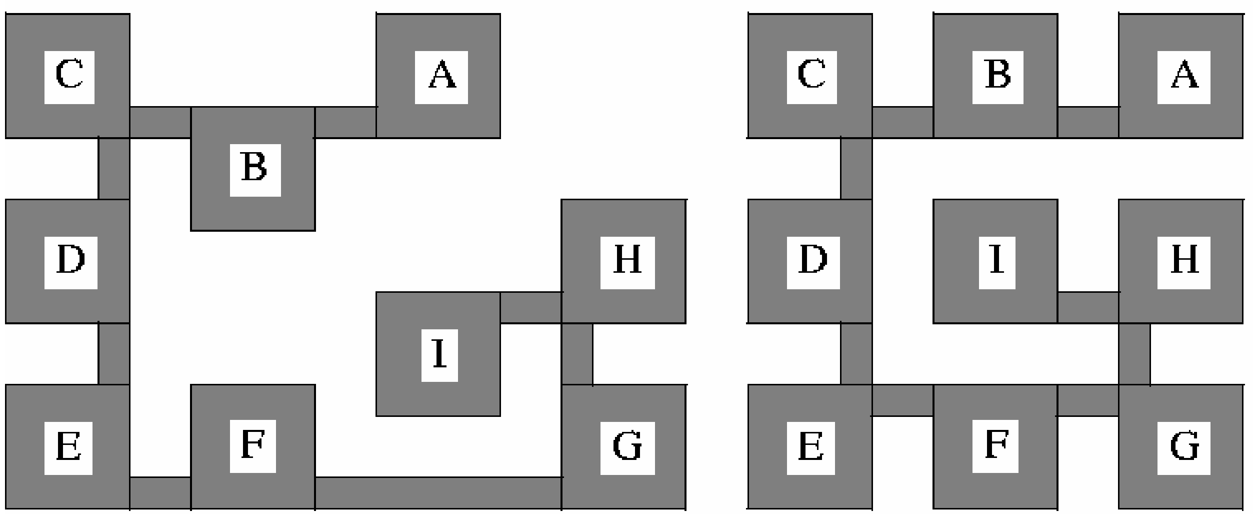
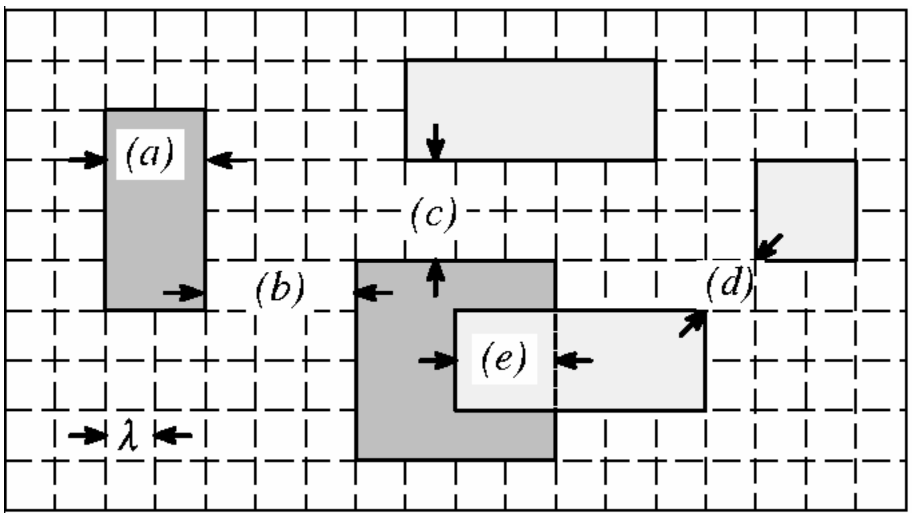
Unit 5F: Layout Compaction

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



Design Rules

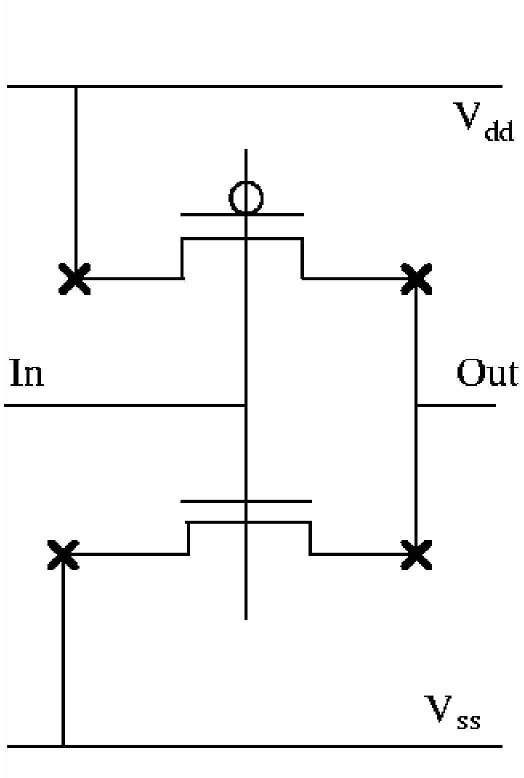
- **Design rules:** restrictions on the mask patterns to increase the probability of successful fabrication.



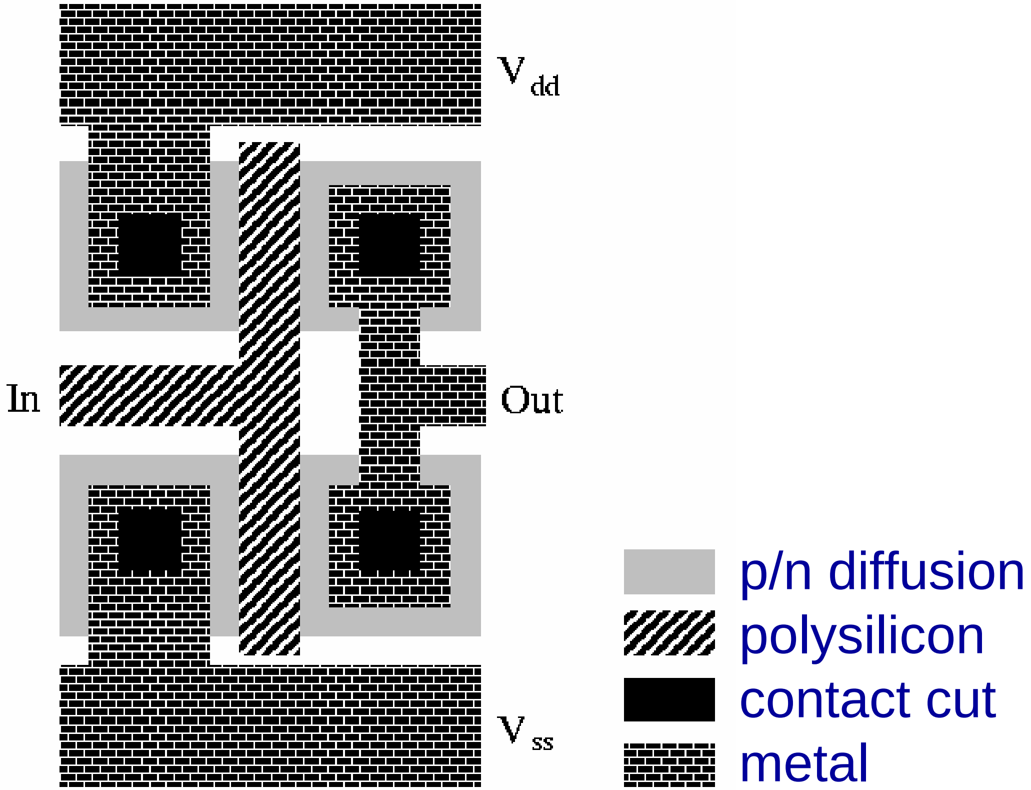
- Patterns and design rules are often expressed in λ rules.
- Most common design rules:
  - minimum-width rules (valid for

    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).

CMOS Inverter Layout Example



Symbolic layout



Geometric layout

Symbolic Layout

- Geometric (mask) layout: coordinates of the layout patterns (rectangles) are absolute (or in multiples of λ).
- 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 other 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.

Compaction and Its Applications

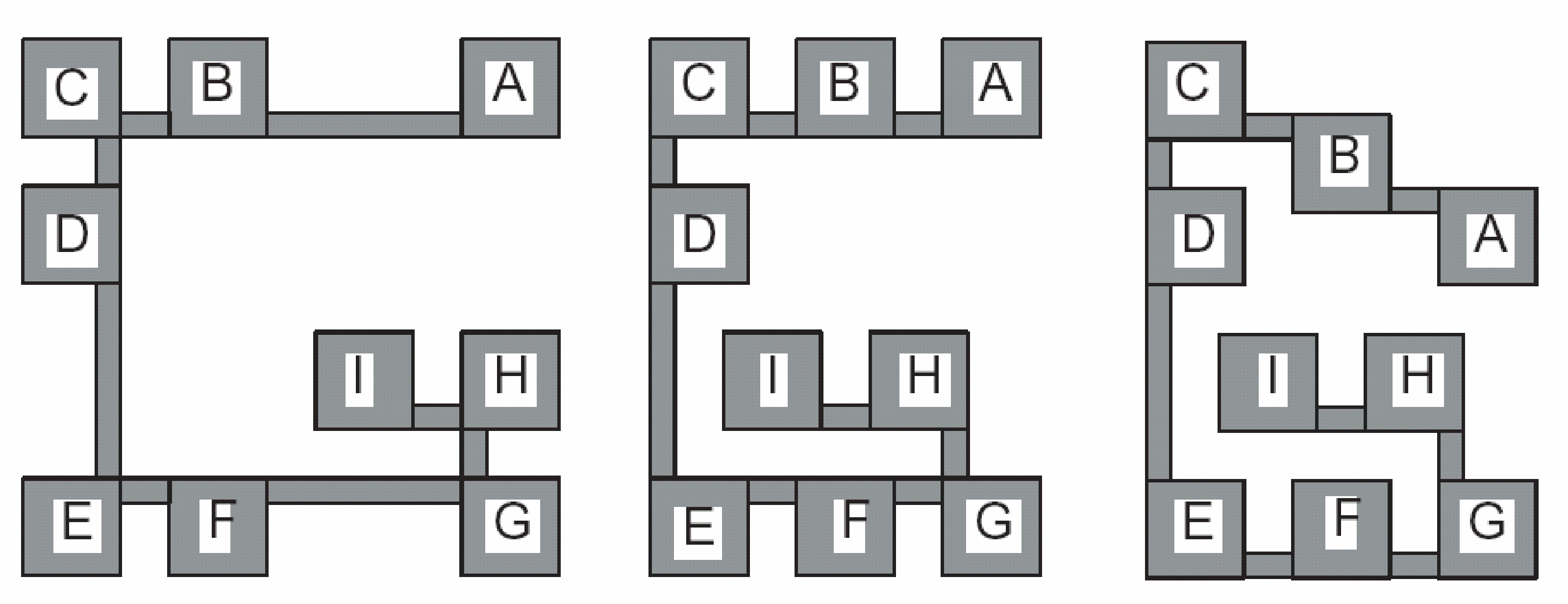
- A compaction program or compactor generates layout at the mask level. It attempts to make the layout as dense as possible.
- Applications of compaction:
  - Area minimization: remove redundant space in layout at the mask level.
  - Layout compilation: generate mask-level layout from symbolic layout.
  - 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.

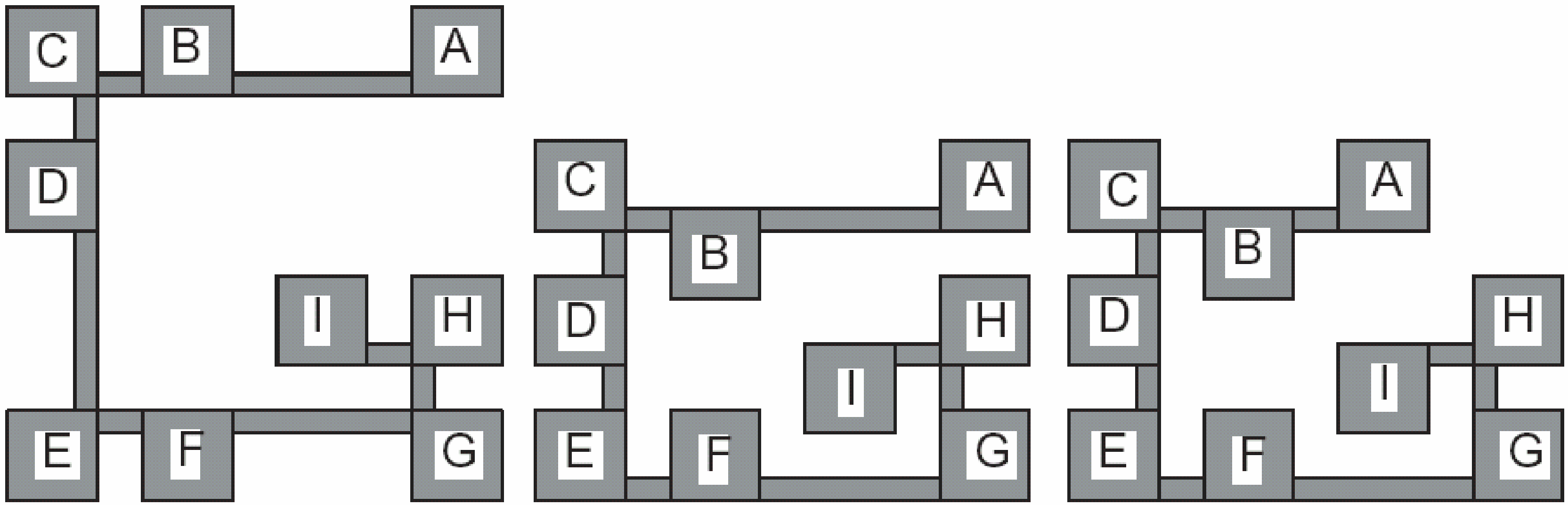
1D Compaction: *X* Followed By *Y*

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



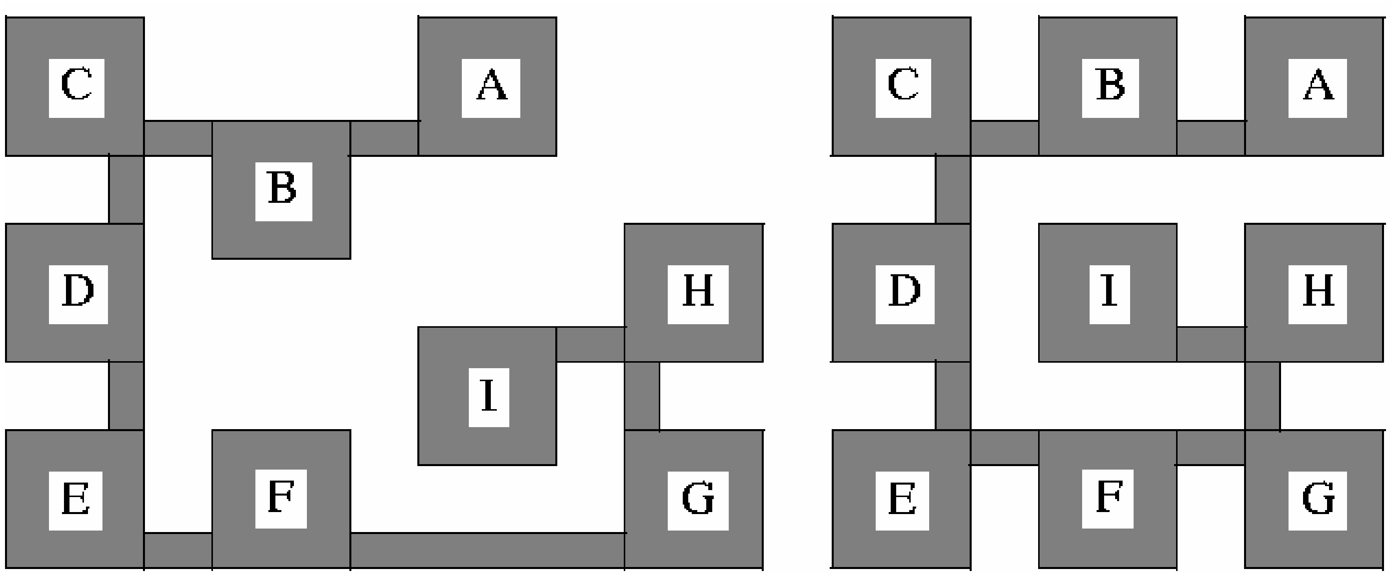
1D Compaction: *Y* Followed By *X*

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



2D Compaction

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

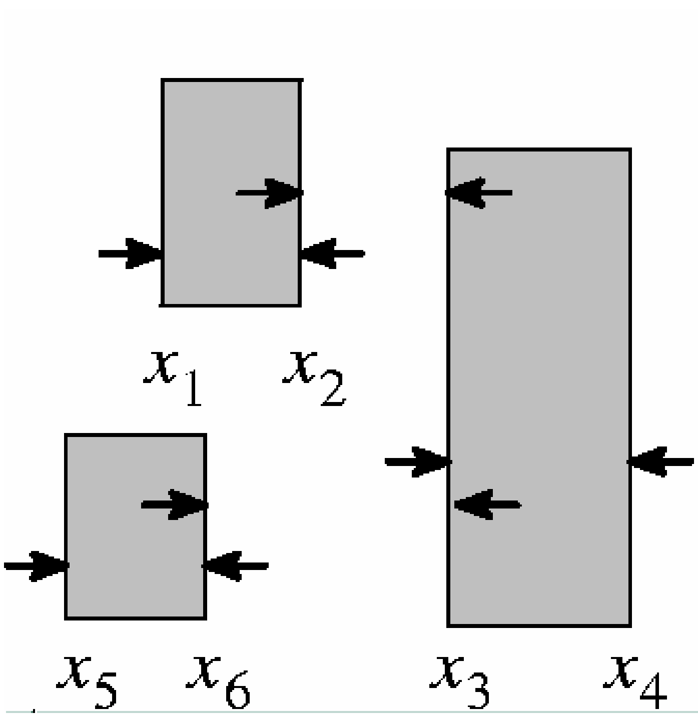


- Since 2D compaction is NP-complete, most compactors are based on repeated 1D compaction.

Inequalities for Distance Constraints

- Minimum-distance design rules can be expressed as inequalities.

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

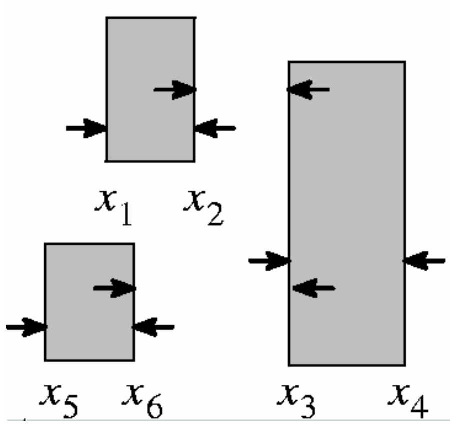


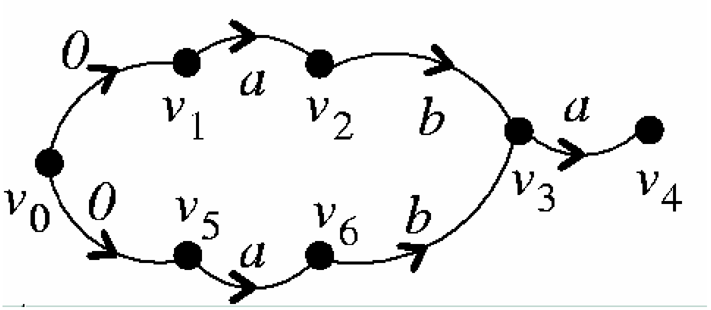
- For example, if the minimum width is *a* and the minimum separation is *b*, then

$$x_2 - x_1 \geq a$$
$$x_3 - x_2 \geq b$$
$$x_3 - x_6 \geq b$$

The 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.

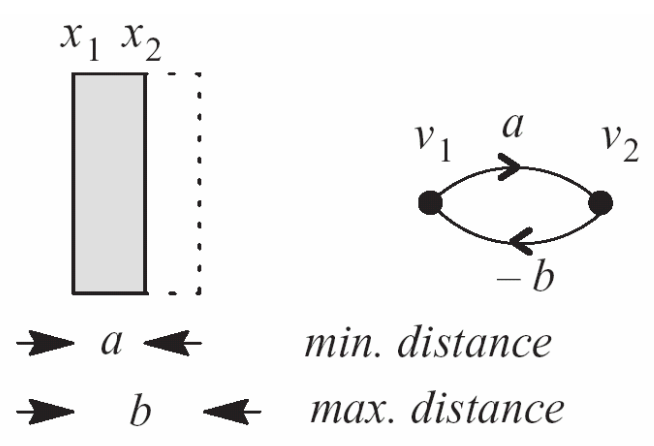


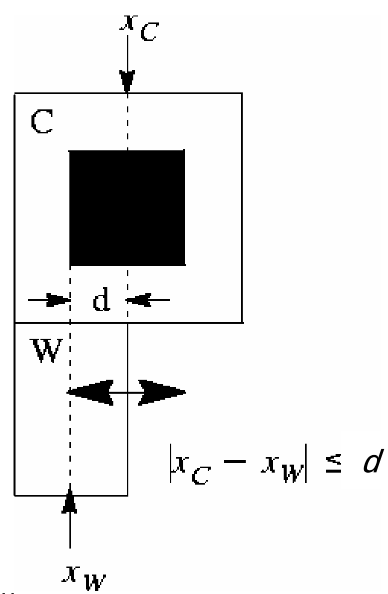


constraint graph

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.

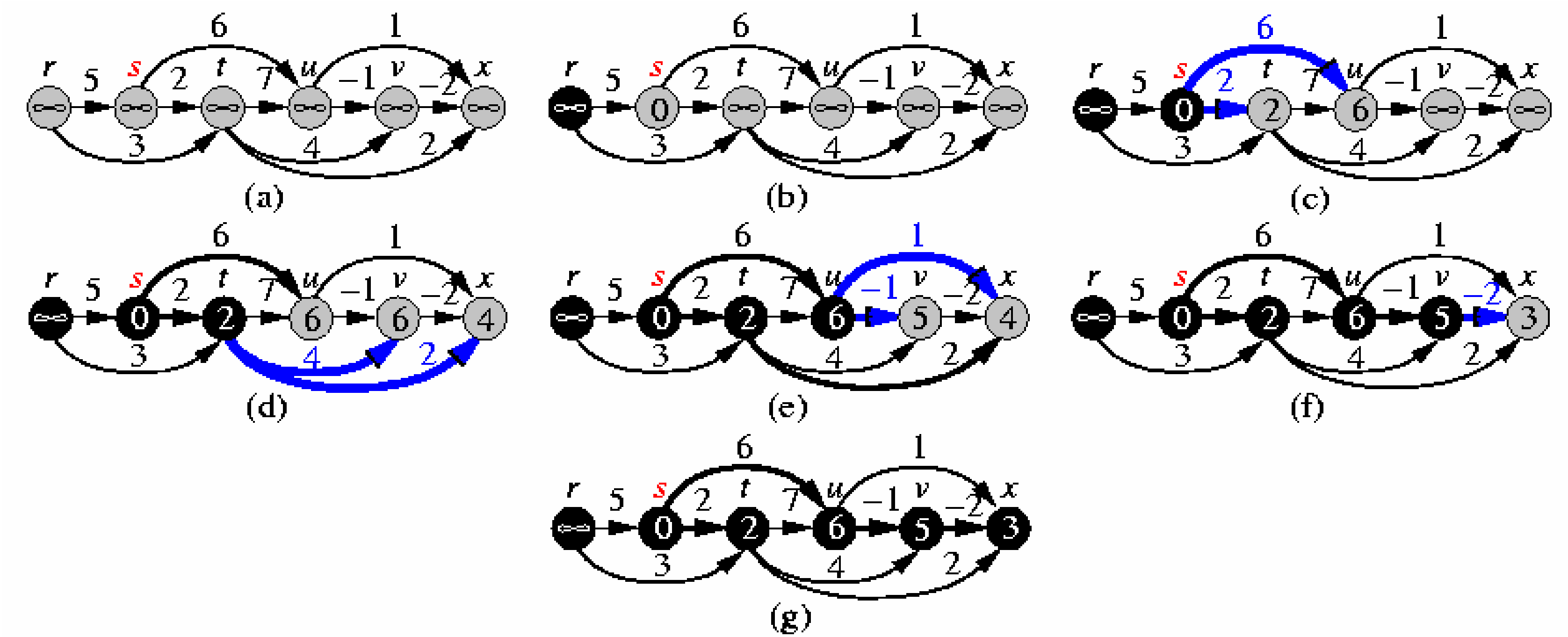




Shortest Path for Directed Acyclic Graphs (DAGs)

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).

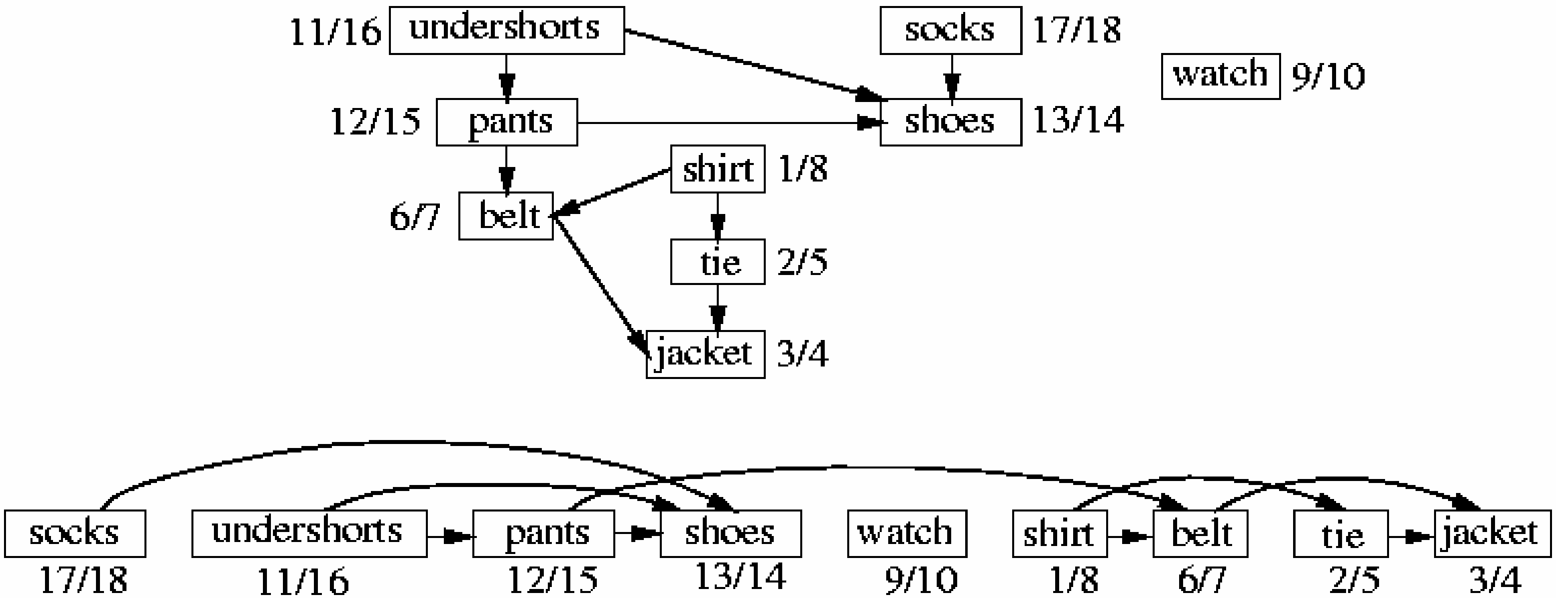


Topological Sort

- 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)
> 1. call DFS(G) to compute finishing times $f[v]$ for each vertex v
> 2. as each vertex is finished, insert it onto the front of a linked list
> 3. **return** the linked list of vertices

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



Vertices are arranged from left to right in order of decreasing finishing times.

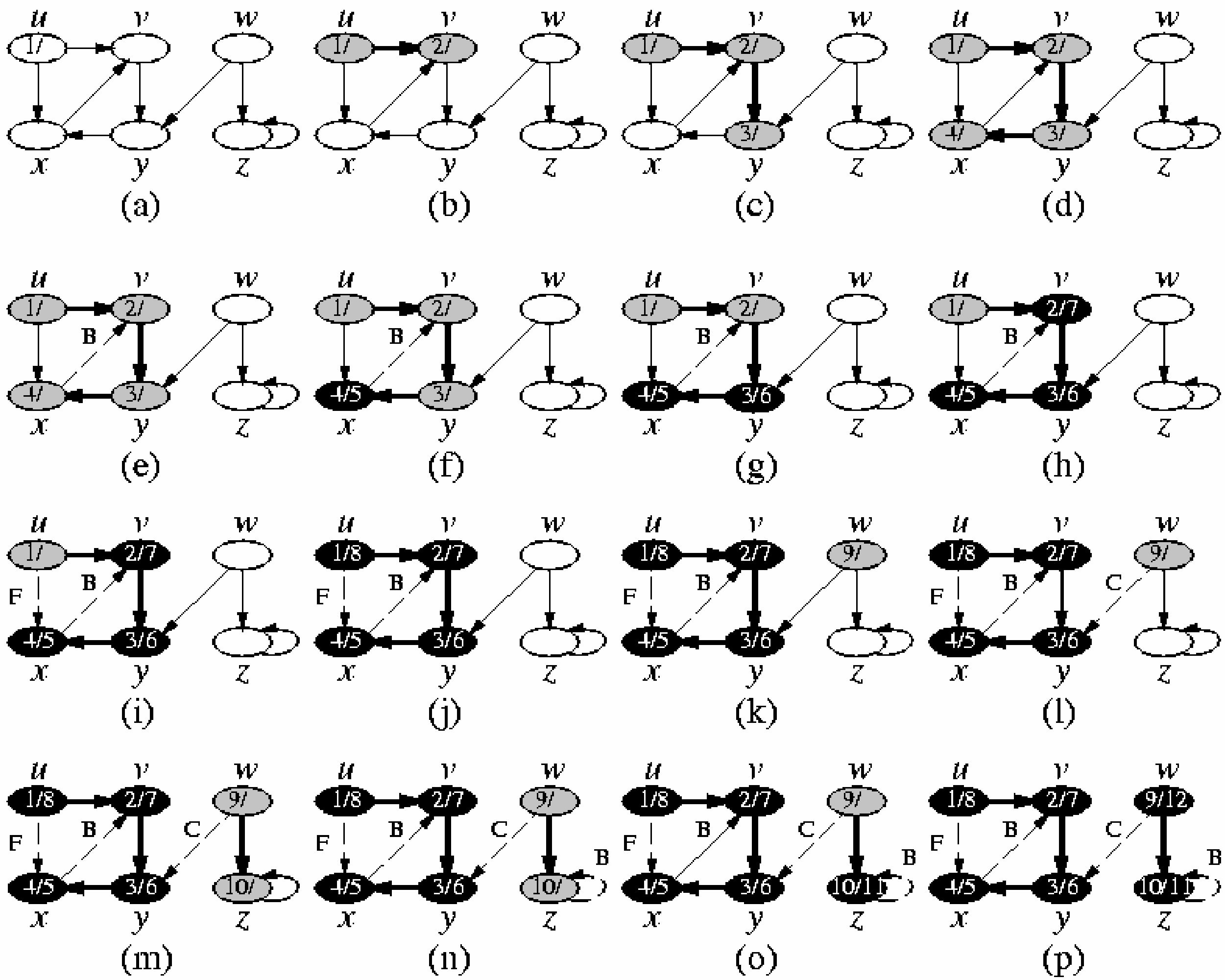
Depth-First Search (DFS)

```
DFS(G)
1. for each vertex u ∈ V[G]
2.    color[u] ← WHITE;
3.    π [u] ←NIL;
4. time ← 0;
5. for  each vertex u ∈ V[G]
6.     if color[u] = WHITE
7.         DFS-Visit(u).

DFS-Visit(u)
1. color[u] ← GRAY;
  /* white vertex u has just been
     discovered. */
2. d[u] ← time ← time + 1;
3. for  each vertex v ∈ Adj[u]
      /* Explore edge (u,v). */
4.    if color[v] = WHITE
5.        π [v] ← u;
6.         DFS-Visit(v);
7. color[u] ← BLACK;
   /* Blacken u; it is finished. */
8. f[u] ← time ← 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); π[*u*]: predecessor.
- Time complexity: $O(V+E)$ (adjacency list).

DFS Example



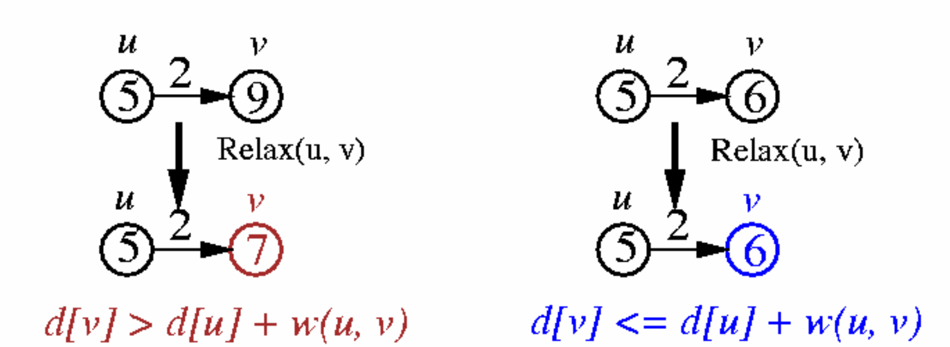
- *color*[*u*]: white → gray → black.
- Depth-first **forest**: $G_\pi = (V, E_\pi)$, $E_\pi = \{(\pi[v], v) \in E \mid v \in V, \pi[v] \neq \text{NIL}\}$.

Relaxation

Initialize-Single-Source(G, s)
1. **for** each vertex $v \in V[G]$
2. $\quad d[v] \leftarrow \infty$;
 $\quad$ /* upper bound on the weight of a shortest path from s to v */
3. $\quad \pi[v] \leftarrow$ NIL; /* predecessor of v */
4. $d[s] \leftarrow 0$;

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

- $d[v] \leq d[u] + w(u, v)$ after calling 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.
- Let $s \leadsto u \rightarrow v$ be a shortest path. If $d[u] = \delta(s, u)$ prior to the call Relax(u, v, w), then $d[v] = \delta(s, v)$ after the call.



Longest-Path Algorithm for DAGs

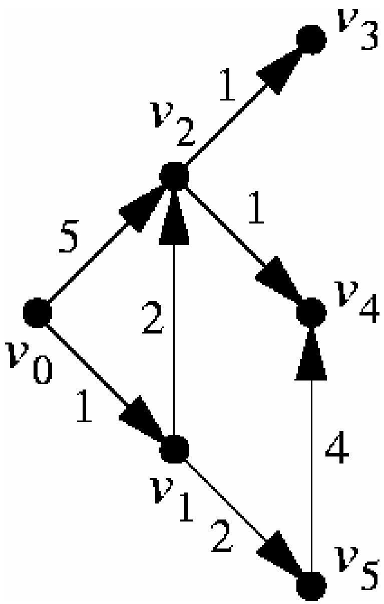
```
longest-path(G)
{
  for (i ← 1; i ≤ n; i ← i + 1)
    p_i ← "in-degree of v_i";
  Q ← {v_0};
  while (Q ≠ ∅) {
     v_i ← "any element from Q";
     Q ← Q \{v_i};
     for each v_j "such that" (v_i, v_j) ∈ E {
        x_j ← max(x_j, x_i + d_ij);
        p_j ← p_j − 1;
        if (p_j ≤ 0)
           Q ← Q ∪ {v_j};
     }
  }
}
```

```
main ()
{
  for (i ← 0; i ≤ n; i ← i + 1)
     x_i ← 0;
  longest-path(G);
}
```

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

Chang, Huang, Li, Lin, Liu

DAG Longest-Path Example

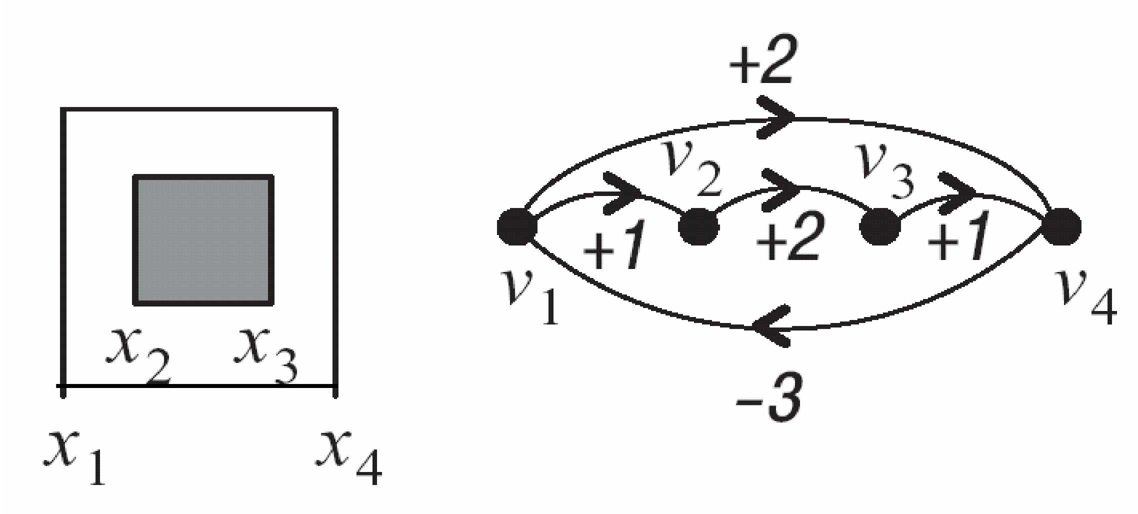


- Runs in a breadth-first search manner.
- p_i: in-degree of v_i.
- x_i: longest-path length from v_0 to v_i.
- Time complexity: $O(V+E)$.

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.



The Liao-Wong Algorithm

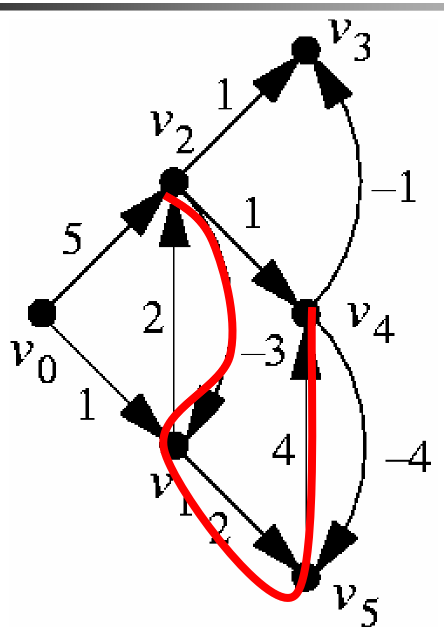
- Split the edge set E of the constraint graph into two subsets:
  - Forward edges E_f: related to minimum-distance constraints.
  - Backward edges E_b: related to maximum-distance constraints.
- The graph $G(V, E_f)$ is acyclic; the longest distance for each vertex can be computed with the procedure "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: $O(E_b \times E_f)$.

Pseudo Code: The Liao-Wong Algorithm

```
count ← 0;
for (i ← 1; i ≤ n; i ← i + 1)
    x_i ← −∞;
x_0 ← 0;

do { flag ← 0;
        longest-path(G_f);
        for each (v_i, v_j) ∈ E_b
            if (x_j < x_i + d_ij) {
                x_j ← x_i + d_ij;
                flag ← 1;
            }
        count ← count +1;
        if (count > |E_b| && flag)
            error("positive cycle")
}
while (flag);
```

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: $O(E_b \times E_f)$.

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

x1 < x2 - 3

x3 < x4 - 1

x5 = x4 - 4

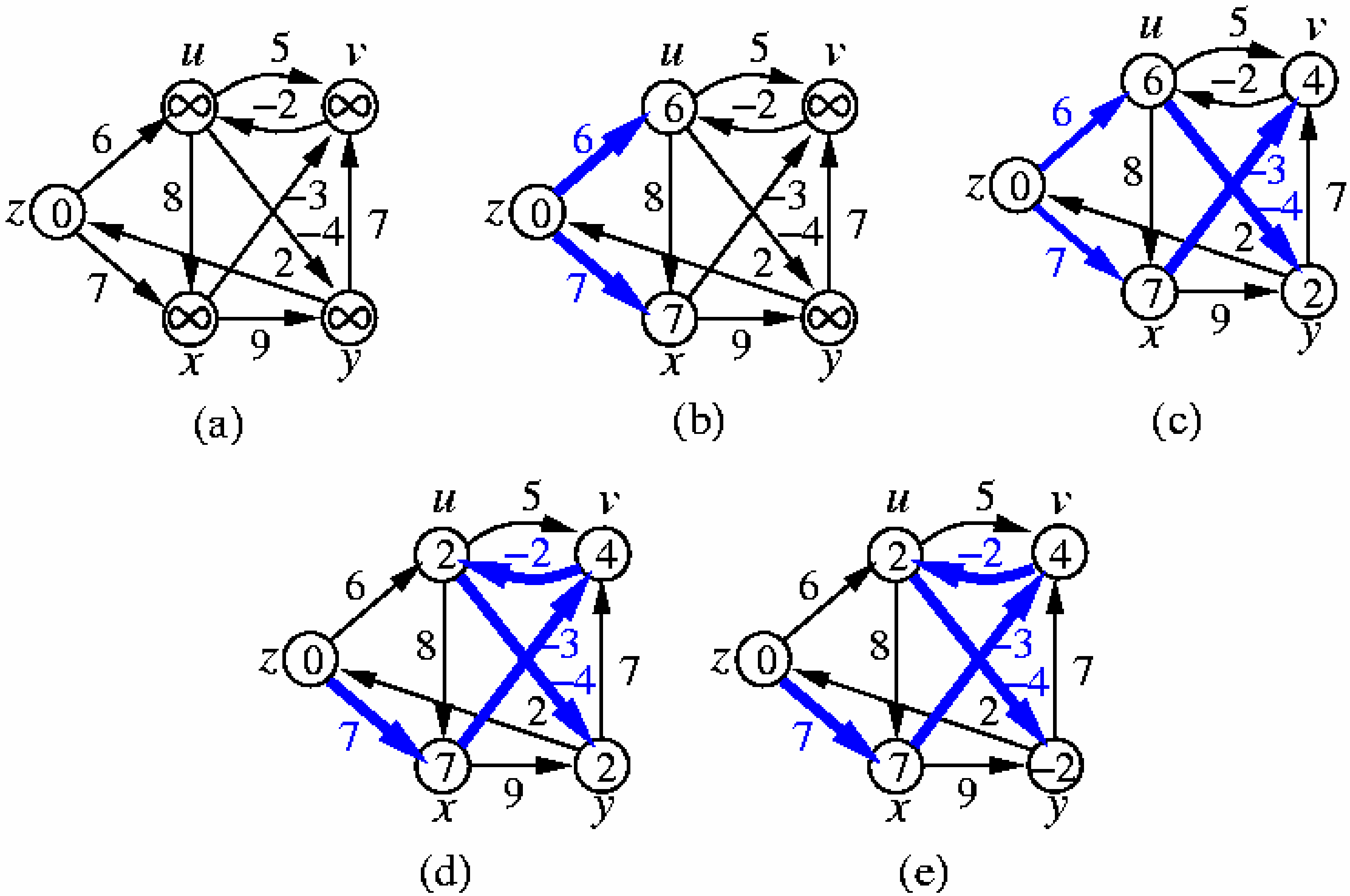
The Bellman-Ford Algorithm for Shortest Paths

```
Bellman-Ford(G,w, s)
1. Initialize-Single-Source(G, s);
2. for  i ← 1 to |V[G]|-1
3.   for  each edge (u, v) ∈ E[G]
4.         Relax(u, v, w);
5. for each edge (u, v) ∈ 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 cycle reachable from the source; TRUE otherwise.
- Time complexity: $O(VE)$.

Example for Bellman-Ford for Shortest Paths

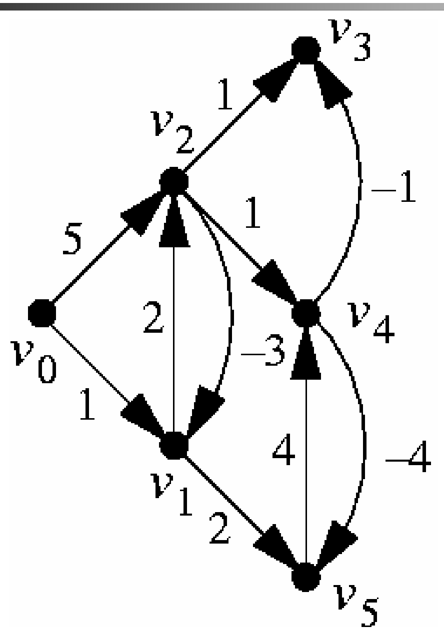
relax edges in lexicographic order: (u, v), (u, x), (u, y), ..., (z, u), (z, x)



The Bellman-Ford Algorithm for Longest Paths

$$
\begin{array}{l}
\textbf{for } (i \leftarrow 1;\ i \le n;\ i \leftarrow i+1) \\
\quad x_i \leftarrow -\infty; \\
x_0 \leftarrow 0; \\
\text{count} \leftarrow 0; \\
S_1 \leftarrow \{v_0\}; \\
S_2 \leftarrow \emptyset; \\
\textbf{while } (\text{count} \le n \ \&\&\ S_1 \ne \emptyset)\ \{ \\
\quad \textbf{for each } v_i \in S_1 \\
\quad\quad \textbf{for each } v_j \text{ ``such that'' } (v_i, v_j) \in E \\
\quad\quad\quad \textbf{if } (x_j < x_i + d_{ij})\ \{ \\
\quad\quad\quad\quad x_j \leftarrow x_i + d_{ij}; \\
\quad\quad\quad\quad S_2 \leftarrow S_2 \cup \{v_j\} \\
\quad\quad\quad \} \\
\quad S_1 \leftarrow S_2; \\
\quad S_2 \leftarrow \emptyset; \\
\quad \text{count} \leftarrow \text{count} + 1; \\
\} \\
\textbf{if } (\text{count} > n) \\
\quad \text{error(``positive cycle'');}
\end{array}
$$

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: $O(VE)$.

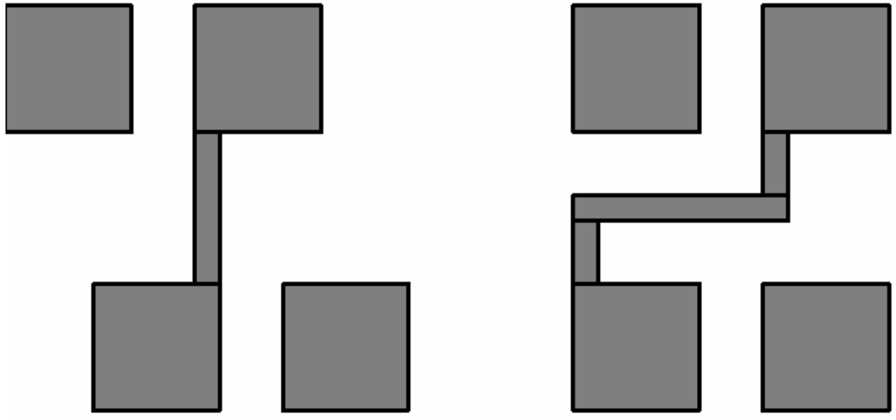
S_1	x_1	x_2	x_3	x_4	x_5
“not initialized”	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$\{v_0\}$	1	5	$-\infty$	$-\infty$	$-\infty$
$\{v_1, v_2\}$	2	5	6	6	3
$\{v_1, v_3, v_4, v_5\}$	2	5	6	7	4
$\{v_4, v_5\}$	2	5	6	8	4
$\{v_4\}$	2	5	7	8	4
$\{v_3\}$	2	5	7	8	4

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.

Remarks on Constraint-Graph Compaction

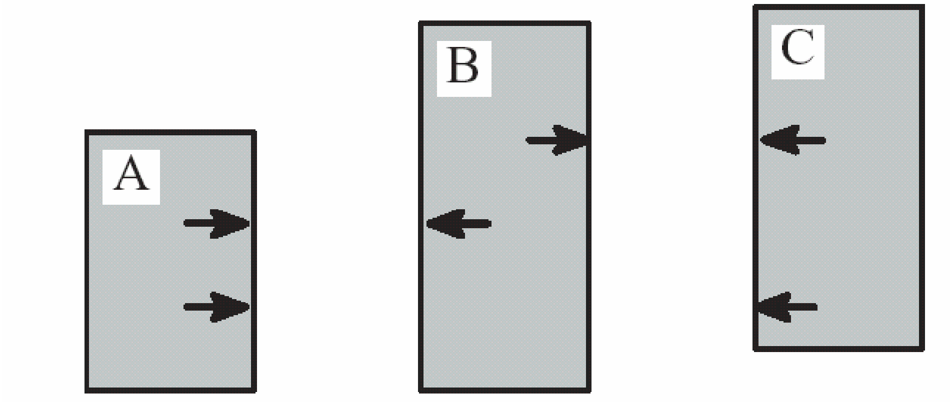
- Noncritical layout elements: Every element outside the critical paths has freedom on its best position => may use this freedom to optimize some cost function.
- Automatic jog insertion: The quality of the layout can further be improved by automatic jog insertion.



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

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 $O(n \log n)$.