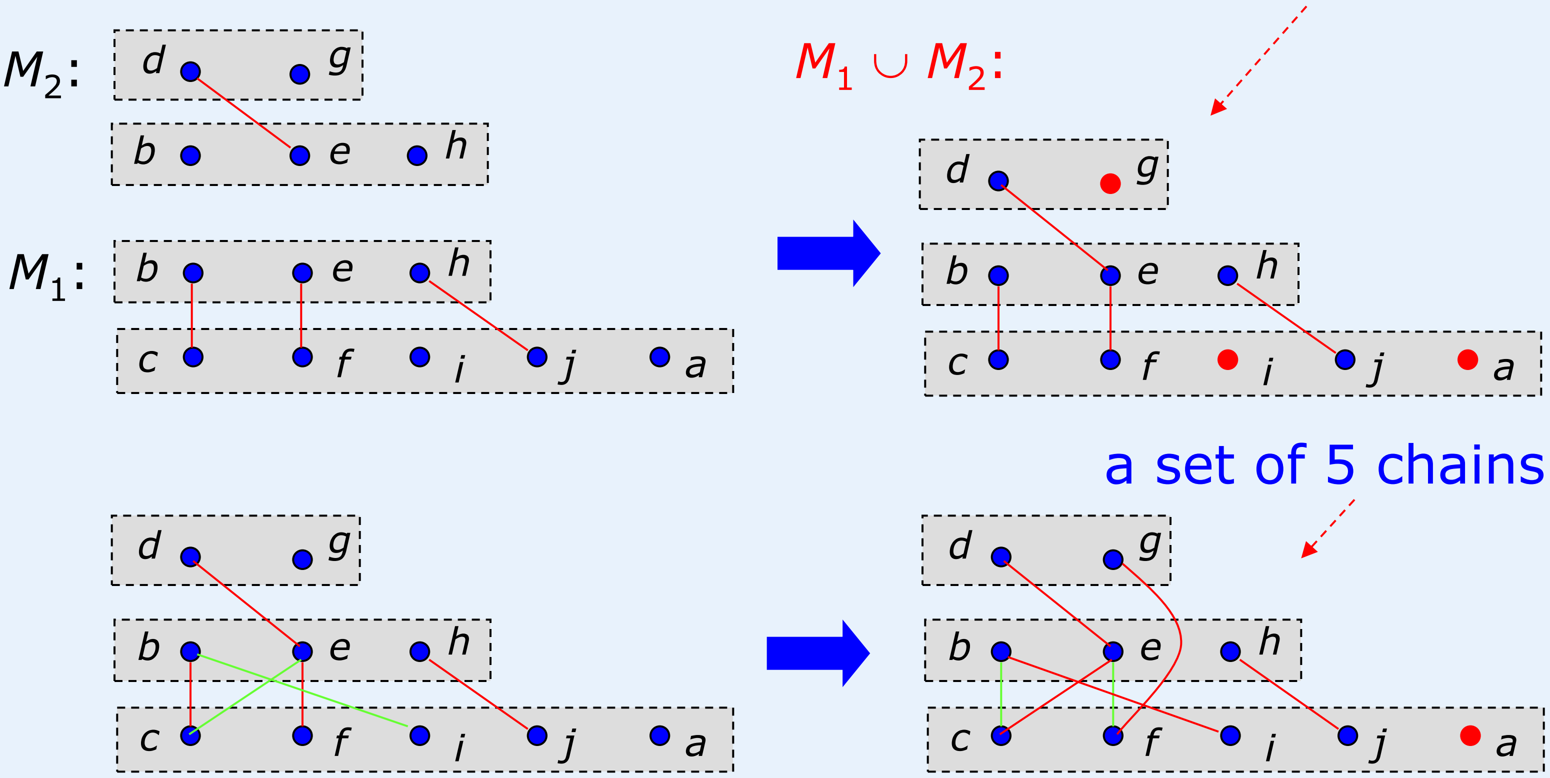
Example.



Transitive Closure Compression

Example.

a set of 6 chains



a set of 5 chains

Virtual Nodes

- $V_i' = V_i \cup$ {virtual nodes introduced into V_i}.
- $\boldsymbol{C}_i = C_j^i \cup$ {all the new edges from the nodes in V_i to the virtual nodes introduced into V_{i-1}}.
- $G(V_i, V_{i-1}', \boldsymbol{C}_i)$ represents the bipartite graph containing V_j and V_{i-1}'.

Transitive Closure Compression

Definition (*virtual nodes for actual nodes*) Let $G(V, E)$ be a DAG, divided into $V_0, ..., V_{h-1}$ (i.e., $V = V_0 \cup ... \cup V_{h-1}$). Let M_i be a maximum matching of the bipartite graph $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$ and v be a free actual node (in V_{i-1}') relative to M_i ($i = 1, ..., h - 1$). Add a virtual node v' into V_i. In addition, for each node $u \in V_{i+1}$, a new edge $u \rightarrow v'$ will be created if one of the following two conditions is satisfied:

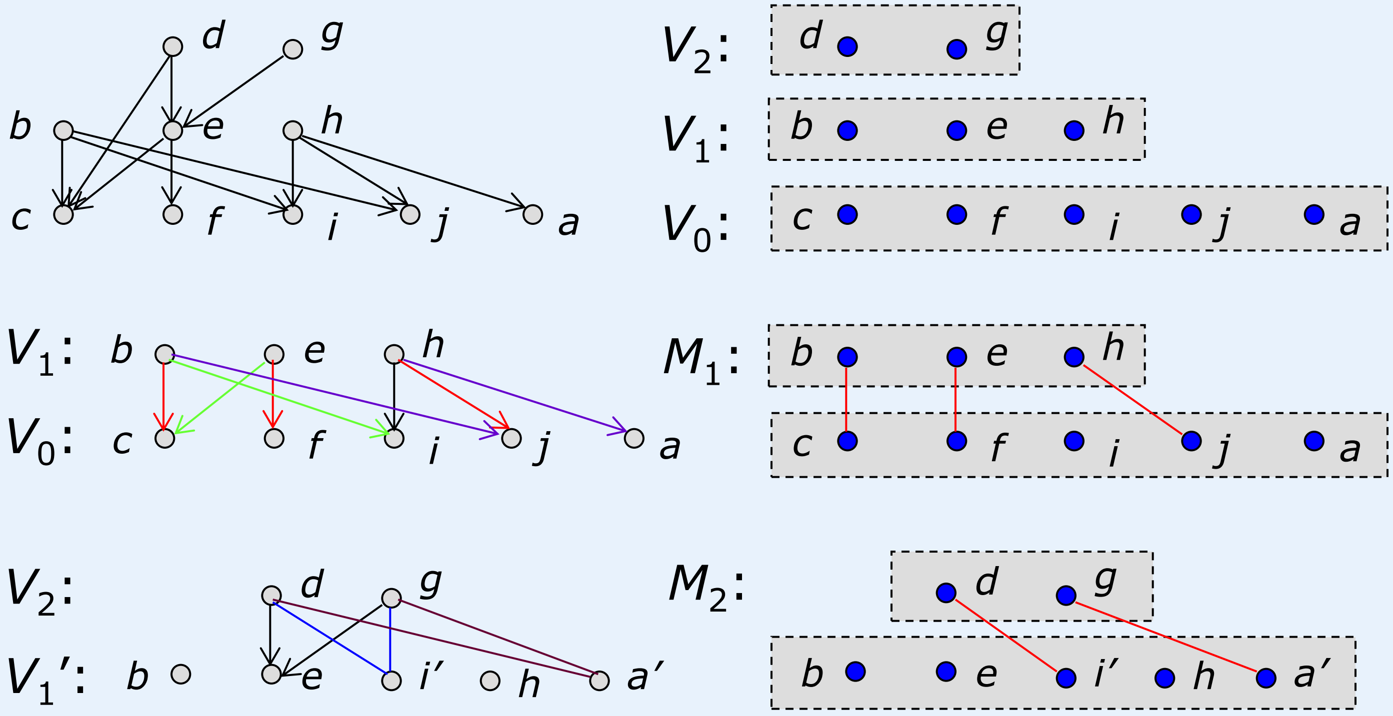
1. $u \rightarrow v \in E$; or
2. There exists an edge (v_1, v_2) covered by M_i such that v_1 and v are connected through an alternating path relative to M_i; and $u \in B_{i+1}(v_1)$ or $u \in B_{i+1}(v_2)$.

v is called the source of v', denoted $s(v')$.

$B_j(v)$ represents a set of links with each pointing to one of v's parents, which appears in V_j.

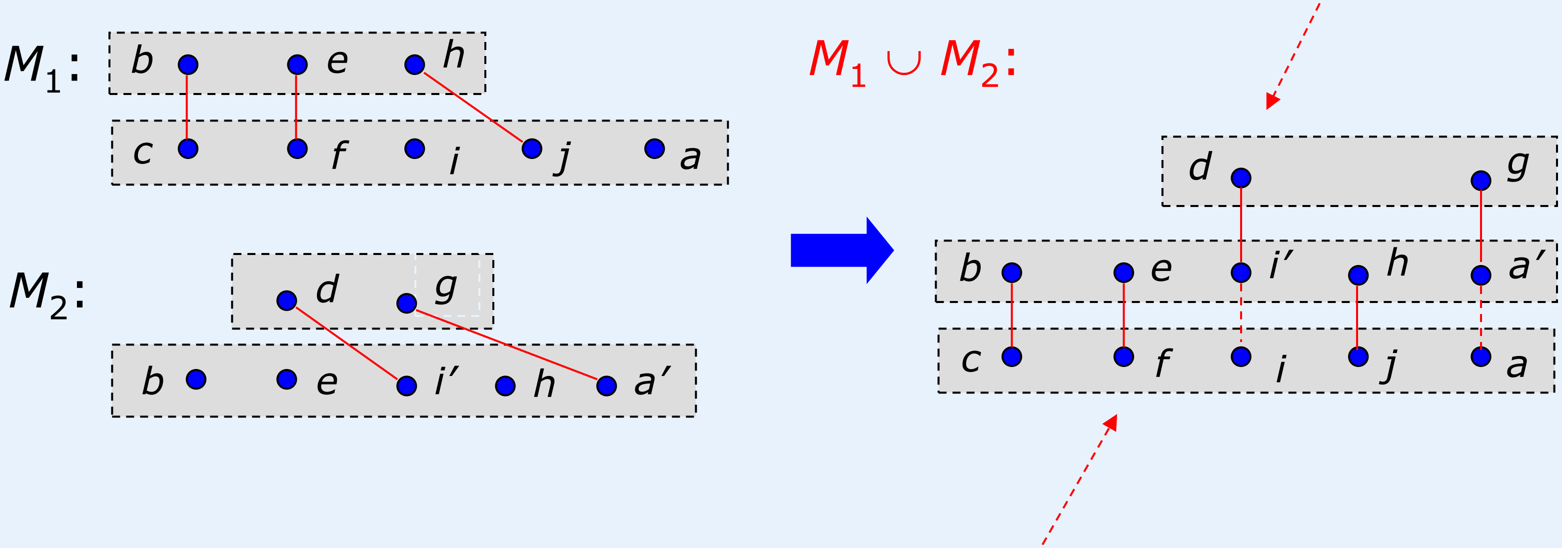
Transitive Closure Compression

Example.



Transitive Closure Compression

Example.



a set of 5 chains

To obtain the final result, the virtual nodes have to be resolved.

Transitive Closure Compression

- **Virtual node resolution**

Definition (*alternating graph*) Let M_i be a maximum matching of $G(V_i, V_{i-1}'; \boldsymbol{C}_i)$. The alternating graph $\vec{G}_i$ with respect to M_i is a directed graph with the following sets of nodes and edges:

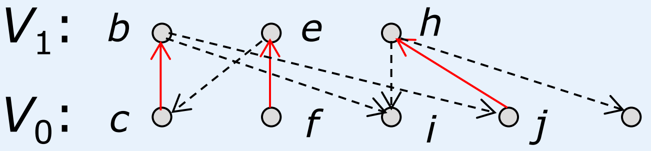
$V(\vec{G}_i) = V_i \cup V_{i-1}'$, and

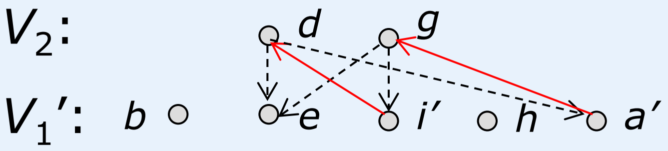
$E(\vec{G}_i) = \{u \rightarrow v \mid u \in V_{i-1}', v \in V_i, \text{and } (u, v) \in M_i\} \cup$

$\{v \rightarrow u \mid u \in V_{i-1}', v \in V_i, \text{and } (u, v) \in \boldsymbol{C}_i \backslash M_i\}.$

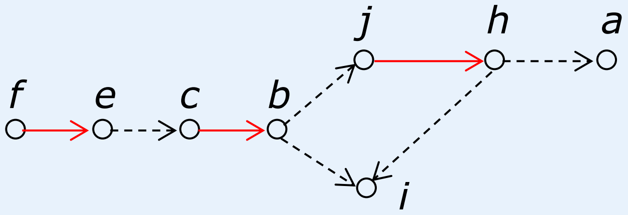
Transitive Closure Compression

Example.

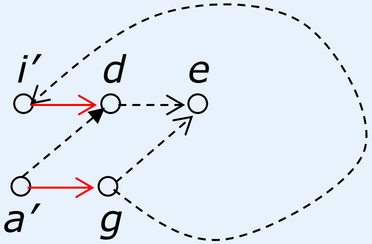




$\vec{G}_1$:



$\vec{G}_2$:



Transitive Closure Compression

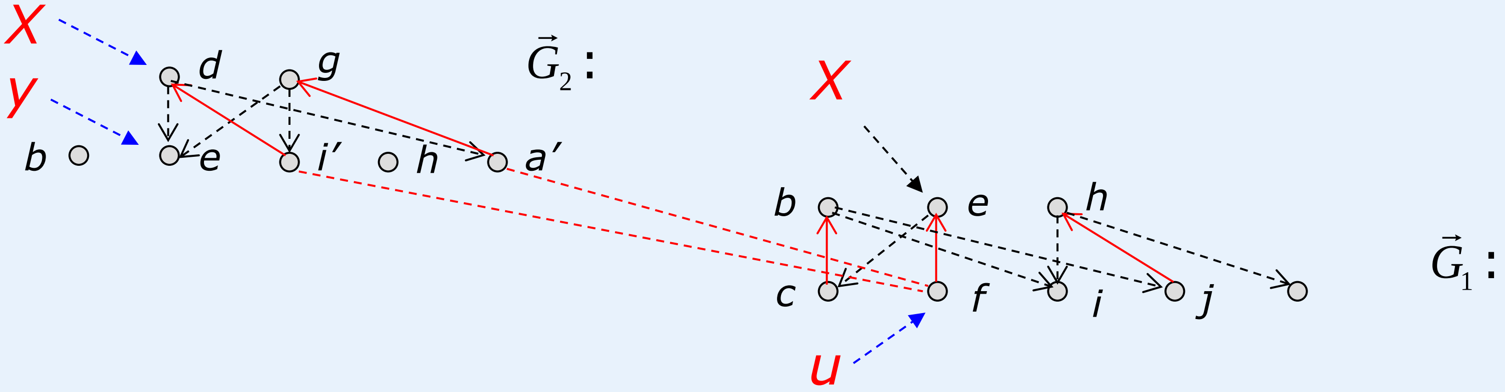
Combined graph:

Combine $\vec{G}_{i+1}$ and $\vec{G}_i$ by connecting some nodes v' in $\vec{G}_{i+1}$ to some nodes u in $\vec{G}_i$ if the following conditions are satisfied.

(i) v' is a virtual node appearing in V_i'.

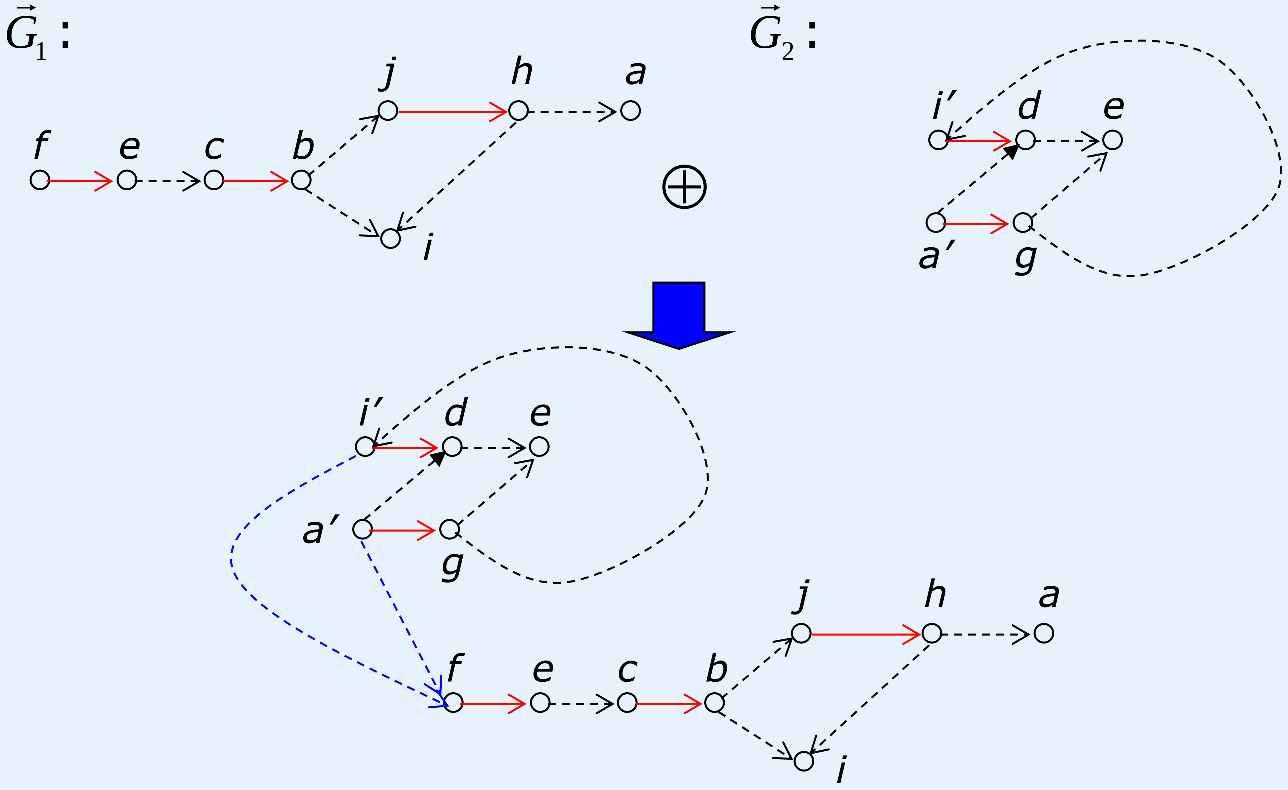
(Note that $V(\vec{G}_{i+1}) = V_{i+1} \cup V_i'$.)

(ii) There exist a node x in V_{i+1} and a node y in V_i such that $(x, v') \in M_{i+1}$, $x \to y \in \boldsymbol{C}_{i+1}$, and $(y, u) \in M_i$.



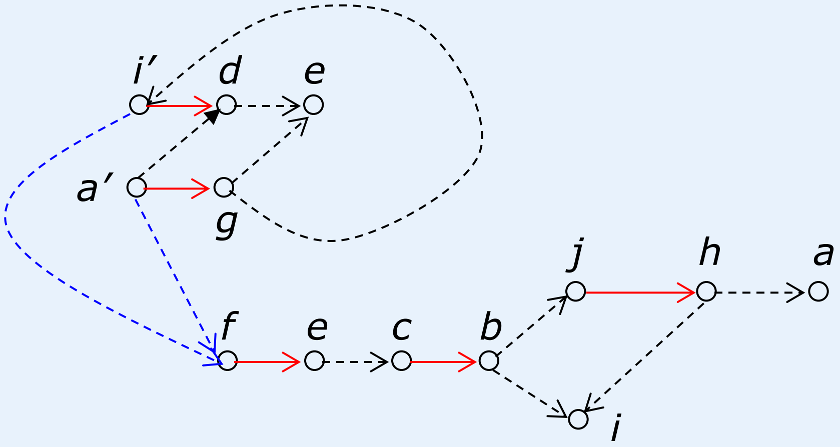
Transitive Closure Compression

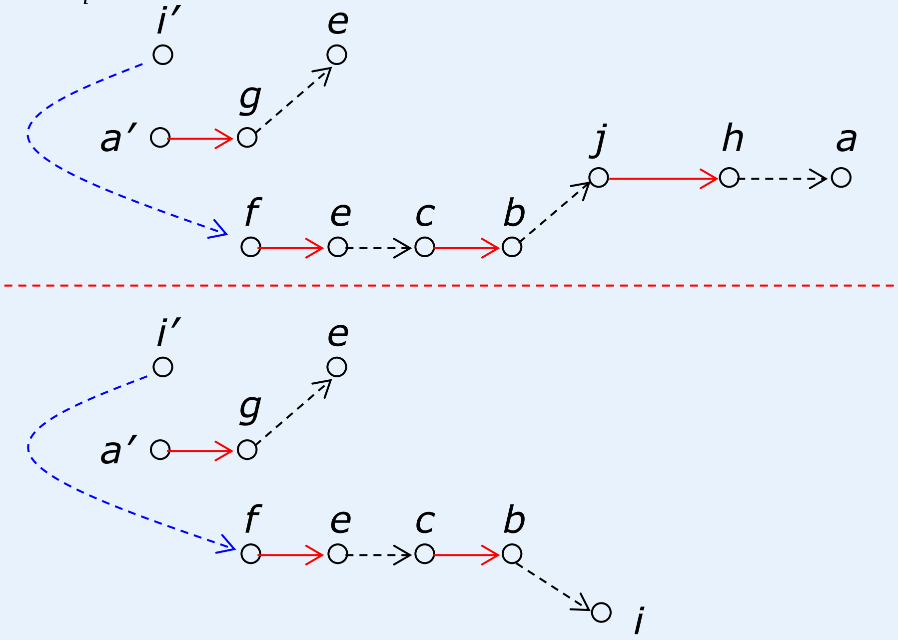
Example.



Transitive Closure Compression

In order to resolve as many virtual nodes (appearing in *V*i′) as possible, we need to find a maximum set of node-disjoint paths (i.e., no two of these paths share any nodes), each starting at virtual node (in $\vec{G}_{i+1}$) and ending at a free node in $\vec{G}_{i+1}$, or ending at a free node in $\vec{G}_i$.



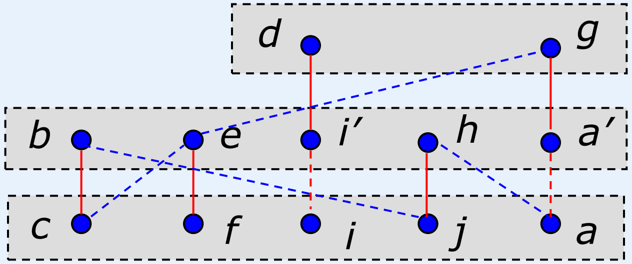


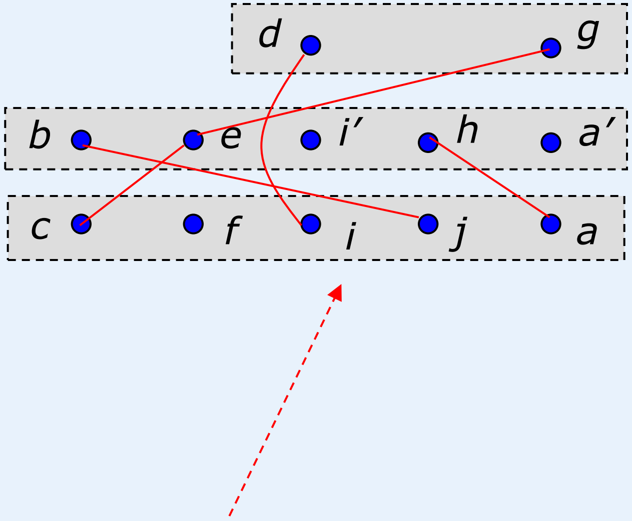
Transitive Closure Compression

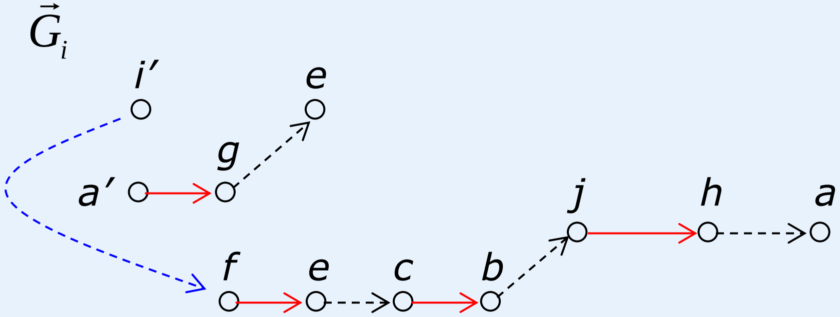
- The problem of finding a maximal set of node-disjoint paths can be solved by transforming it to a maximum flow problem.
- Generally, to find a maximum flow in a network, we need O(n^3) time. However, a network as constructed above is a 0-1 network. In addition, for each node v, we have either $d_{in}(v) \leq 1$ or $d_{out}(v) \leq 1$, where $d_{in}(v)$ and $d_{out}(v)$ represent the indegree and outdegree of v in $\vec{G}_{i+1} \oplus \vec{G}_i$, respectively. It is because each path in $\vec{G}_{i+1} \oplus \vec{G}_i$ is an alternating path relative to M_{i+1} or relative to M_i. So each node except sources and sinks is an end node of an edge covered by M_{i+1} or by M_i. As shown in ([14]), it needs only O($n^{2.5}$) time to find a maximum flow in such kind of networks.

Transitive Closure Compression

$M_1 \cup M_2$:





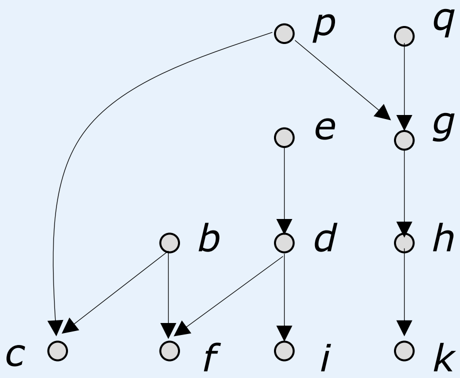


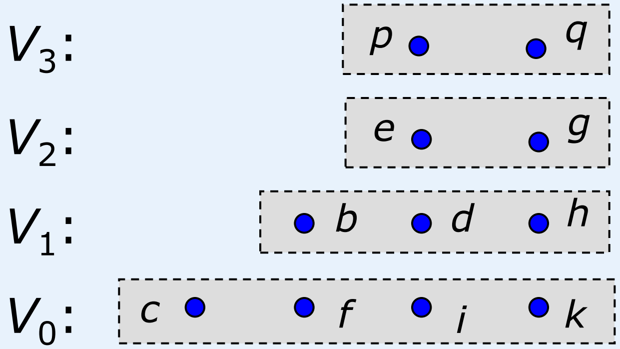
Virtual nodes will be removed.

Transitive Closure Compression

Definition (*virtual nodes for virtual nodes*) Let M_i be a maximum matching of the bipartite graph $G(V_i, V_{i-1}'; \mathbf{C}_i)$ and v' be a free virtual node (in V_{i-1}') relative to M_i ($i = 1, \ldots, h - 1$). Add a virtual node v'' into V_i. Set $s(v'')$ to be $w = s(v')$. Let $l(w) = j$. For each node $u \in V_{i+1}$, a new $u \to v'$ will be created if there exists an edge (v_1, v_2) covered by M_{j+1} such that v_1 and w are connected through an alternating path relative to M_{j+1}; and $u \in B_{i+1}(v_1)$ or $u \in B_{i+1}(v_2)$.

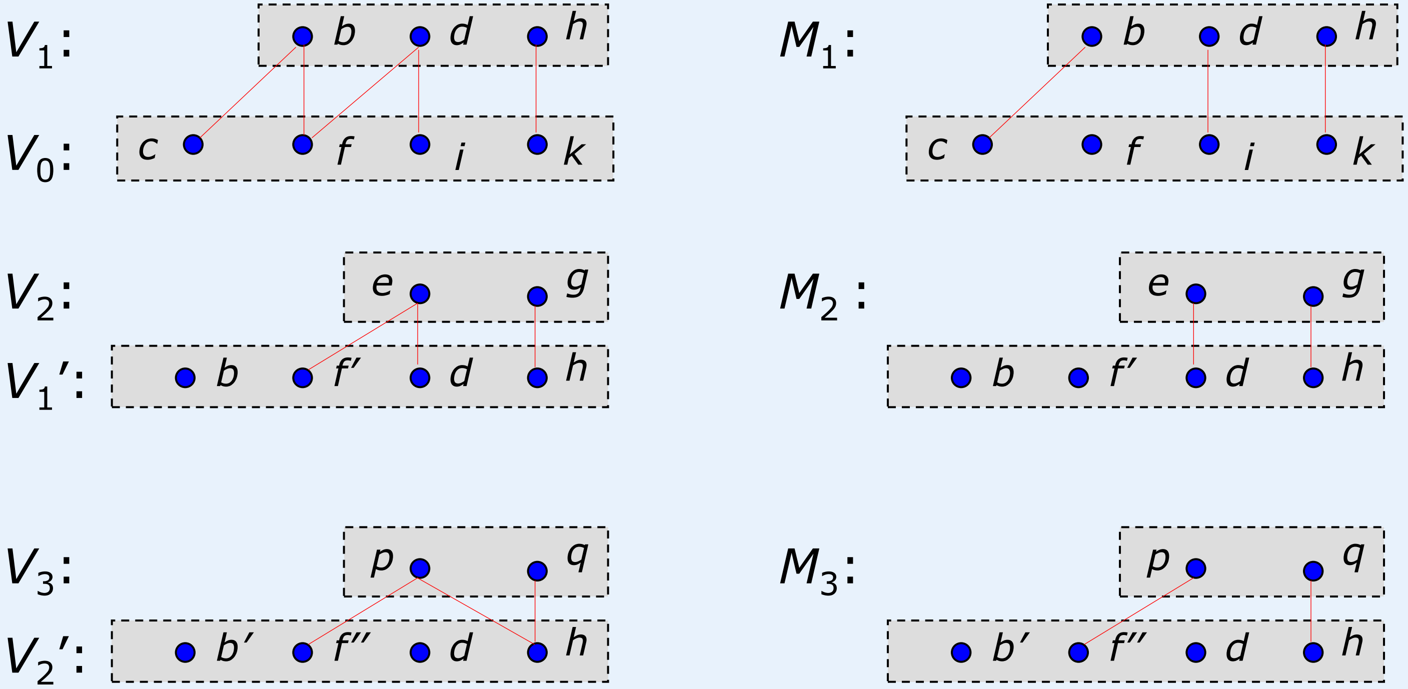
Example.



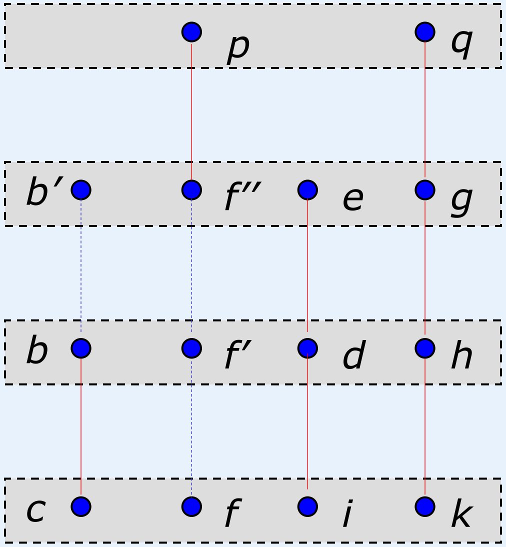


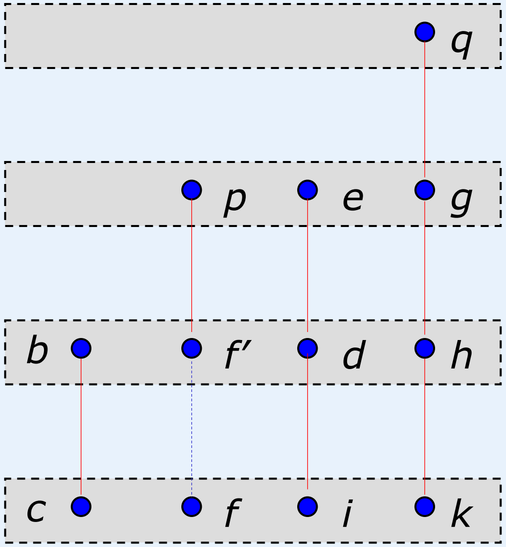
Transitive Closure Compression

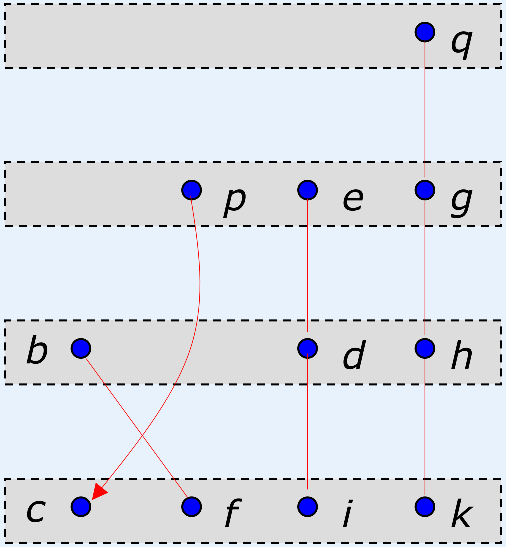
Example.



Transitive Closure Compression







Node-disjoint Paths in Combined Graphs

Now we discuss an algorithm for finding a maximal set of node-disjoint paths in a combined graph $\vec{G}_{i+1} \oplus \vec{G}_i$. Its time complexity is bounded by $O(e \cdot n^{1/2})$, where $n = V(\vec{G}_{i+1} \oplus \vec{G}_i)$ and $e = E(\vec{G}_{i+1} \oplus \vec{G}_i)$. 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 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 $\vec{G}_{i+1} \oplus \vec{G}_i$, 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 $u \rightarrow v$ with $d_{out}(u) = d_{in}(v) = 1$ is called a *bridge*.

Transitive Closure Compression

- Our algorithm works in multiple phases.
- In each phase, the arcs in $\vec{G}_{i+1} \oplus \vec{G}_i$ will be marked or unmarked.
- We also call a virtual node in $\vec{G}_{i+1} \oplus \vec{G}_i$ 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 $\vec{G}_{i+1} \oplus \vec{G}_i$ by *A*.
At the very beginning of the first phase, all the arcs in *A* are unmarked.

Transitive Closure Compression

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 (appearing in).
- Define V_j ($j > 0$) as below:

$$
\begin{aligned}
E_{j\text{-}1} = \{\ & u \rightarrow v \in E(A) \mid u \in V_{j\text{-}1}, v \notin V_0 \cup V_1 \cup \ldots \cup V_{j\text{-}1}, \\
& u \rightarrow v \text{ is unmarked}\} \cup \\
\{\ & v \rightarrow u \in E(A) \mid u \in V_{j\text{-}1}, v \notin V_0 \cup V_1 \cup \ldots \cup V_{j\text{-}1}, \\
& v \rightarrow u \text{ is marked}\}, \\
V_j = \quad \{\ & v \in V(A) \mid \text{for some } u, u \rightarrow v \text{ is unmarked and} \\
& u \rightarrow v \in E_{j\text{-}1}\} \cup \\
\{\ & v \in V(A) \mid \text{for some } u, v \rightarrow u \text{ is marked and} \\
& v \rightarrow u \in E_{j\text{-}1}\}.
\end{aligned}
$$

Transitive Closure Compression

Define $j^* = \min\{j \mid V_j \cap \{\text{unsaturated terminus}\} \neq \phi\}$. (Note that the terminus appearing in $\vec{G}_{i+1}$ are the free nodes relative to M_{i+1}; and those appearing in $\vec{G}_i$ are the free nodes relative to M_i.)

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

If $j^* = 1$, then

$V(A^{(k)}) = V_0 \cup (V_{j^*} \cap \{\text{unsaturated terminus}\})$,

$E(A^{(k)}) = \{u \rightarrow v \mid u \in V_{j^*\text{-}1}, \text{ and } v \in \{\text{unsaturated terminus}\}\}$.

If $j^* > 1$, then

$V(A^{(k)}) = V_0 \cup V_1 \cup \ldots \cup V_{j^*\text{-}1} \cup (V_{j^*} \cap \{\text{unsaturated terminus}\})$,

$E(A^{(k)}) = E_0 \cup E_1 \cup \ldots \cup E_{j^*\text{-}2} \cup \{u \rightarrow v \mid u \in E_{j^*\text{-}1}, \text{ and }$
$v \in \{\text{unsaturated terminus}\}\}$.

The sets V_j are called *levels*.

Transitive Closure Compression

In , 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 $A^{(k)}$, or a marked arc $v_{j+1} \rightarrow v_j$ in $A^{(k)}$.

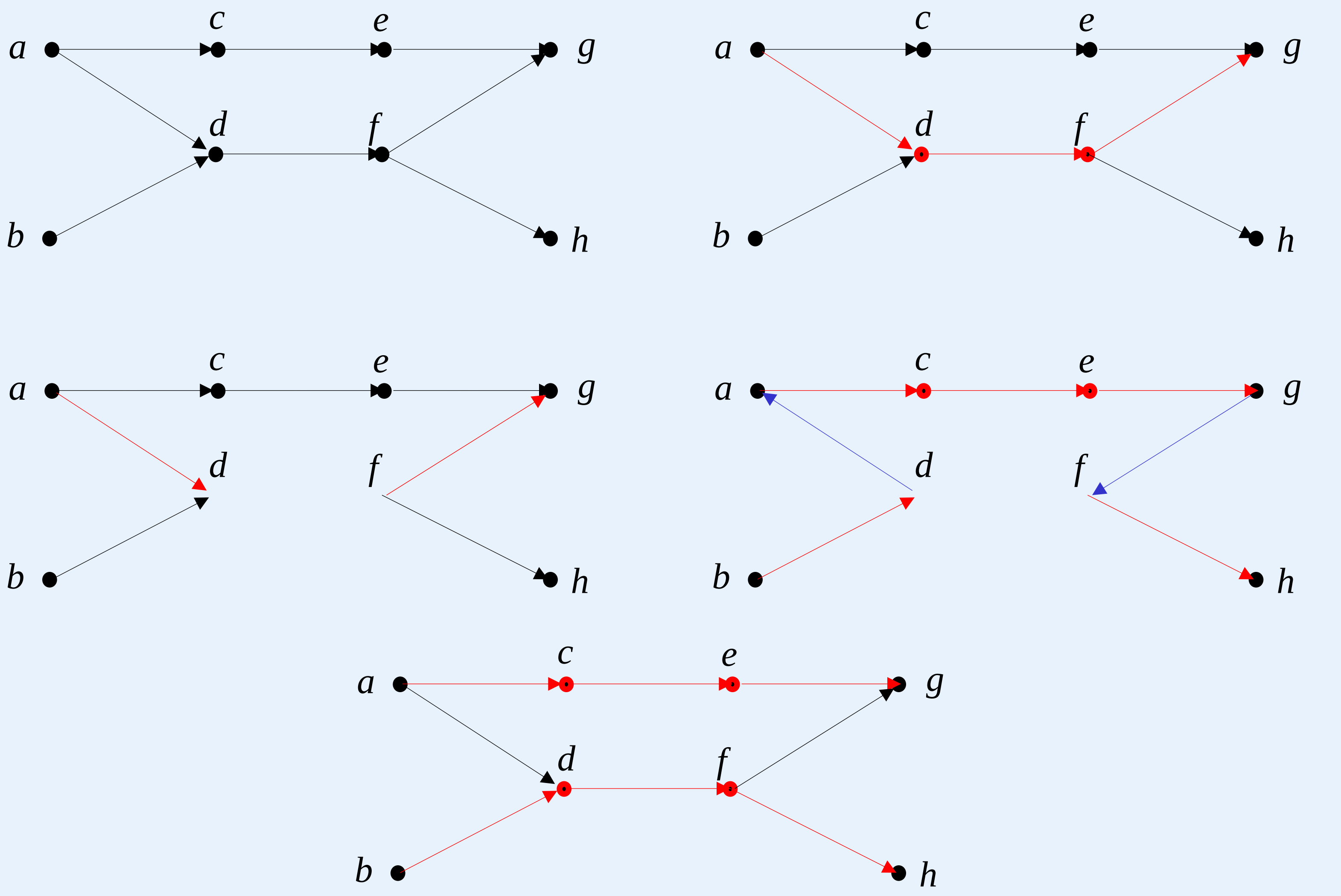
Our algorithm will explore to find a set of node-disjoint complete sequences (i.e., no two of them share any nodes.) Then, we mark and unmark the arcs along each complete sequence as follows.

(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 $A^{(k)}$ there exists j such that $V_j = \Phi$ and $V_i \cap$ {unsaturated terminus} = Φ 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.

Transitive Closure Compression



Transitive Closure Compression

```
Algorithm subgraph-exploring()
begin
let $v$ be the first element in $V0$;
push($v$, $H$); mark $v$ 'accessed';
while $H$ is not empty do {
    $v$ := top($H$); (*the top element of $H$ is assigned to $v$.*)
    while neighbor($v$) $\neq$ F do {
        let $u$ be the first element in neighbor($v$);
        if $u$ is accessed then remove $u$ from neighbor($v$)
        else {push($u$, $H$); mark $u$ 'accessed'; $v$ := $u$;}
}
    if $v$ is neither in $V_j^*$ nor in $V_0$ then pop($H$)
    else {if $v$ is in $V_j^*$ then output all the elements in $H$;
(*all the elements in $H$ make up a complete sequence.*)
            remove all elements in $H$;
            let $v$ be the next element in $V_0$;
            push($v$, $H$); mark $v$;
}
end
```

Transitive Closure Compression

```
Algorithm node-disjoint-paths($A$)
begin
  $k$ := 1;
  construct $A^{(1)}$ ;
  while $A^{(k)} \neq \Phi$ do {
      call subgraph-exploring($A^{(k)}$);
      let $P_1$, ... $P_l$ be all the found complete sequences;
      for $j$ = 1 to $l$ do
      {  let $P_j = v_1, v_2, \ldots , v_m$;
         mark $v_i \rightarrow v_{i+1}$ or unmark $v_{i+1} \rightarrow v_i$ ($i$ = 1, ..., $m$ - 1)
         according to (i) and (ii) above;
      }
      $k$ := $k$ + 1; construct $A^{(k)}$;
  }
end
```