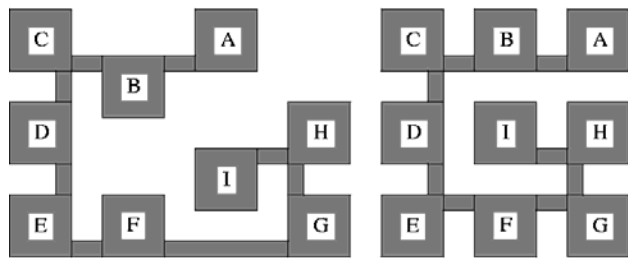


## Unit 8: Layout Compaction

- Course contents
  - Design rules
  - Symbolic layout
  - Constraint-graph compaction
- Readings: Chapter 6



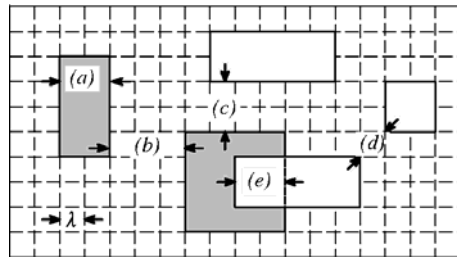
Unit 8

NTUEE / ntro. EDA

1

## Design Rules

- **Design rules:** restrictions on the mask patterns to increase the probability of successful fabrication.
- Patterns and design rules are often expressed in  $\lambda$  rules.
- Most common design rules:



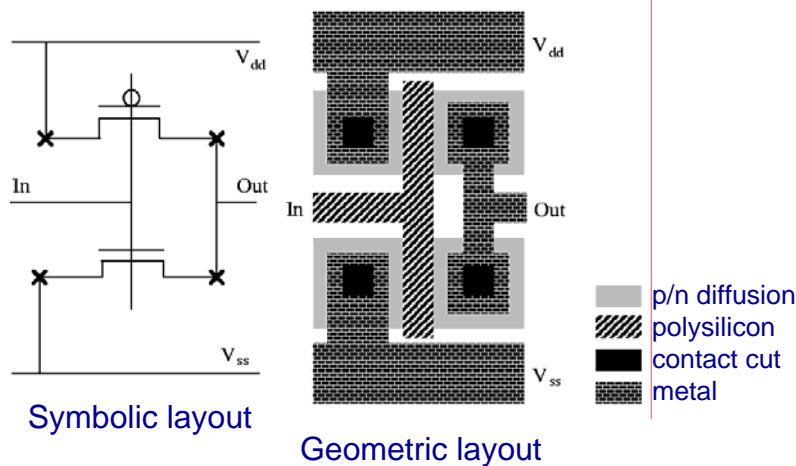
- minimum-width rules (valid for a mask pattern of a specific layer): (a).
- minimum-separation rules (between mask patterns of the same layer or different layers): (b), (c).
- minimum-overlap rules (mask patterns in different layers): (e).

Unit 8

NTUEE / ntro. EDA

2

## CMOS Inverter Layout Example



Unit 8

NTUEE / ntro. EDA

3

## Symbolic Layout

- **Geometric (mask) layout:** coordinates of the layout patterns (rectangles) are absolute (or in multiples of  $\lambda$ ).
- **Symbolic (topological) layout:** only relations between layout elements (below, left to, etc) are known.
  - Single symbols are used to represent elements located in several layers, e.g. transistors, contact cuts.
  - The *length*, *width* or *layer* of a wire or another layout element might be left unspecified.
  - Mask layers not directly related to the functionality of the circuit do not need to be specified, e.g. n-well, p-well.
- The symbolic layout can work with a technology file that contains all design rule information for the target technology to produce the geometric layout.

Unit 8

NTUEE / ntro. EDA

4

## Compaction and Its Applications

---

- A **compaction program** or **compactor** generates layout at the mask level. It attempts to make the layout as dense as possible.
- Applications of compaction:
  - **Area minimization**: remove redundant space in layout at the mask level.
  - **Layout compilation**: generate mask-level layout from symbolic layout.
  - **Redesign**: automatically remove design-rule violations.
  - **Rescaling**: convert mask-level layout from one technology to another.

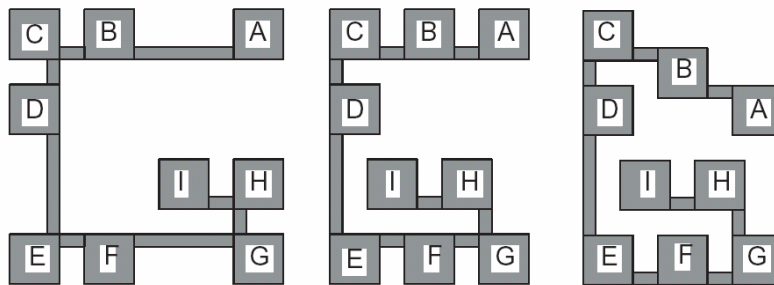
## Aspects of Compaction

---

- Dimension:
  - 1-dimensional (1D) compaction: layout elements only are moved or shrunk in one dimension (x or y direction).
    - Is often performed first in the x-dimension and then in the y-dimension (or vice versa).
  - 2-dimensional (2D) compaction: layout elements are moved and shrunk simultaneously in two dimensions.
- Complexity:
  - 1D compaction can be done in polynomial time.
  - 2D compaction is NP-hard.

## 1D Compaction: X Followed By Y

- Each square is  $2\lambda * 2\lambda$ , minimum separation is  $1\lambda$ .
- Initially, the layout is  $11\lambda * 11\lambda$ .
- After compacting along the **x** direction, then the **y** direction, we have the layout size of  $8\lambda * 11\lambda$ .



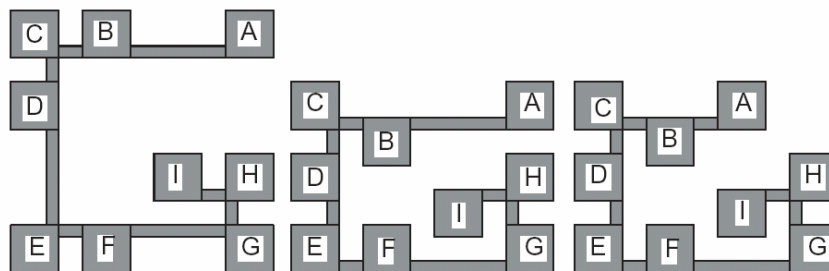
Unit 8

NTUEE / ntro. EDA

7

## 1D Compaction: Y Followed By X

- Each square is  $2\lambda * 2\lambda$ , minimum separation is  $1\lambda$ .
- Initially, the layout is  $11\lambda * 11\lambda$ .
- After compacting along the **y** direction, then the **x** direction, we have the layout size of  $11\lambda * 8\lambda$ .



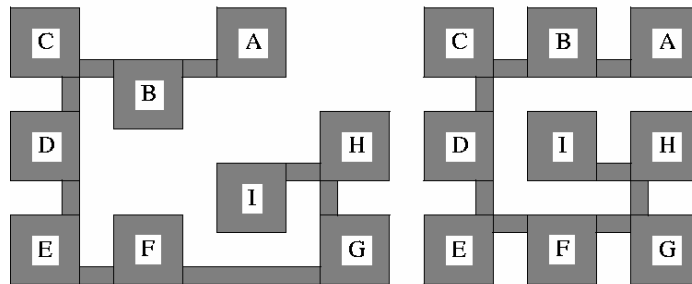
Unit 8

NTUEE / ntro. EDA

8

## 2D Compaction

- Each square is  $2\lambda \times 2\lambda$ , minimum separation is  $1\lambda$ .
- Initially, the layout is  $11\lambda \times 11\lambda$ .
- After **2D compaction**, the layout size is only  $8\lambda \times 8\lambda$ .



- Since 2D compaction is NP-complete, most compactors are based on repeated 1D compaction.

Unit 8

NTUEE / ntro. EDA

9

## Inequalities for Distance Constraints

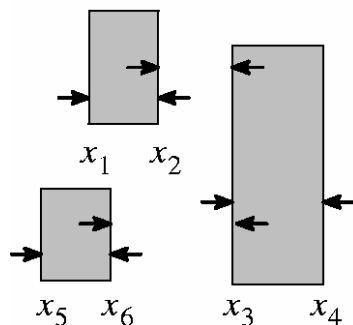
- Minimum-distance design rules can be expressed as inequalities.
- For example, if the minimum width is  $a$  and the minimum separation is  $b$ , then

$$x_j - x_i \geq d_{ij}$$

$$x_2 - x_1 \geq a$$

$$x_3 - x_2 \geq b$$

$$x_3 - x_6 \geq b$$



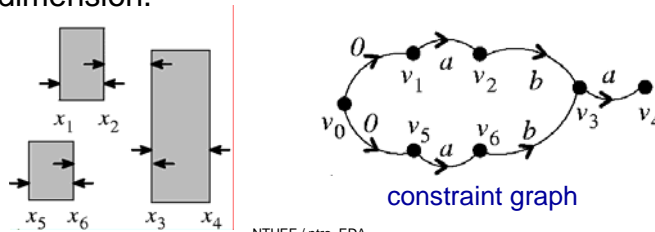
Unit 8

NTUEE / ntro. EDA

10

## Constraint Graph

- The inequalities can be used to construct a constraint graph  $G(V, E)$ :
  - There is a vertex  $v_i$  for each variable  $x_i$ ,
  - For each inequality  $x_j - x_i \geq d_{ij}$  there is an edge  $(v_i, v_j)$  with weight  $d_{ij}$ .
  - There is an extra source vertex,  $v_0$ ; it is located at  $x = 0$ ; all other vertices are at its right.
- If all the inequalities express minimum-distance constraints, the graph is **acyclic** (DAG).
- The longest path in a constraint graph determines the layout dimension.



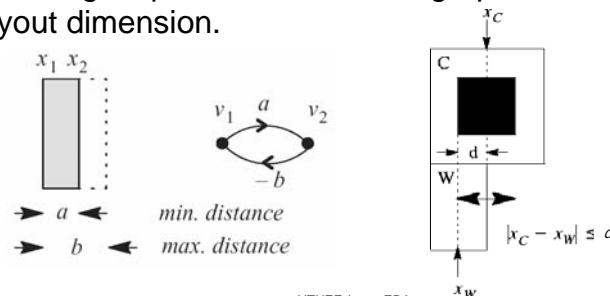
Unit 8

NTUEE / intro. EDA

11

## Maximum-Distance Constraints

- Sometimes the distance of layout elements is bounded by a maximum, e.g., when the user wants a maximum wire width, maintains a wire connecting to a via, etc.
  - A maximum distance constraint gives an inequality of the form:  $x_j - x_i \leq c_{ij}$  OR  $x_i - x_j \geq -c_{ij}$
  - Consequence for the constraint graph: **backward edge**
    - $(v_j, v_i)$  with weight  $d_{ji} = -c_{ij}$ ; the graph is not acyclic anymore.
- The longest path in a constraint graph determines the layout dimension.



Unit 8

NTUEE / intro. EDA

12

## Longest-Path Algorithm for DAGs

```

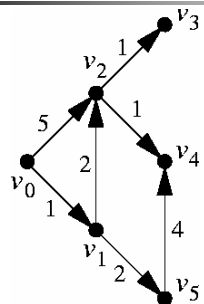
longest-path( $G$ )
{
  for ( $i \leftarrow 1; i \leq n; i \leftarrow i + 1$ )
     $p_i \leftarrow$  "in-degree of  $v_i$ ";
   $Q \leftarrow \{v_0\}$ ;
  while ( $Q \neq \emptyset$ ) {
     $v_i \leftarrow$  "any element from  $Q$ ";
     $Q \leftarrow Q \setminus \{v_i\}$ ;
    for each  $v_j$  "such that"  $(v_i, v_j) \in E$  {
       $x_j \leftarrow \max(x_j, x_i + d_{ij})$ ;
       $p_j \leftarrow p_j - 1$ ;
      if ( $p_j \leq 0$ )
         $Q \leftarrow Q \cup \{v_j\}$ ;
    }
  }
}

main ()
{
  for ( $i \leftarrow 0; i \leq n; i \leftarrow i + 1$ )
     $x_i \leftarrow 0$ ;
  longest-path( $G$ );
}

```

- $p_i$ : in-degree of  $v_i$ .
- $x_i$ : longest-path length from  $v_0$  to  $v_i$ .

## DAG Longest-Path Example



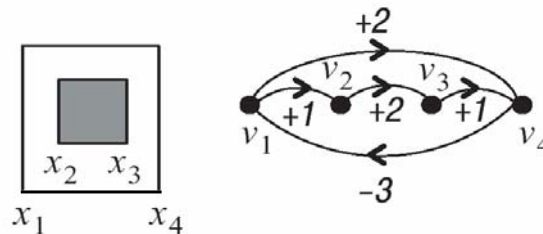
- Runs in a breadth-first search manner.
- $p_i$ : in-degree of  $v_i$ .
- $x_i$ : longest-path length from  $v_0$  to  $v_i$ .
- Time complexity:  $O(V+E)$ .

$Q$	$p_1$	$p_2$	$p_3$	$p_4$	$p_5$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$
"not initialized"	1	2	1	2	1	0	0	0	0	0
$\{v_0\}$	0	1	1	2	1	1	5	0	0	0
$\{v_1\}$	0	0	1	2	0	1	5	0	0	3
$\{v_2, v_5\}$	0	0	0	1	0	1	5	6	6	3
$\{v_3, v_5\}$	0	0	0	1	0	1	5	6	6	3
$\{v_5\}$	0	0	0	0	0	1	5	6	7	3
$\{v_4\}$	0	0	0	0	0	1	5	6	7	3

## Longest-Paths In Cyclic Graphs

---

- Constraint-graph compaction with maximum-distance constraints requires solving the longest-path problem in cyclic graphs.
- Two cases are distinguished:
  - There are positive cycles: No feasible solution for longest paths. We shall detect the cycles.
  - All cycles are negative: Polynomial-time algorithms exist.



Unit 8

NTUEE / ntro. EDA

15

## The Liao-Wong Longest-Path Algorithm

---

- Split the edge set  $E$  of the constraint graph into two subsets:
  - Forward edges  $E_f$ : related to minimum-distance constraints.
  - Backward edges  $E_b$ : related to maximum-distance constraints.
- The graph  $G(V, E_f)$  is acyclic; the longest distance for each vertex can be computed with the procedure “longest-path”.
- Repeat :
  - Update longest distances by processing the edges from  $E_b$ .
  - Call “longest-path” for  $G(V, E_f)$ .
- Worst-case time complexity:  $O(E_b \times E_f)$ .

Unit 8

NTUEE / ntro. EDA

16



## Pseudo Code: The Liao-Wong Algorithm

```

count ← 0;
for (i ← 1; i ≤ n; i ← i + 1)
    xi ← -∞;
x0 ← 0;

do { flag ← 0;
    longest-path(Gf);
    for each (vi, vj) ∈ Eb
        if (xj < xi + dij) {
            xj ← xi + dij;
            flag ← 1;
        }
    count ← count + 1;
    if (count > |Eb| && flag)
        error("positive cycle")
}
while (flag);

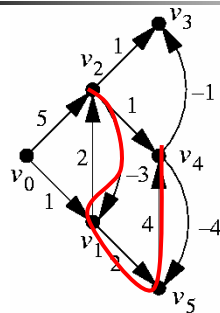
```

Unit 8

NTUEE / intro. EDA

17

## Example for the Liao-Wong Algorithm



- Two edge sets: forward edges  $E_f$  and backward edges  $E_b$
- $x_i$ : longest-path length from  $v_0$  to  $v_i$ .
- Call "longest-path" for  $G(V, E_f)$ .
- Update longest distances by processing the edges from  $E_b$ .
- Time complexity:  $O(E_b \times E_f)$ .

$$x_1 < x_2 - 3$$

$$x_3 < x_4 - 1$$

$$x_5 = x_4 - 4$$

Step	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$
Initialize	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
Forward 1	1	5	6	7	3
Backward 1	2	5	6	7	3
Forward 2	2	5	6	8	4
Backward 2	2	5	7	8	4
Forward 3	2	5	7	8	4
Backward 3	2	5	7	8	4

Unit 8

NTUEE / intro. EDA

18

## The Bellman-Ford Algorithm for Longest Paths

```

for (i ← 1; i ≤ n; i ← i + 1)
  xi ← -∞; /* n: upper bound of vertex ID */
x0 ← 0; /* n+1: total # of vertices */
count ← 0;
S1 ← {v0}; /* current wave front */
S2 ← ∅; /* next wave front */
while (count ≤ n && S1 ≠ ∅) {
  for each vi ∈ S1
    for each vj "such that" (vi, vj) ∈ E
      if (xj < xi + dij) {
        xj ← xi + dij;
        S2 ← S2 ∪ {vj}
      }
  S1 ← S2;
  S2 ← ∅;
  count ← count + 1;
}
if (count > n)
  error("positive cycle");

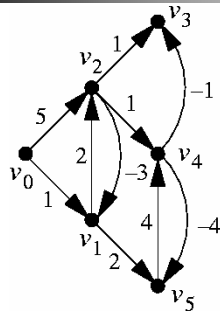
```

Unit 8

NTUEE / intro. EDA

19

## Example of Bellman-Ford for Longest Paths



- Repeated "wave front propagation."
- $S_1$ : the current wave front.
- $x_i$ : longest-path length from  $v_0$  to  $v_i$ .
- After  $k$  iterations, it computes the longest-path values for paths going through  $k-1$  intermediate vertices.
- Time complexity:  $O(VE)$ .

$S_1$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$
"not initialized"	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
{ $v_0$ }	1	5	$-\infty$	$-\infty$	$-\infty$
{ $v_1, v_2$ }	2	5	6	6	3
{ $v_1, v_3, v_4, v_5$ }	2	5	6	7	4
{ $v_4, v_5$ }	2	5	6	8	4
{ $v_4$ }	2	5	7	8	4
{ $v_3$ }	2	5	7	8	4

Unit 8

NTUEE / intro. EDA

20

## Longest and Shortest Paths

- Longest paths become shortest paths and vice versa when edge weights are multiplied by  $-1$ .
- Situation in DAGs: both the longest and shortest path problems can be solved in **linear** time.
- Situation in cyclic directed graphs:
  - All weights are positive: shortest-path problem in P (Dijkstra), no feasible solution for the longest-path problem.
  - All weights are negative: longest-path problem in P (Dijkstra), no feasible solution for the shortest-path problem.
  - No positive cycles: longest-path problem is in P.
  - No negative cycles: shortest-path problem is in P.

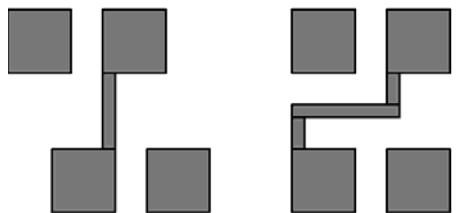
Unit 8

NTUEE / ntro. EDA

21

## Remarks on Constraint-Graph Compaction

- **Noncritical layout elements:** Every element outside the **critical paths** has freedom on its best position => may use this freedom to optimize some cost function.
- **Automatic jog insertion:** The quality of the layout can further be improved by automatic **jog insertion**.



- **Hierarchy:** A method to reduce complexity is hierarchical compaction, e.g., consider cells only.

Unit 8

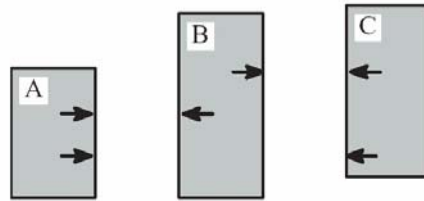
NTUEE / ntro. EDA

22

## Constraint Generation

---

- The set of constraints should be irredundant and generated efficiently.
- An edge  $(v_i, v_j)$  is **redundant** if edges  $(v_i, v_k)$  and  $(v_k, v_j)$  exist and  $w((v_i, v_j)) \leq w((v_i, v_k)) + w((v_k, v_j))$ .
  - The minimum-distance constraints for (A, B) and (B, C) make that for (A, C) redundant.



- Doenhardt and Lengauer have proposed a method for irredundant constraint generation with complexity  $O(n \log n)$ .

## Appendix: Dijkstra's Shortest-Path Algorithm

---

```

Dijkstra( $G, w, s$ )
1. Initialize-Single-Source( $G, s$ );
2.  $S \leftarrow \emptyset$ ;
3.  $Q \leftarrow V[G]$ ;
4. while  $Q \neq \emptyset$ 
5.    $u \leftarrow \text{Extract-Minimum}(Q)$ ;
6.    $S \leftarrow S \cup \{u\}$ ;
7.   for each vertex  $v \in \text{Adj}[u]$ 
8.     Relax( $u, v, w$ );
    
```

- Combines a greedy and a dynamic-programming schemes.
- Works only when all **edge weights are nonnegative**.
- Executes essentially the same as Prim's algorithm.
- Naive analysis:  $O(V^2)$  time by using adjacency lists.
- Can be done in  $O(E \lg V)$  time ( $Q$ : a binary heap) or  $O(E + V \lg V)$  time ( $Q$ : a Fibonacci heap)

## Relaxation

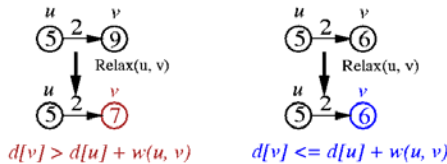
Initialize-Single-Source( $G, s$ )

1. **for** each vertex  $v \in V[G]$
2.  $d[v] \leftarrow \infty$ ;  
/\* upper bound on the weight of a shortest path from  $s$  to  $v$  \*/
3.  $\pi[v] \leftarrow \text{NIL}$ ; /\* predecessor of  $v$  \*/
4.  $d[s] \leftarrow 0$ ;

$\text{Relax}(u, v, w)$

1. **if**  $d[v] > d[u] + w(u, v)$
2.  $d[v] \leftarrow d[u] + w(u, v)$ ;
3.  $\pi[v] \leftarrow u$ ;

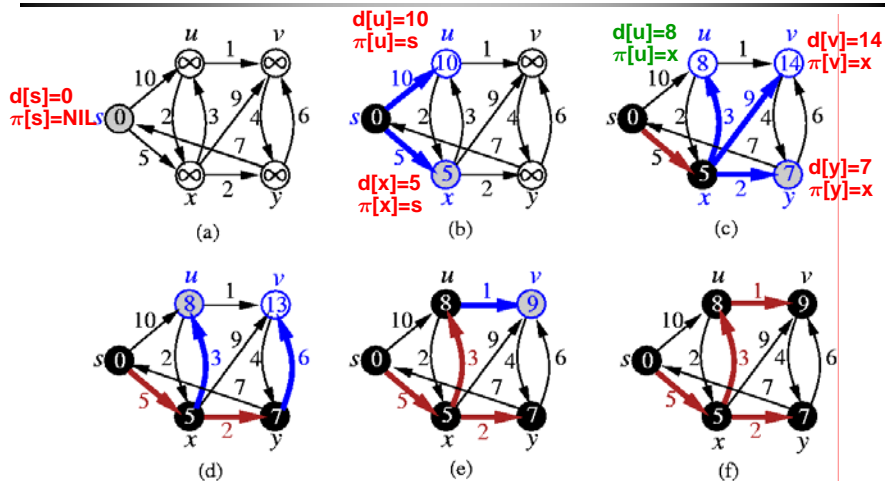
- $d[v] \leq d[u] + w(u, v)$  after calling  $\text{Relax}(u, v, w)$ .
- $d[v] \geq \delta(s, v)$  during the relaxation steps; **once  $d[v]$  achieves its lower bound  $\delta(s, v)$ , it never changes.** ( $\delta(s, v)$ ; shortest path distance from the source  $s$  to  $v$ .)
- Let  $s \rightsquigarrow u \rightarrow v$  be a shortest path. If  $d[u] = \delta(s, u)$  prior to the call  $\text{Relax}(u, v, w)$ , then  $d[v] = \delta(s, v)$  after the call.



Unit 8

25

## Example: Dijkstra's Shortest-Path Algorithm



Unit 8

NTUEE / ntro. EDA

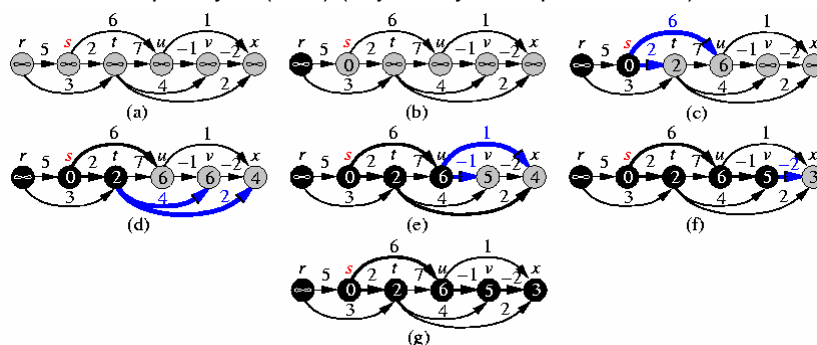
26

## Shortest Path for Directed Acyclic Graphs (DAGs)

DAG-Shortest-Paths( $G, w, s$ )

1. topologically sort the vertices of  $G$ ;
2. Initialize-Single-Source( $G, s$ );
3. **for** each vertex  $u$  taken in topologically sorted order
4.   **for** each vertex  $v \in Adj[u]$
5.     Relax( $u, v, w$ );

- Time complexity:  $O(V+E)$  (adjacency-list representation).



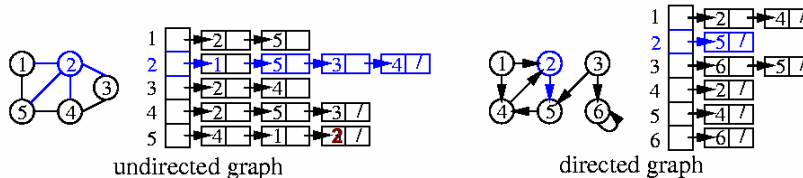
Unit 8

NTU EE / intro. EDA

27

## Representations of Graphs: Adjacency List

- **Adjacency list:** An array  $Adj$  of  $|V|$  lists, one for each vertex in  $V$ . For each  $u \in V$ ,  $Adj[u]$  pointers to all the vertices adjacent to  $u$ .
- Advantage:  $O(V+E)$  storage, good for **sparse** graph.
- Drawback: Need to traverse list to find an edge.



Unit 8

NTU EE / intro. EDA

28

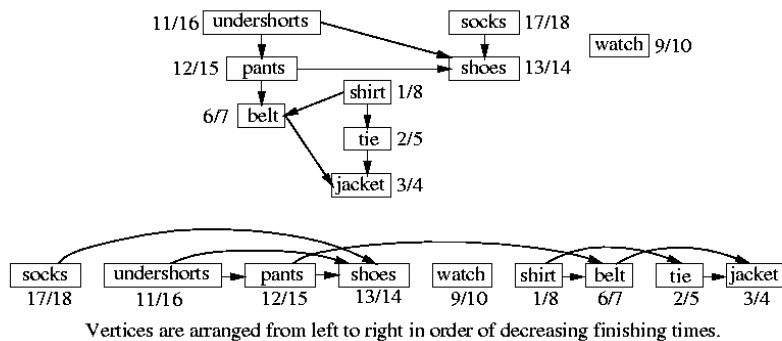
## Topological Sort

- A **topological sort** of a **directed acyclic graph** (DAG)  $G = (V, E)$  is a linear ordering of  $V$  s.t.  $(u, v) \in E \Rightarrow u$  appears before  $v$ .

Topological-Sort( $G$ )

- call DFS( $G$ ) to compute finishing times  $f[v]$  for each vertex  $v$
- as each vertex is finished, insert it onto the front of a linked list
- return** the linked list of vertices

- Time complexity:  $O(V+E)$  (adjacency list).



Unit 8

NTUEE / intro. EDA

29

## Depth-First Search (DFS)

DFS( $G$ )

- for each vertex  $u \in V[G]$
- $color[u] \leftarrow WHITE$ ;
- $\pi[u] \leftarrow NIL$ ;
- $time \leftarrow 0$ ;
- for each vertex  $u \in V[G]$
- if  $color[u] = WHITE$
- DFS-Visit( $u$ ).

DFS-Visit( $u$ )

- $color[u] \leftarrow GRAY$ ;  
/\* white vertex  $u$  has just been discovered. \*/
- $d[u] \leftarrow time \leftarrow time + 1$ ;
- for each vertex  $v \in Adj[u]$   
/\* Explore edge  $(u, v)$ . \*/
- if  $color[v] = WHITE$
- $\pi[v] \leftarrow u$ ;
- DFS-Visit( $v$ );
- $color[u] \leftarrow BLACK$ ;  
/\* Blacken  $u$ ; it is finished. \*/
- $f[u] \leftarrow time \leftarrow time + 1$ .

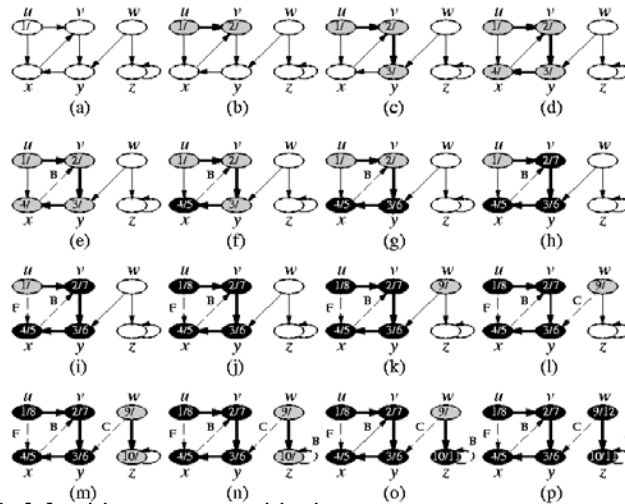
- $color[u]$ : white (undiscovered)  $\rightarrow$  gray (discovered)  $\rightarrow$  black (explored: out edges are all discovered)
- $d[u]$ : discovery time (gray);  
 $f[u]$ : finishing time (black);  
 $\pi[u]$ : predecessor.
- Time complexity:  $O(V+E)$  (adjacency list).

Unit 8

NTUEE / intro. EDA

30

## DFS Example



- $color[u]$ : white  $\rightarrow$  gray  $\rightarrow$  black.
- Depth-first **forest**:  $G_\pi = (V, E_\pi)$ ,  $E_\pi = \{(\pi[v], v) \in E \mid v \in V, \pi[v] \neq NIL\}$ .

Unit 8

NTUEE / intro. EDA

31

## The Bellman-Ford Algorithm for Shortest Paths

```

Bellman-Ford( $G, w, s$ )
1. Initialize-Single-Source( $G, s$ );
2. for  $i \leftarrow 1$  to  $|V[G]|-1$ 
3.   for each edge  $(u, v) \in E[G]$ 
4.     Relax( $u, v, w$ );
5. for each edge  $(u, v) \in E[G]$ 
6.   if  $d[v] > d[u] + w(u, v)$ 
7.     return FALSE;
8. return TRUE
    
```

- Solves the case where **edge weights can be negative**.
- Returns FALSE if there exists a **negative-weight cycle** reachable from the source; TRUE otherwise.
- Time complexity:  $O(VE)$ .

Unit 8

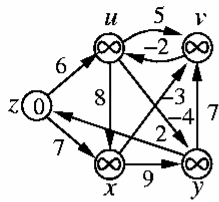
NTUEE / intro. EDA

32

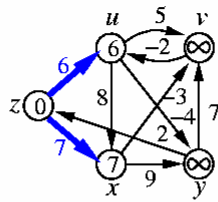


## Example for Bellman-Ford for **Shortest Paths**

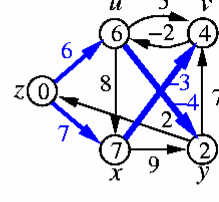
relax edges in lexicographic order:  $(u, v)$ ,  $(u, x)$ ,  $(u, y)$ , ...,  $(z, u)$ ,  $(z, x)$



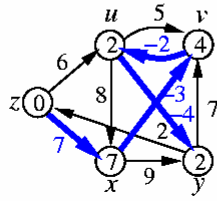
(a)



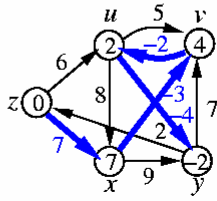
(b)



(c)



(d)



(e)