Unit 4: 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 \( \lambda \) 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).
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 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 $2\lambda \times 2\lambda$, minimum separation is $1\lambda$.
- Initially, the layout is $11\lambda \times 11\lambda$.
- After compacting along the $x$ direction, then the $y$ direction, we have the layout size of $8\lambda \times 11\lambda$.

1D Compaction: Y Followed By X

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

![Diagram of 2D compaction](image)

- 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.

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 ∈ 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. $(u, v) ∈ E ⇒ u$ 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).
Depth-First Search (DFS)

DFS(G)
1. for each vertex \( u \in V[G] \)
2. \( \text{color}[u] \leftarrow \text{WHITE} \);
3. \( \pi[u] \leftarrow \text{NIL} \);
4. \( \text{time} \leftarrow 0 \);
5. for each vertex \( u \in V[G] \)
6. \( \text{if color}[u] = \text{WHITE} \)
7. \( \text{DFS-Visit}(u) \).

DFS-Visit(u)
1. \( \text{color}[u] \leftarrow \text{GRAY}; \)
2. \( \text{d}[u] \leftarrow \text{time} \leftarrow \text{time} + 1; \)
3. for each vertex \( v \in \text{Adj}[u] \)
4. \( \text{if color}[v] = \text{WHITE} \)
5. \( \pi[v] \leftarrow u; \)
6. \( \text{DFS-Visit}(v); \)
7. \( \text{color}[u] \leftarrow \text{BLACK}; \)
8. \( \text{f}[u] \leftarrow \text{time} \leftarrow \text{time} + 1. \)

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

DFS Example

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

Initialize-Single-Source(G, s)
1. for each vertex v ∈ V[G]
2. d[v] ← ∞;
   /* upper bound on the weight of a shortest path from s to v */
3. π[v] ← NIL; /* predecessor of v */
4. d[s] ← 0;

Relax(u, v, w)
1. if d[v] > d[u]+w(u, v)
2. d[v] ← d[u]+w(u, v);
3. π[v] ← u;

• d[v] ≤ d[u] + w(u, v) after calling Relax(u, v, w).
• d[v] ≥ δ(s, v) during the relaxation steps; once d[v] achieves its lower bound δ(s, v), it never changes.
• Let s → u → v be a shortest path. If d[u] = δ(s, u) prior to the call Relax(u, v, w), then d[v] = δ(s, v) after the call.

Longest-Path Algorithm for DAGs

longest-path(G)
{ }
for (i ← 1; i ≤ n; i ← i + 1)
   pi ← "in-degree of vi";
   Q ← {v0};
while (Q ≠ ∅) [ 
   vi ← "any element from Q";
   Q ← Q \{v0};
for each vj "such that" (v0, vj) ∈ E [
   xj ← max(xj, xi + dij);
   pj ← pj − 1;
   if (pj ≥ 0) 
   Q ← Q ∪ {vj};
] } 

main ()
{ }
for (i ← 0; i ≤ n; i ← i + 1)
x0 ← 0;
longest-path(G);

• pi: in-degree of vi.
• xj: longest-path length from v0 to vi.
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: \( O(V+E) \).

<table>
<thead>
<tr>
<th>( C )</th>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>( p_3 )</th>
<th>( p_5 )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
<th>( x_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;not initialized&quot;</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{v_0}</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{v_1}</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{v_2, v_3}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>{v_2, v_5}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>{v_5}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>{v_4}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

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 $G(V, E)$ 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

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

do { flag ← 0;
    longest-path($G_f$);
    for each $(v_i, v_j) \in E_B$
        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: $O(E_b \times E_f)$.

<table>
<thead>
<tr>
<th>Step</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>Forward 1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Backward 1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Forward 2</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Backward 2</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Forward 3</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Backward 3</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

\[
x_1 < x_2 - 3
\]
\[
x_3 < x_4 - 1
\]
\[
x_5 = x_4 - 4
\]

### The Bellman-Ford Algorithm for Shortest Paths

- 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: \((u, v), (u, x), (u, y), ..., (z, u), (z, x)\)

(a) \(z\)
(b) \(z\)
(c) \(z\)

(d) \(z\)
(e) \(z\)

The Bellman-Ford Algorithm for Longest Paths

```plaintext
for (i ← 1; i ≤ n; i ← i + 1)
    \(x_i ← -∞;\)
    \(x_{v_0} ← 0;\)
    count ← 0;
    \(S_1 ← \{v_0\};\)
    \(S_2 ← \emptyset;\)
    while (count ≤ n && \(S_1 \neq \emptyset\)) {
        for each \(v_j \in S_1\)
            for each \(v_j\) "such that" \((v_j, v_j) \in E\)
                if \(x_j < x_i + d_{ij}\) {
                    \(x_j ← x_i + d_{ij};\)
                    \(S_2 ← S_2 ∪ \{v_j\}\)
                }
        \(S_1 ← S_2;\)
        \(S_2 ← \emptyset;\)
        count ← count + 1;
    }
    if (count > n)
        error("positive cycle");
```
Example of Bellman-Ford for Longest Paths

- Repeated “wave front propagation.”
- $S_i$: 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: $O(VE)$.

<table>
<thead>
<tr>
<th>$S_i$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>“not initialized”</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>${v_0}$</td>
<td>1</td>
<td>5</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>${v_1, v_2}$</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>${v_1, v_3, v_4, v_5}$</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>${v_4, v_5}$</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>${v_4}$</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>${v_5}$</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

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.

![Diagram of compaction comparison]

- **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.

![Diagram of constraint generation]

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