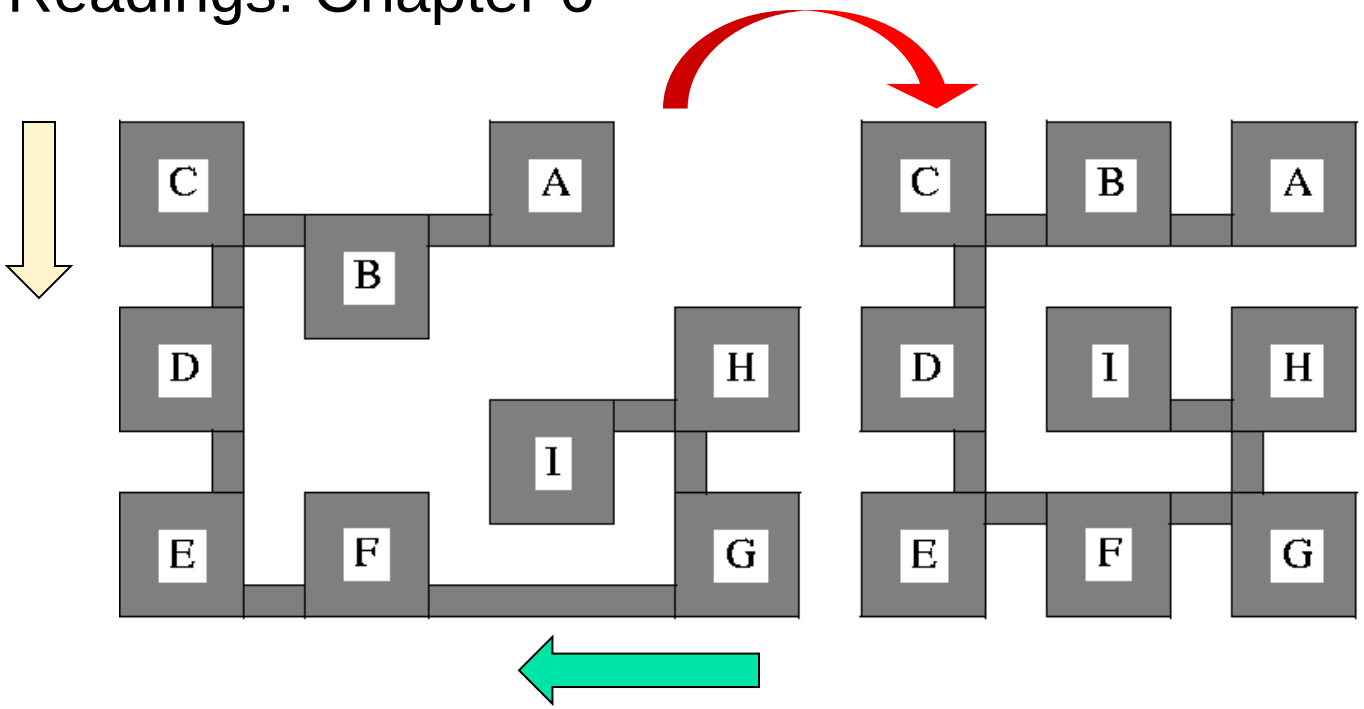
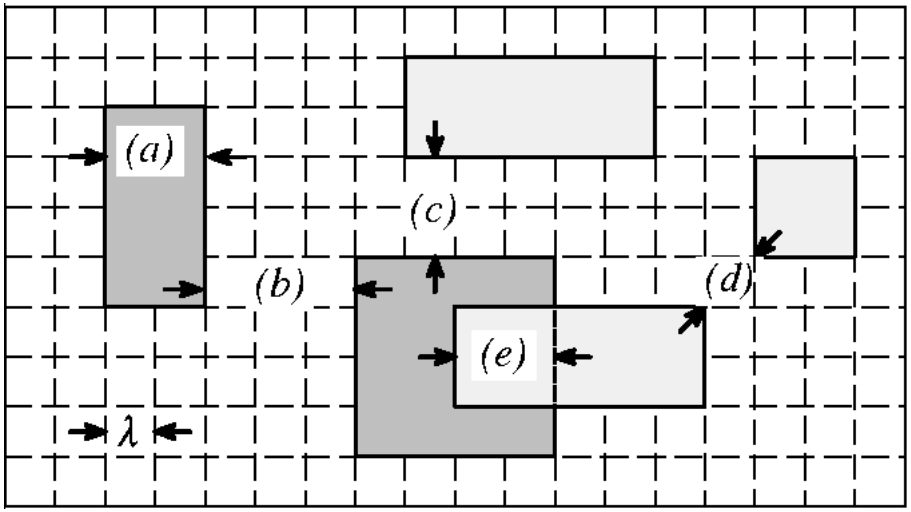
Unit 5F: Layout Compaction

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



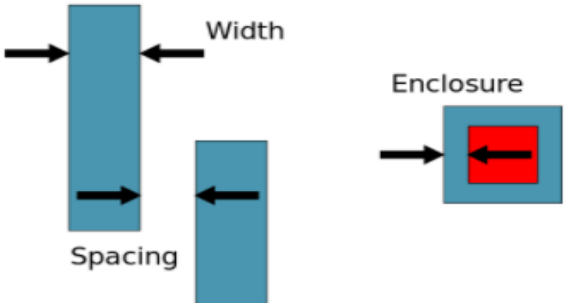
Design Rules (Prerequisite)

- **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, d).
  - minimum-**overlap** rules (mask patterns in different layers): (e).

The three basic DRC checks



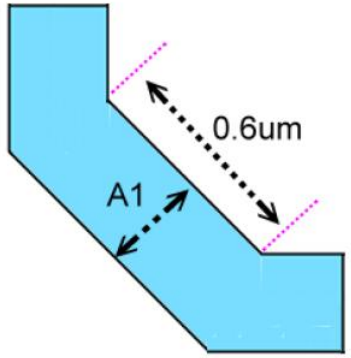
Different Width For 45-Degree Wire (In An Old Technology)

奇景盃 IC Layout 競賽 Design Rule

Himax

Metal-1 Rule

Rule No.	Description Layout	Rule
Layer : Metal 1	Metal 1	
M1.W.1	Minimum width of a M1 region	A >= 0.15 um
M1.W.2	Width of 45-degree bent M1 (length >=0.6 um)	A1 >= 0.18 um



Himax_附件三.PDF

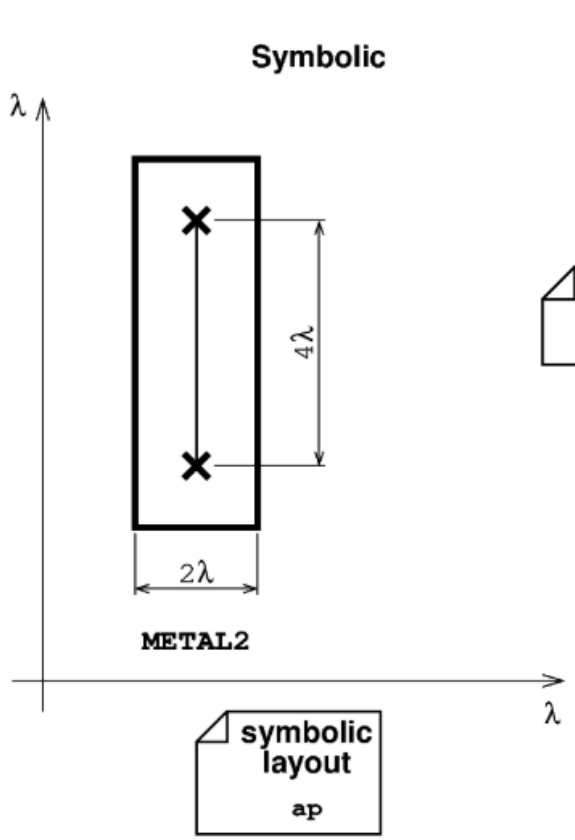
DRC rules used in Himax IC layout competition (mature node – 0.13um)

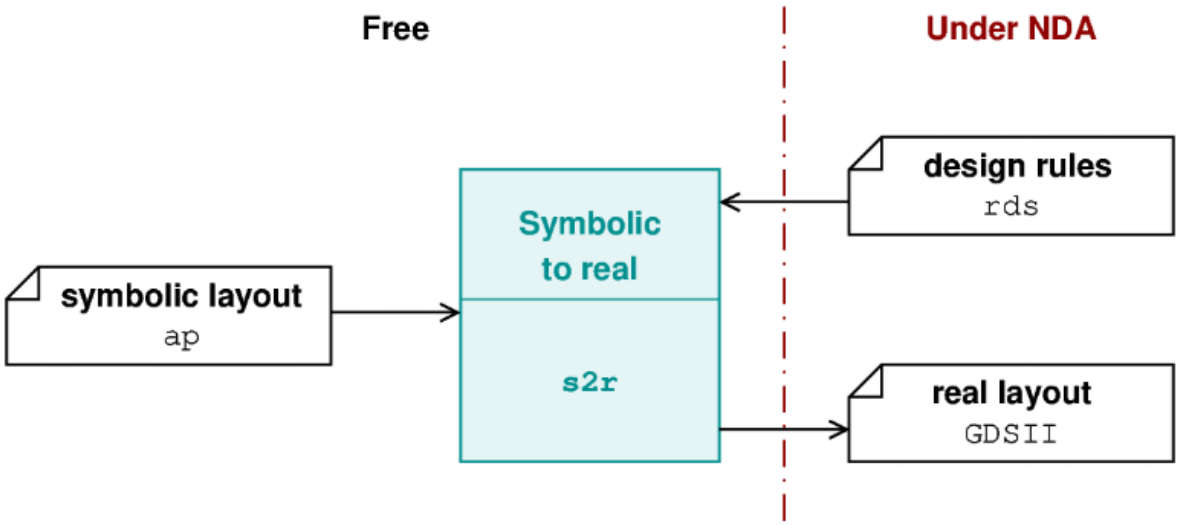
Worthwhile to show this PDF – lots of basic info

Not easy to deal with design rules.
Hence, compaction was brought in.

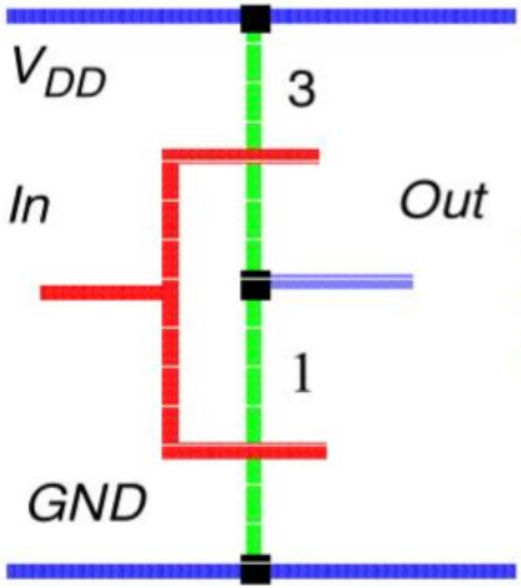
Layout: Geometric & Symbolic

- Geometric (mask) layout: coordinates of the layout patterns (rectangles) are absolute (or in multiples of λ).
- Symbolic (topological) layout: only relations among 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.





Symbolic Layout

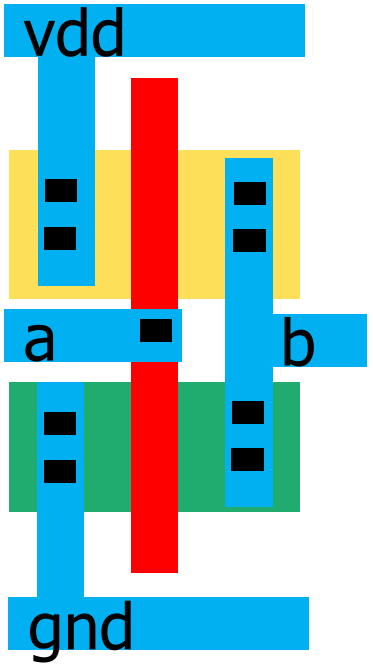


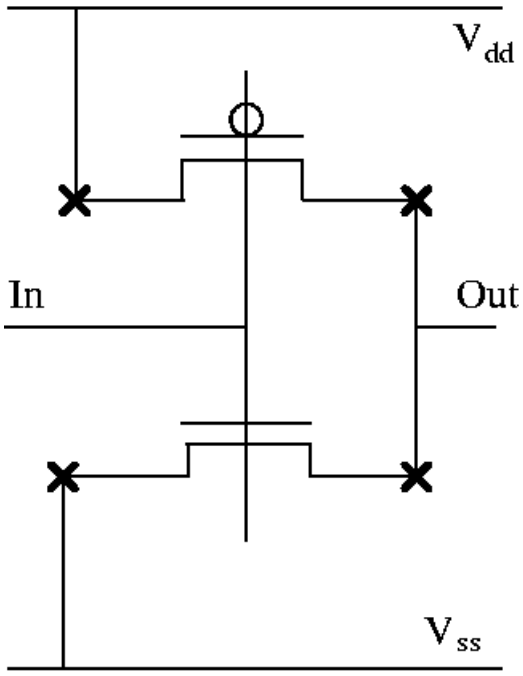
Stick diagram of inverter

- Dimensionless layout entities
- Only topology is important
- Final layout generated by "compaction" program

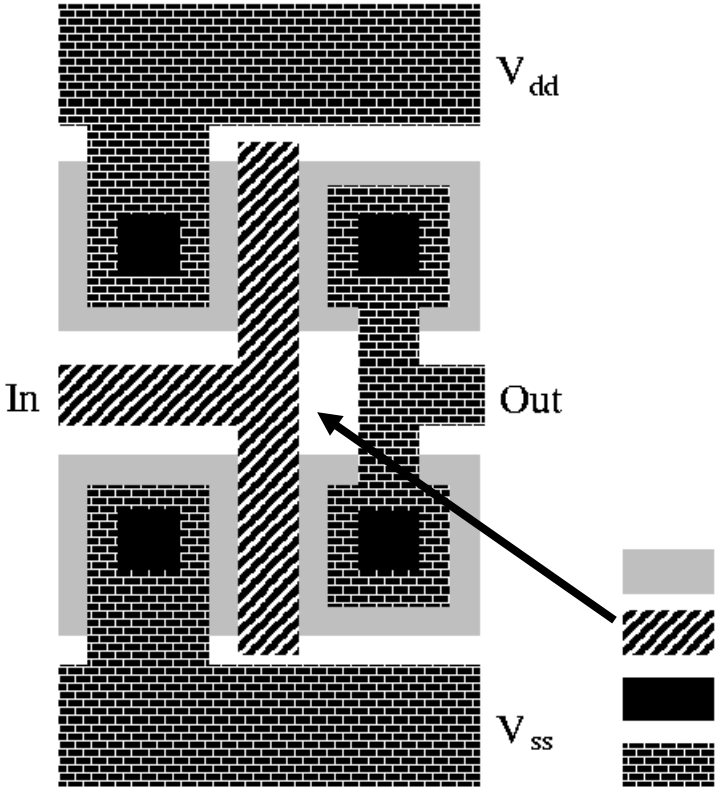
CMOS Inverter Layout Example

When poly has no bend





Symbolic layout



Geometric layout

p/n diffusion
polysilicon
contact cut
metal

Allow bending

Compaction and Its Applications

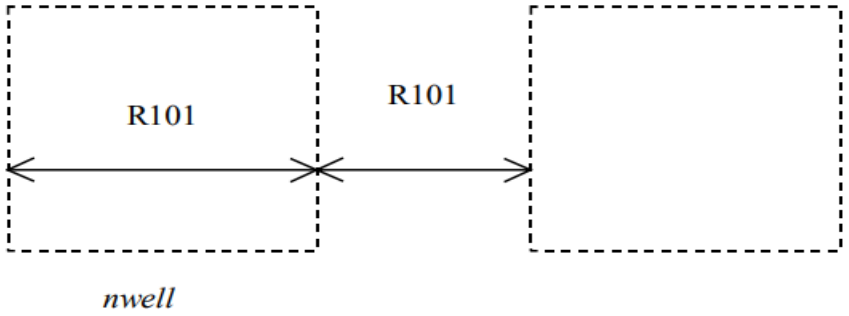
- A compaction program or compactor generates layout at the mask level. It attempts to make the layout as dense as possible.
- Applications of compaction:
  - Area minimization: remove redundant space in layout at the mask level.
  - Layout compilation: generate mask-level layout from symbolic layout.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

CMOS Lambda Design Rules
(This approach lasted for several years)

https://pages.jh.edu/aandreo1/216/Archives/2010/Handouts/appendixa.pdf (6 metal, 0.12um)

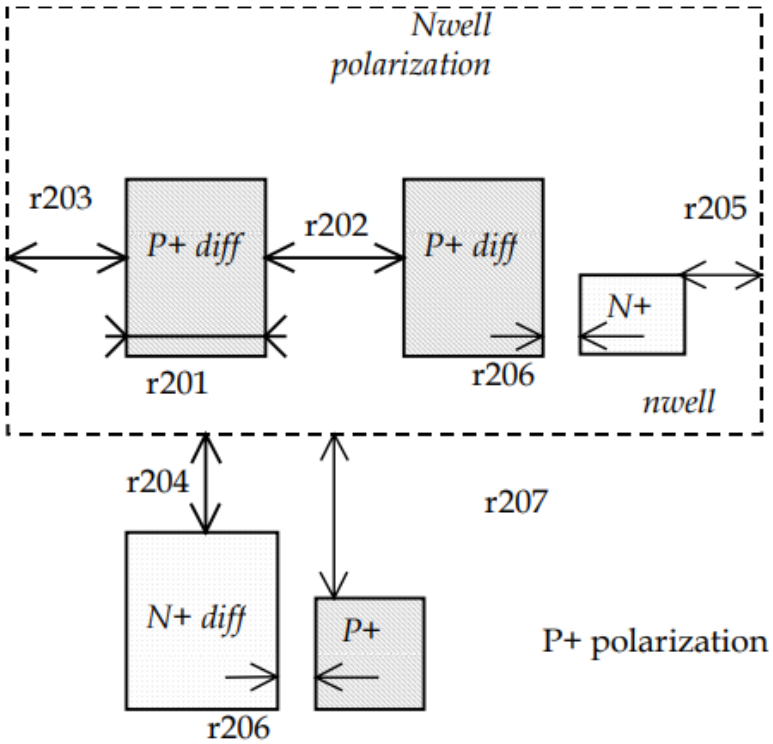
N-Well

r101	Minimum well size	12 λ
r102	Between wells	12 λ
r110	Minimum well area	144 λ^2



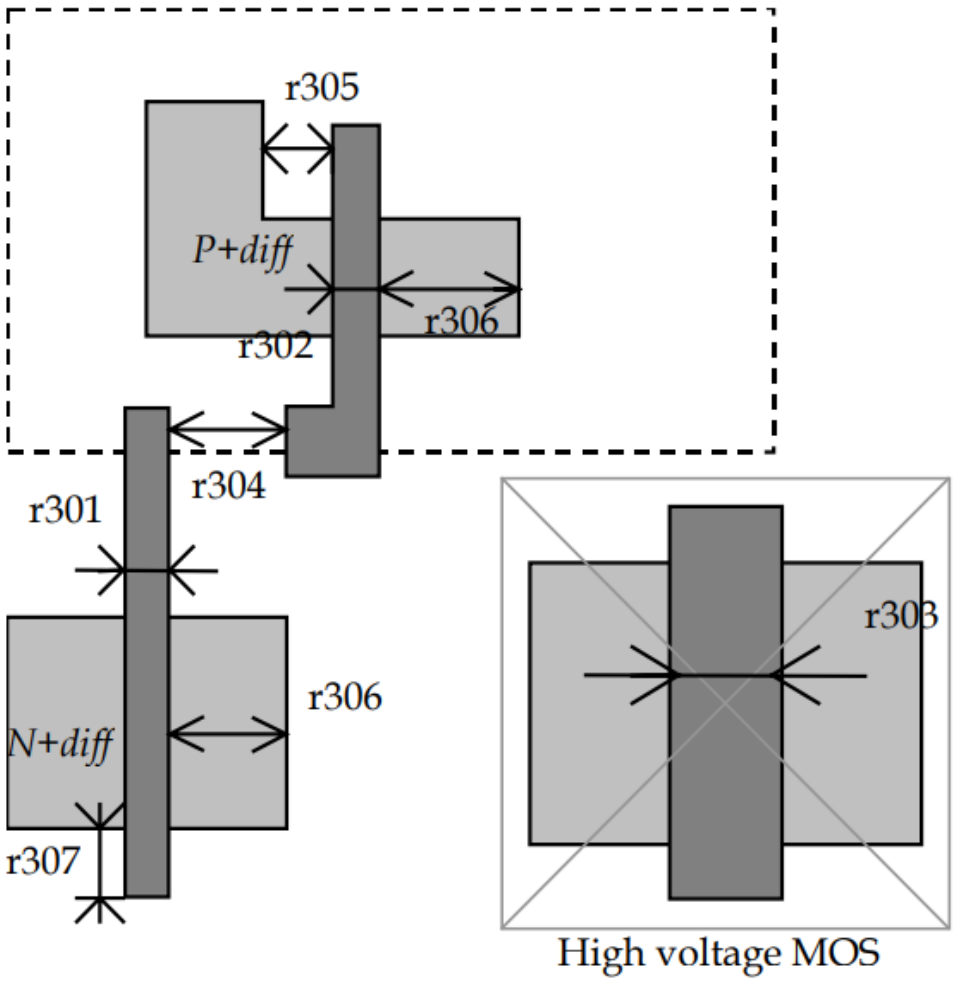
Diffusion

r201	Minimum N+ and P+ diffusion width	4 λ
r202	Between two P+ and N+ diffusions	4 λ
r203	Extra nwell after P+ diffusion :	6 λ
r204:	Between N+ diffusion and nwell	6 λ
r205	Border of well after N+ polarization	2 λ
r206	Between N+ and P+ polarization	0 λ
r207	Border of Nwell for P+ polarization	6 λ
r210	Minimum diffusion area	24 λ^2



Polysilicon

r301	Polysilicon width	2λ
R302	Polysilicon gate on diffusion	2 λ
R303	Polysilicon gate on diffusion for high voltage MOS	4 λ
R304	Between two polysilicon boxes	3 λ
R305	Polysilicon vs. other diffusion	2 λ
R306	Diffusion after polysilicon	4 λ
R307	Extra gate after polysilicon	3 λ
r310	Minimum surface	8 λ^2



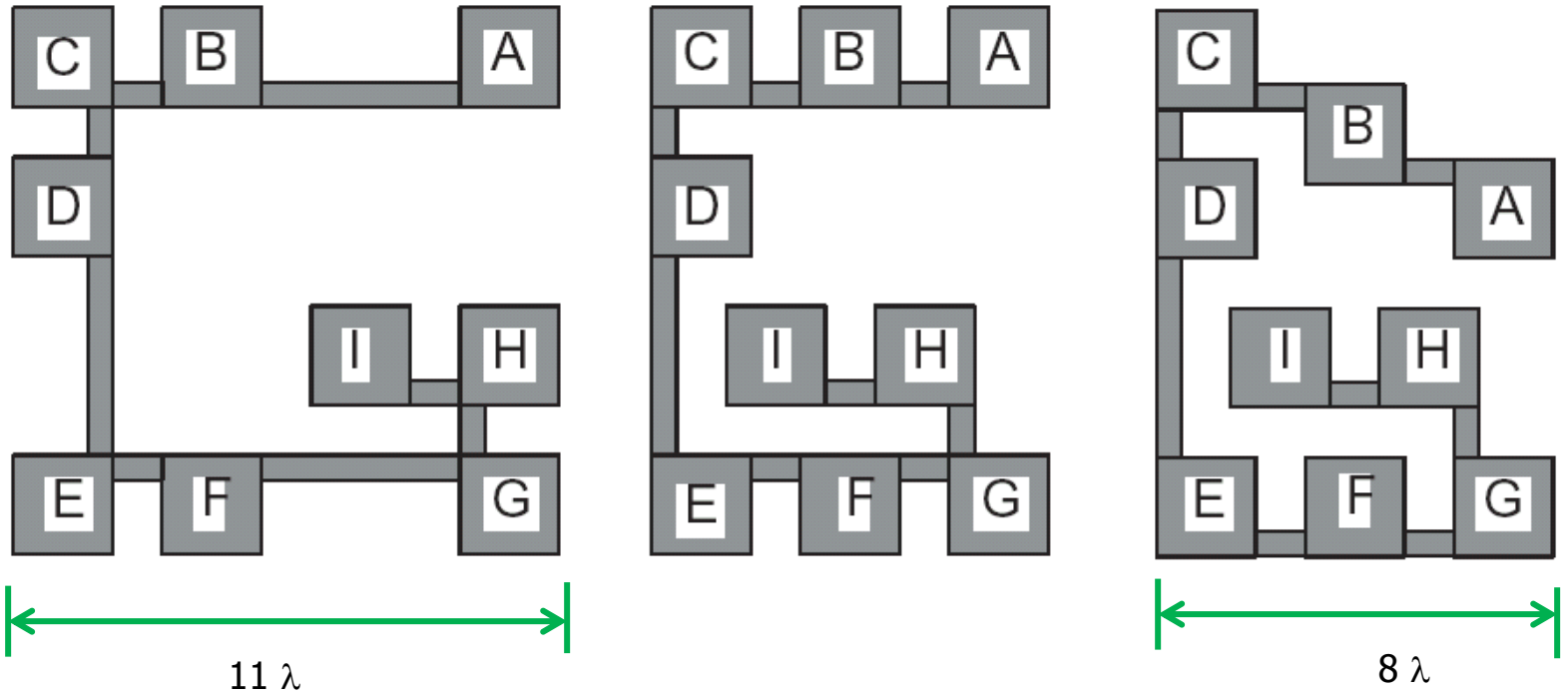
High voltage MOS

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

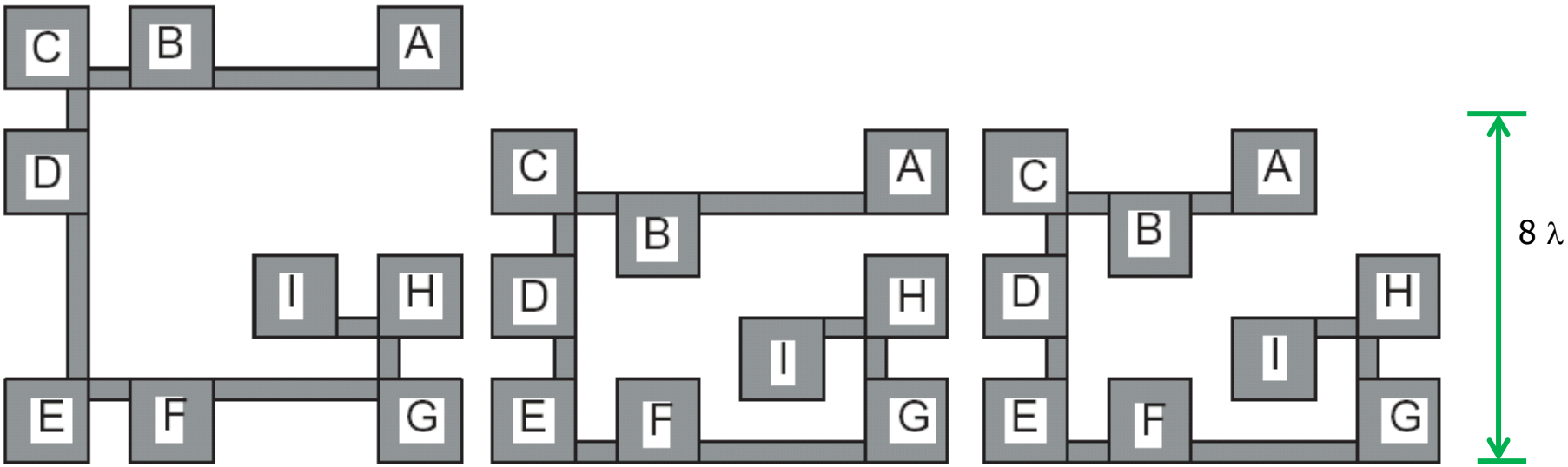
1D Compaction: *X* Followed By *Y*

- 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.



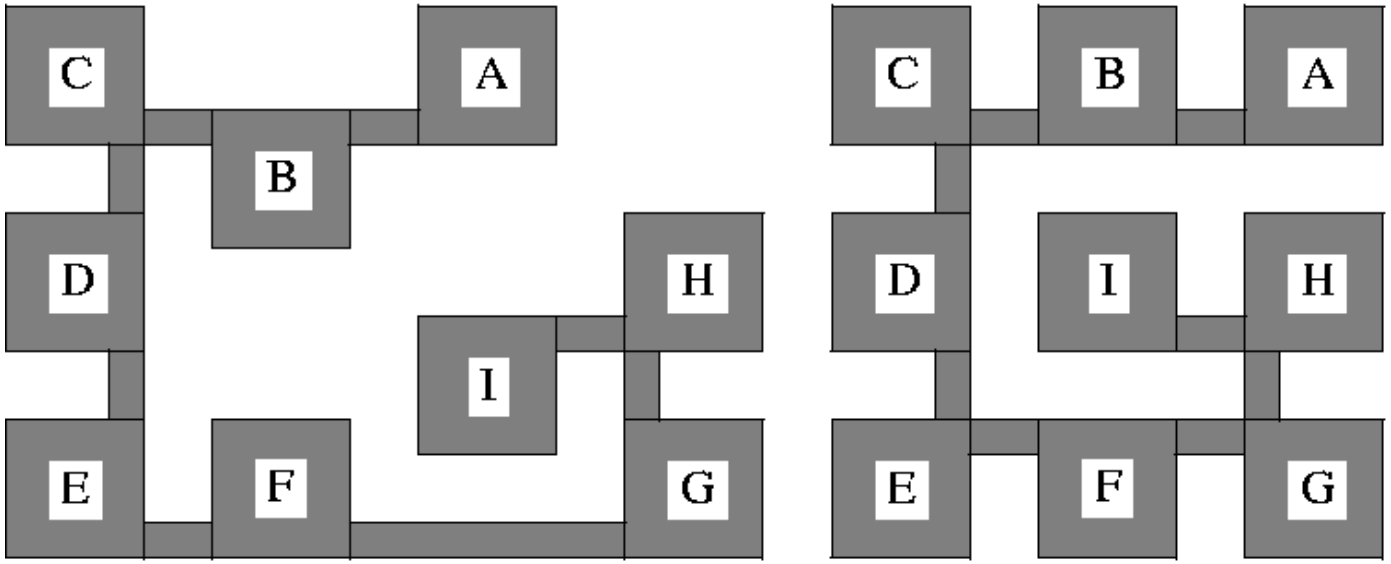
1D Compaction: *Y* Followed By *X*

- 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.

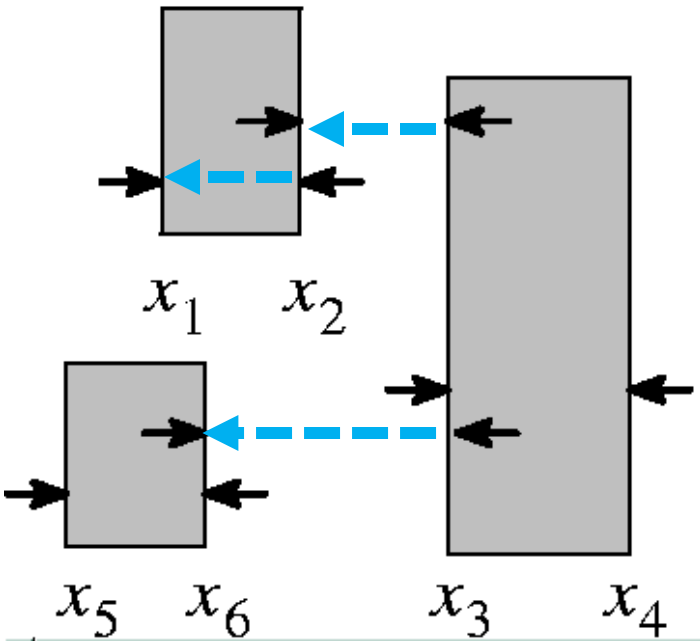


- Since 2D compaction is NP-complete, most compactors are based on repeated 1D compaction.

Inequalities for Distance Constraints

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

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

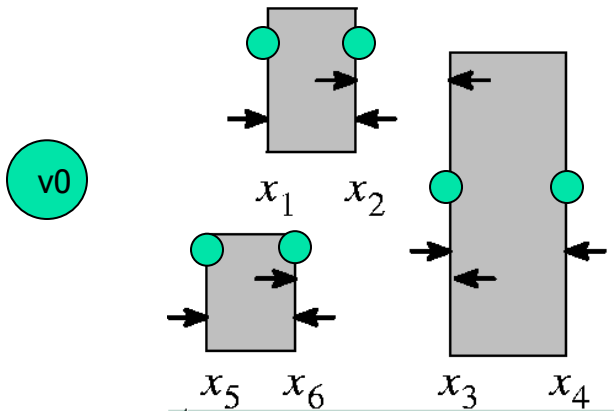


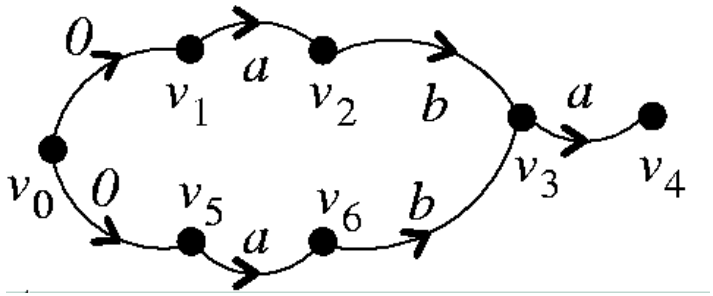
- For example, if the minimum width is *a* and the minimum separation is *b*, then

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

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.

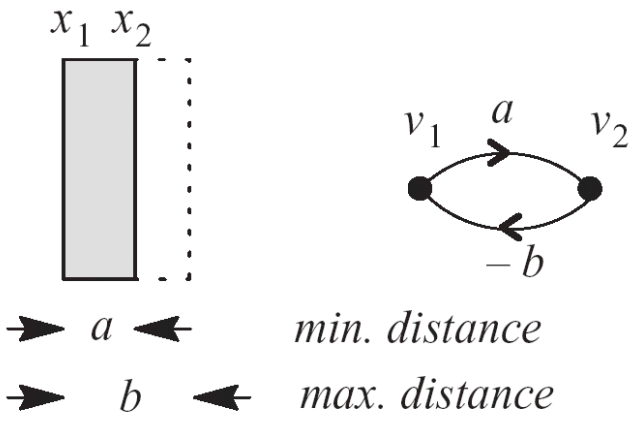


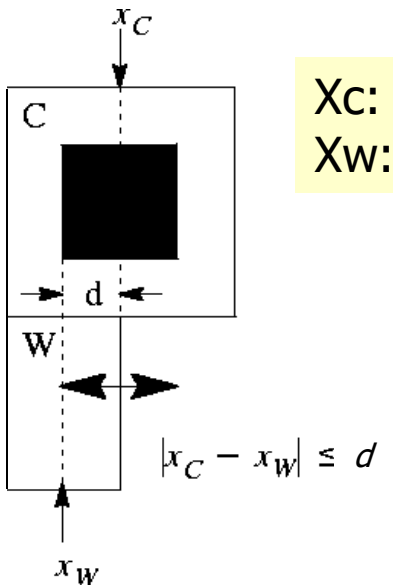


constraint graph

Maximum-Distance Constraints

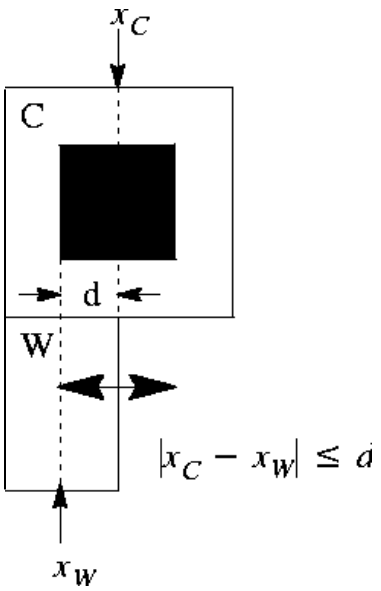
- Sometimes the distance of layout elements is bounded by a maximum, e.g., when the user wants a **maximum wire width**, maintains a wire connecting to a via, etc.
  - A maximum distance constraint gives an inequality of the form: $x_j - x_i \leq c_{ij}$ or $x_i - x_j \geq -c_{ij}$
  - Consequence for 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.

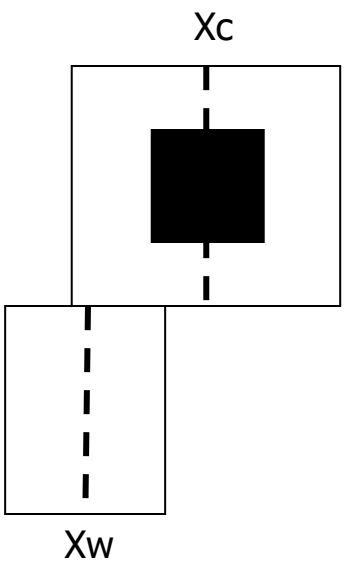




Xc: center of contact
Xw: center of wire

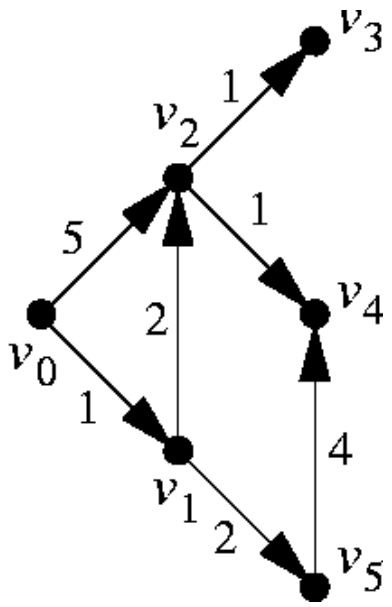
Xc: center of contact
Xw: center of wire





In fact, usually we seldom see this

DAG Longest-Path Example

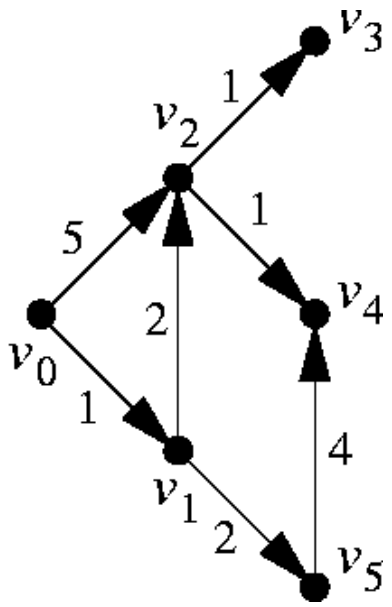


- Runs in a breadth-first search manner.
- p_i: in-degree of v_i.
- x_i: longest-path length from v_0 to v_i.
- Time complexity: $O(V+E)$
- Q: queue

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

v0→v1: dist 1 (and v1 in-degree decremented)
v0→v2: dist 5 (and v2 in-degree decremented)
v1 and v2 will be added to Q

DAG Longest-Path Example

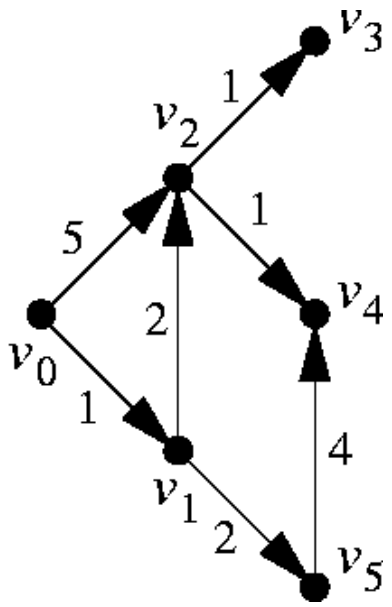


- Runs in a breadth-first search manner.
- p_i: in-degree of v_i.
- x_i: longest-path length from v_0 to v_i.
- Time complexity: $O(V+E)$
- Q: queue

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
$\{v_1\}$ $\sqrt{2}$	0	0	1	2	0	1	5	0	0	3

v1→v5: dist 1+2=3 (& v5 in-degree decremented)
v1→v2: dist 1+2<5 (& v2 in-degree decremented)
v5 will be added to Q

DAG Longest-Path Example

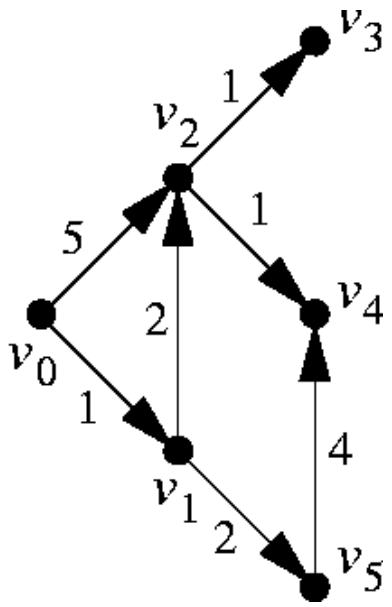


- Runs in a breadth-first search manner.
- p_i: in-degree of v_i.
- x_i: longest-path length from v_0 to v_i.
- Time complexity: $O(V+E)$
- Q: queue

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

v2→v3: dist 5+1=6 (& v3 in-degree decremented)
v2→v4: dist 5+1=6 (& v4 in-degree decremented)
v3,v4 will be added to Q; next we expand from v5

DAG Longest-Path Example

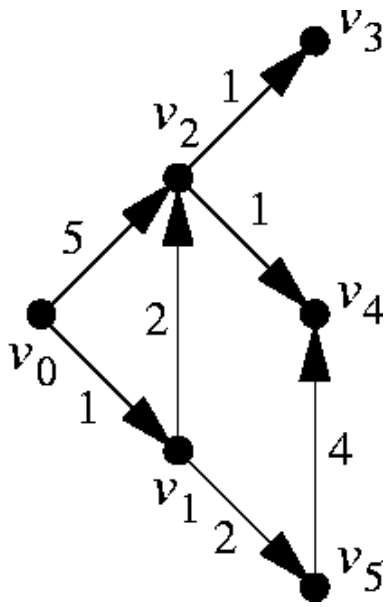


- Runs in a breadth-first search manner.
- p_i: in-degree of v_i.
- x_i: longest-path length from v_0 to v_i.
- Time complexity: $O(V+E)$
- Q: queue

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

v5→v4: dist 3+4=7>6 (& v4 in-degree decremented)

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: queue

Is depth-first search a good idea?

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-Path Algorithm for DAGs

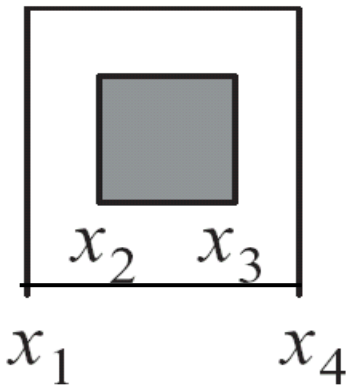
```
longest-path(G)
{
  for (i ← 1; i ≤ n; i ← i + 1)
    p_i ← "in-degree of v_i";
  Q ← {v0};
  while (Q ≠ ∅) {
    v_i ← "any element from Q";
    Q ← Q \{v_i};
    for each v_j "such that" (v_i, v_j) ∈ E {
      x_j ← max(x_j, x_i + d_ij);
      p_j ← p_j − 1;
      if (p_j ≤ 0)
        Q ← Q ∪ {v_j};
    }
  }
}
```

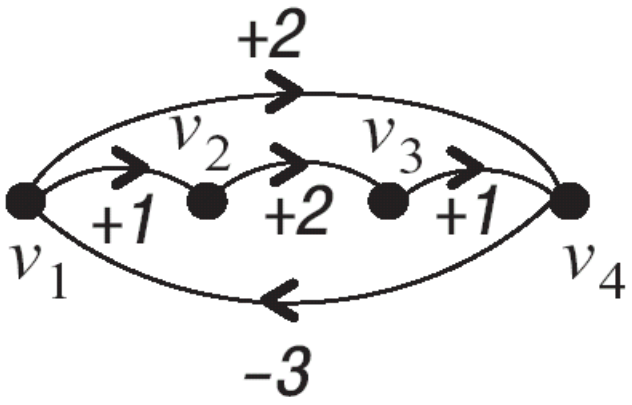
```
main ()
{
  for (i ← 0; i ≤ n; i ← i + 1)
    x_i ← 0;
  longest-path(G);
}
```

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

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.





Max dist

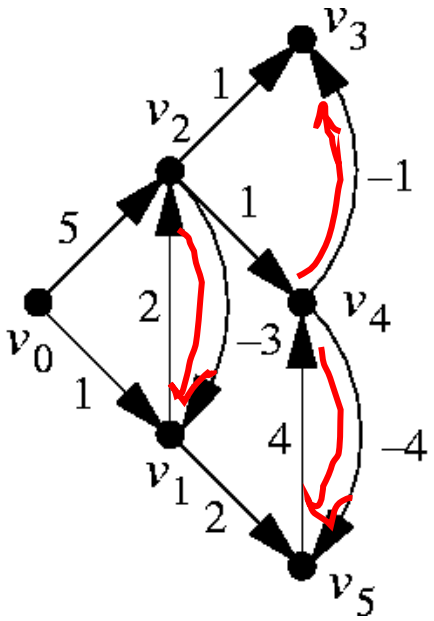
Just a heads up: Longest and Shortest Paths

- 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

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: $O(E_b \times E_f)$.

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

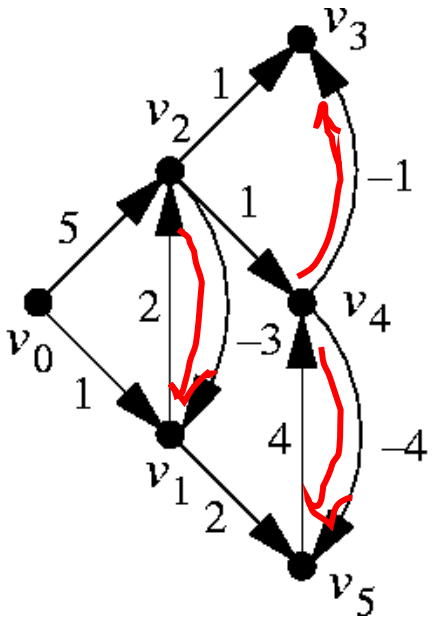
First, from v0, find longest paths to each node

x2-x1 <=3

x4-x3 <=1

x4-x5 <=4

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

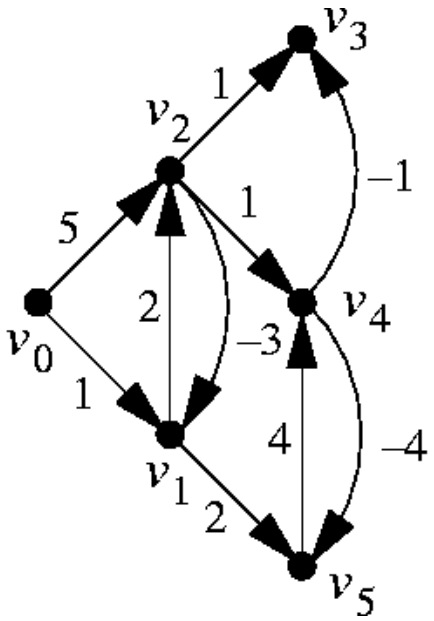
x2-x1 <=3

x4-x3 <=1

x4-x5 <=4

First, from v0, find longest paths to each node

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

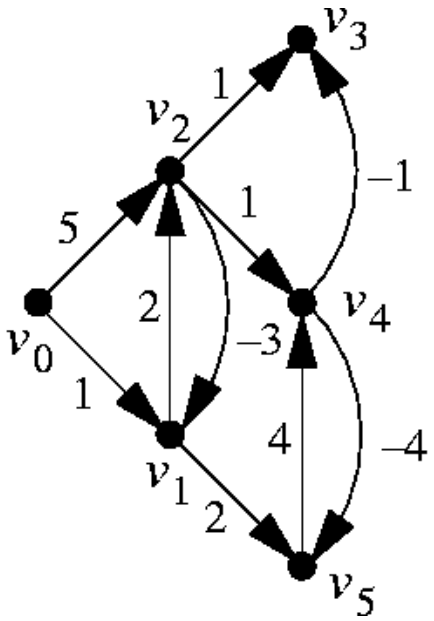
x2-x1 <= 3

x4-x3 <= 1

x4-x5 <= 4

Consider 1st backward edge (v2→v1): 5+(-3)=2 new v1 distance.
Then, re-do longest paths in Ef with v1 at cost=2

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

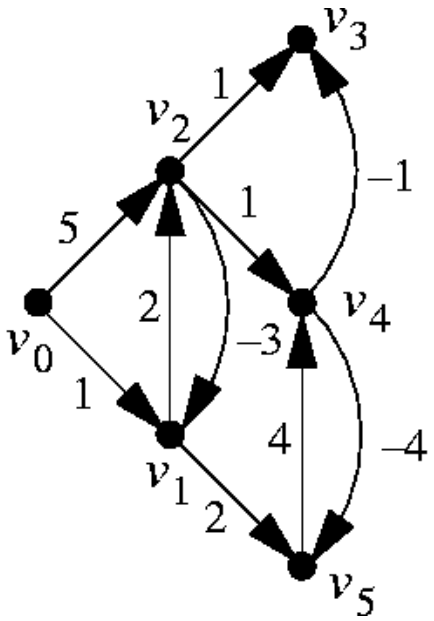
x2-x1 <= 3

x4-x3 <= 1

x4-x5 <= 4

v5 new dist=2+2=4, v2 dist unchanged
v4 new dist=4+4=8, v3 dist unchanged
Then, consider another backward edge (v4-v3)

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

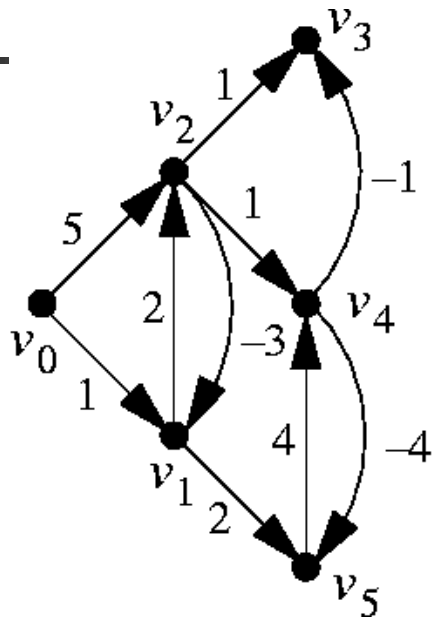
x2-x1 <= 3

x4-x3 <= 1

x4-x5 <= 4

Then, consider another backward edge (v4->v3)
v3 new dist=8-1=7 >6 → changed; then, re-do LP

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

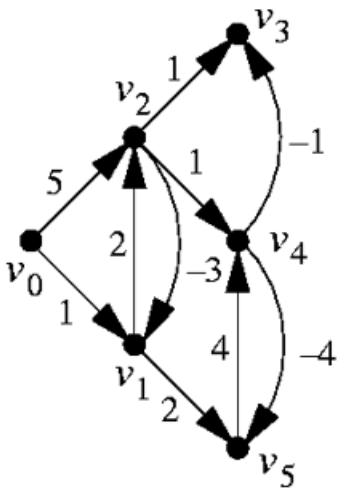
x2-x1 <= 3

x4-x3 <= 1

x4-x5 <= 4

Consider v4→v5: 8-4=4 → v5 dist unchanged

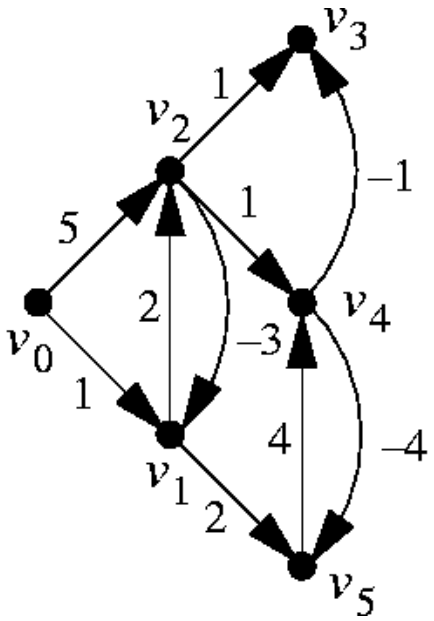
Starting From Different Backward Edges?



- The previous table of steps may let you think we just pick a backward edge one at a time
- But that will lead to a question – from which backward edge will we start?
- In previous example, we know (v2→v1) backward edge changes the longest path value at node v1; then, we need to propagate the new values

	step	x1	x2	x3	x4	x5			
	initialize	neg	neg	neg	neg	neg			
	forward	1	5	6	7	3			
v4->v5	backward-1	1	5	6	7	3			
	forward-2	1	5	6	7	3		no change	
v4->v3	backward-2	1	5	6	7	3			
	forward-3	1	5	6	7	3		no change	
v2->v1	backward-3	2	5	6	8	4		changes propagated	
	no backward edge unvisited; but we saw x3 is different, unless we need to use backward edge from (v4 to v3) to update x3 to 7. In other words, we need to keep looping thru backward edges								

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 **all** the edges from E_b (**not just one**)
- Time complexity: $O(E_b \times E_f)$.

x2-x1 <= 3

x4-x3 <= 1

x4-x5 <= 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

A More Correct Way To Show The Steps

- After each longest path search, we need to process all the backward edges to get the new values; hence, table will be like:

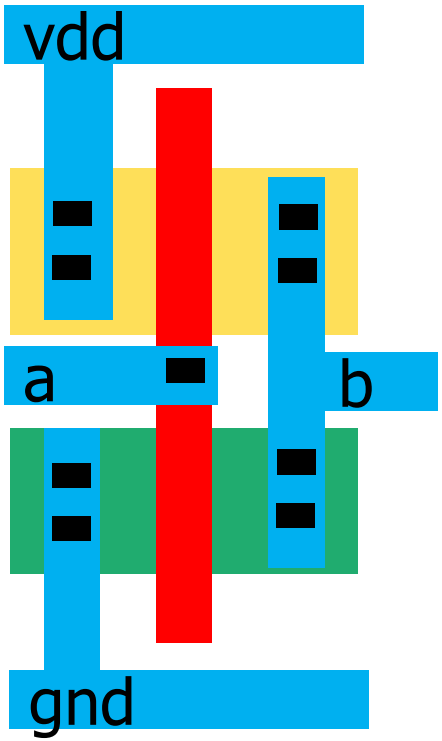
Step	x_1	x_2	x_3	x_4	x_5
Initialize	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
Forward 1	1	5	6	7	3
Backward-1					
v2->v1	2	5	6	7	3
v4->v3	2	5	6	7	3
v4->v5	2	5	6	7	3
Forward-2	2	5	6	8	4
Backward-2					
v2->v1	2	5	6	8	4
v4->v3	2	5	7	8	4
v4->v5	2	5	7	8	4
Forward-3	2	5	7	8	4
Backward-3					
v2->v1	2	5	7	8	4
v4->v3	2	5	7	8	4
v4->v5	2	5	7	8	4

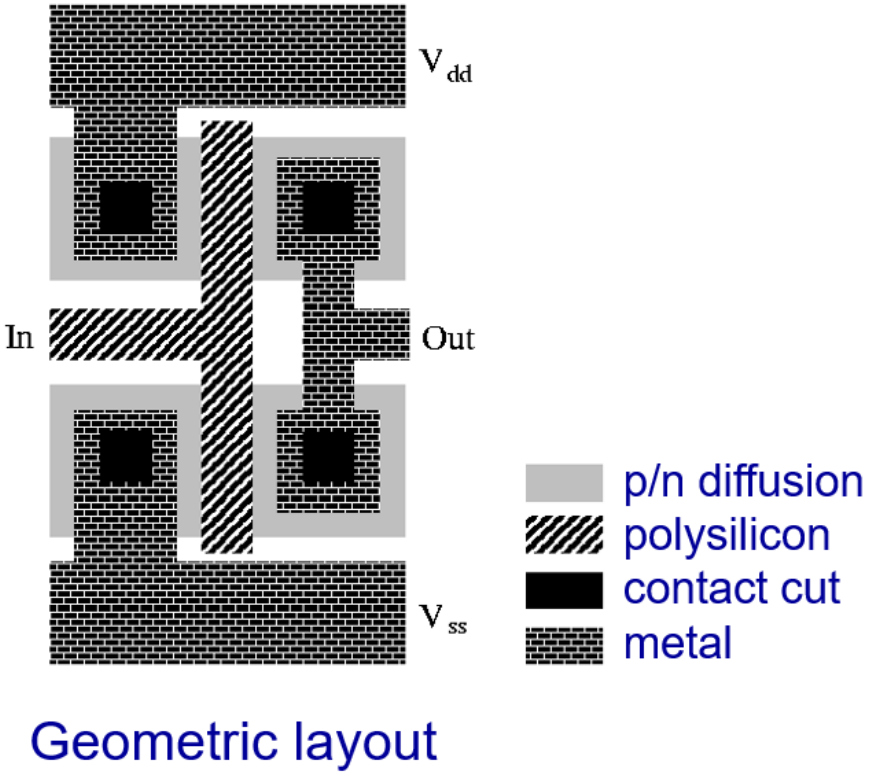
Pseudo Code: The Liao-Wong Algorithm

```
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);
```

If there is any change, continue

How About Applying To Layout Examples? (1)





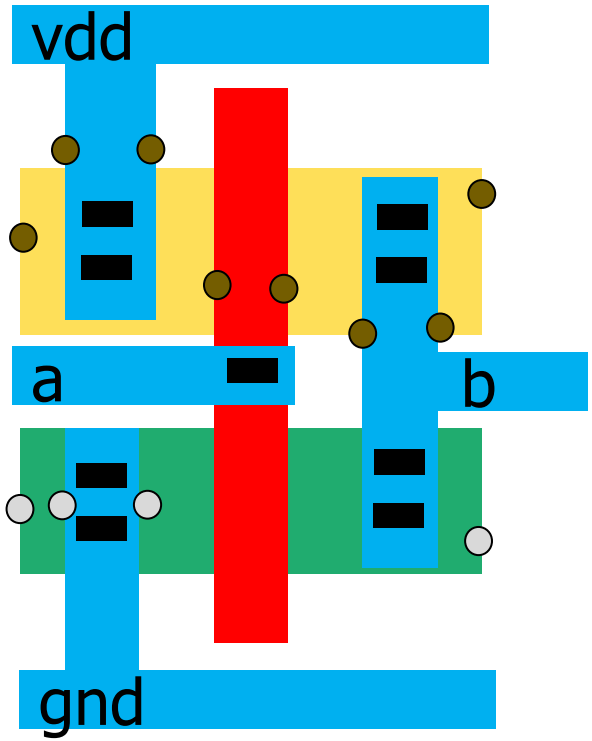
Geometric layout

Can we list all the necessary design rules we need to consider? But this inverter layout consists of poly, diffusion, … (base layers shapes).

How About Applying To Layout Examples? (1)

Critical: how to generate constraint graphs?

Let's try to draw vertices



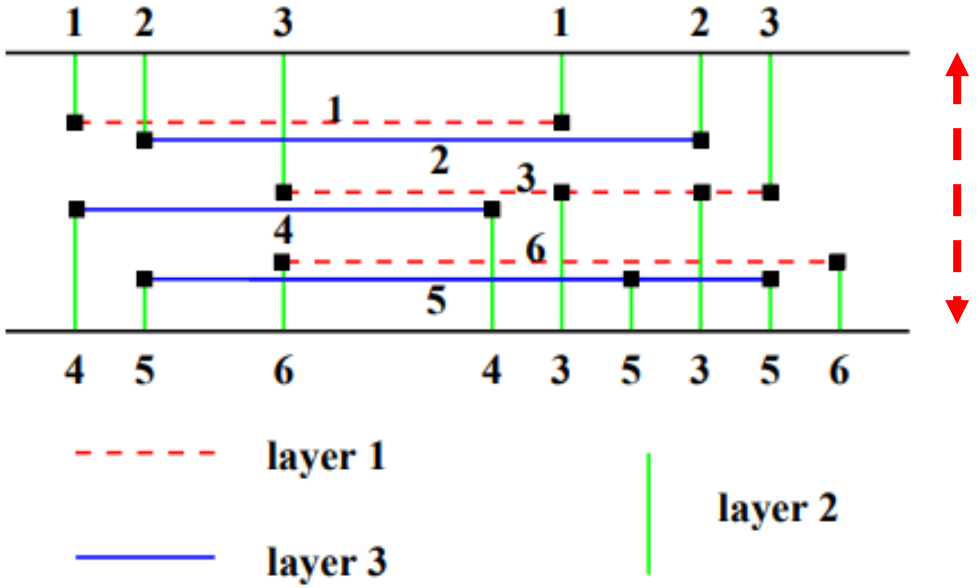
8 vertices on the pmos side
4 on the nmos side
How about contacts?
Forward edges

How about backward edges?

Usually leaf cell layout is customized.
Just give this a thought from compaction point of view

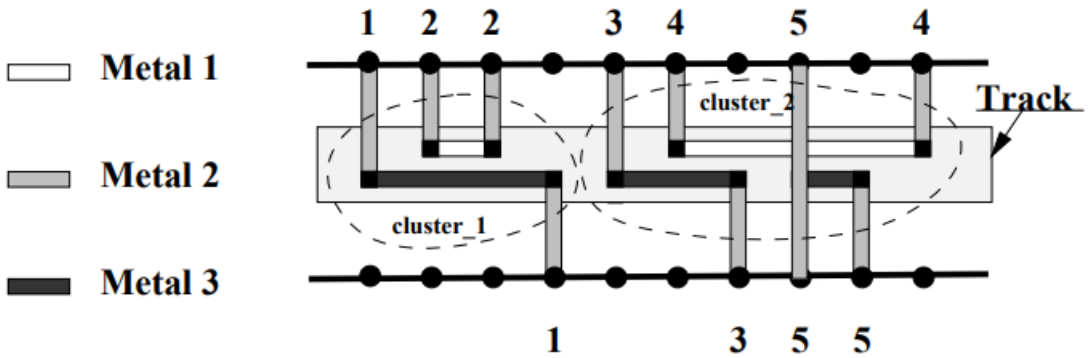
Maintaining contacts in a desirable way is not easy

How About Channel Routing?



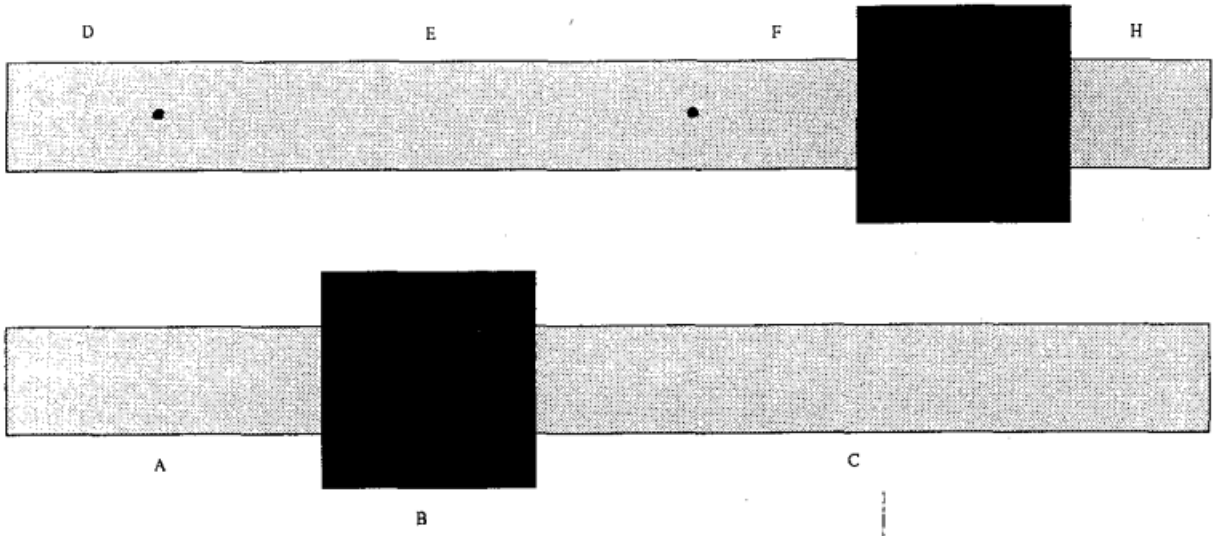
3 layers channel routing:
Pin locations are fixed.
Compaction in Y-direction

(a)



(a) A channel routing layout.

Prior Efforts



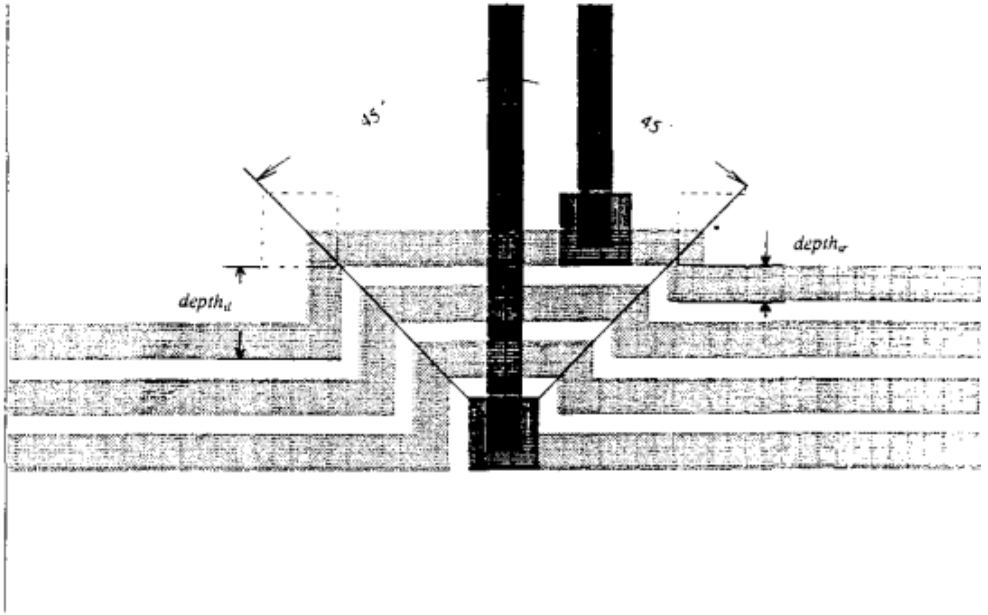


Fig. 4. Bump propagation.

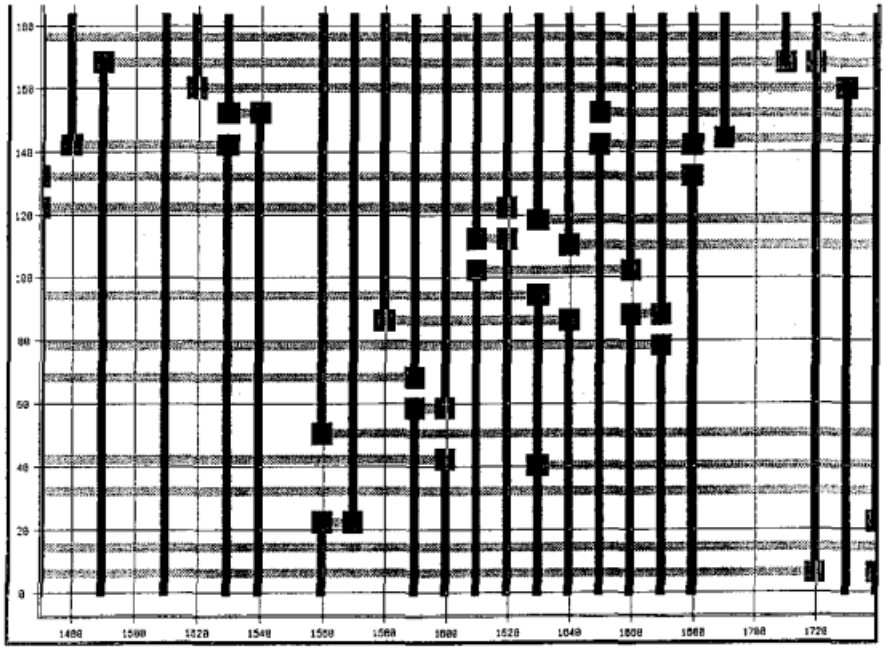


Fig. 5. Initial input.

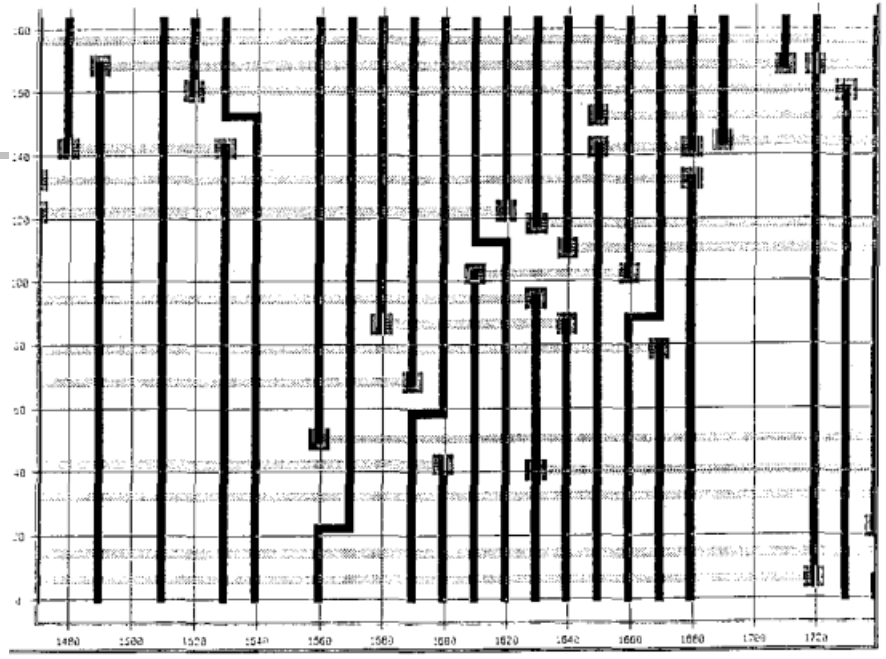


Fig. 6. Result of via minimization.

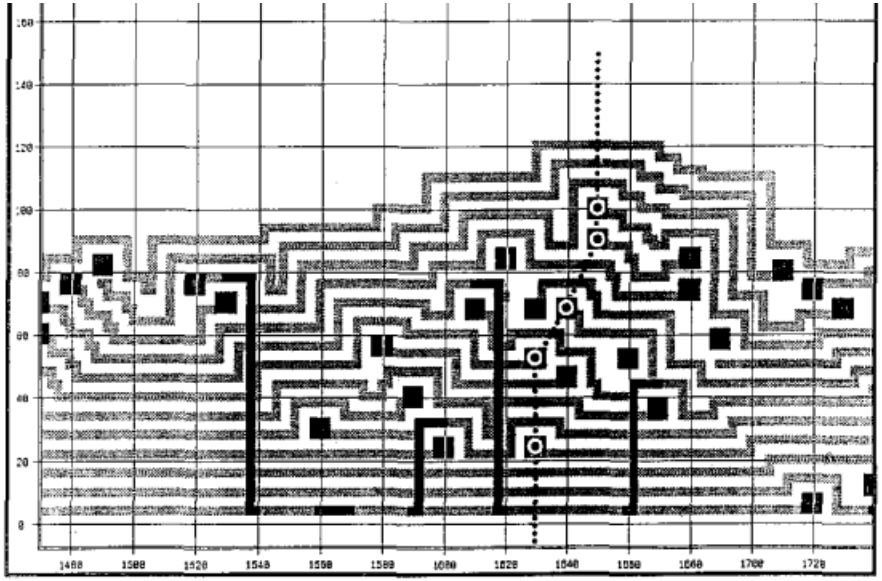


Fig. 7. Result of compaction downward. The height is 125 units. The dotted curve in the middle illustrates the critical path.

Applying Compaction To Building Block Layout

- A commercial tool was developed (around 1988)
  - Placing blocks
  - Symbolic routing first, followed by the compaction to generate the final layout coordinates
  - Pictures below was to illustrate BBL only

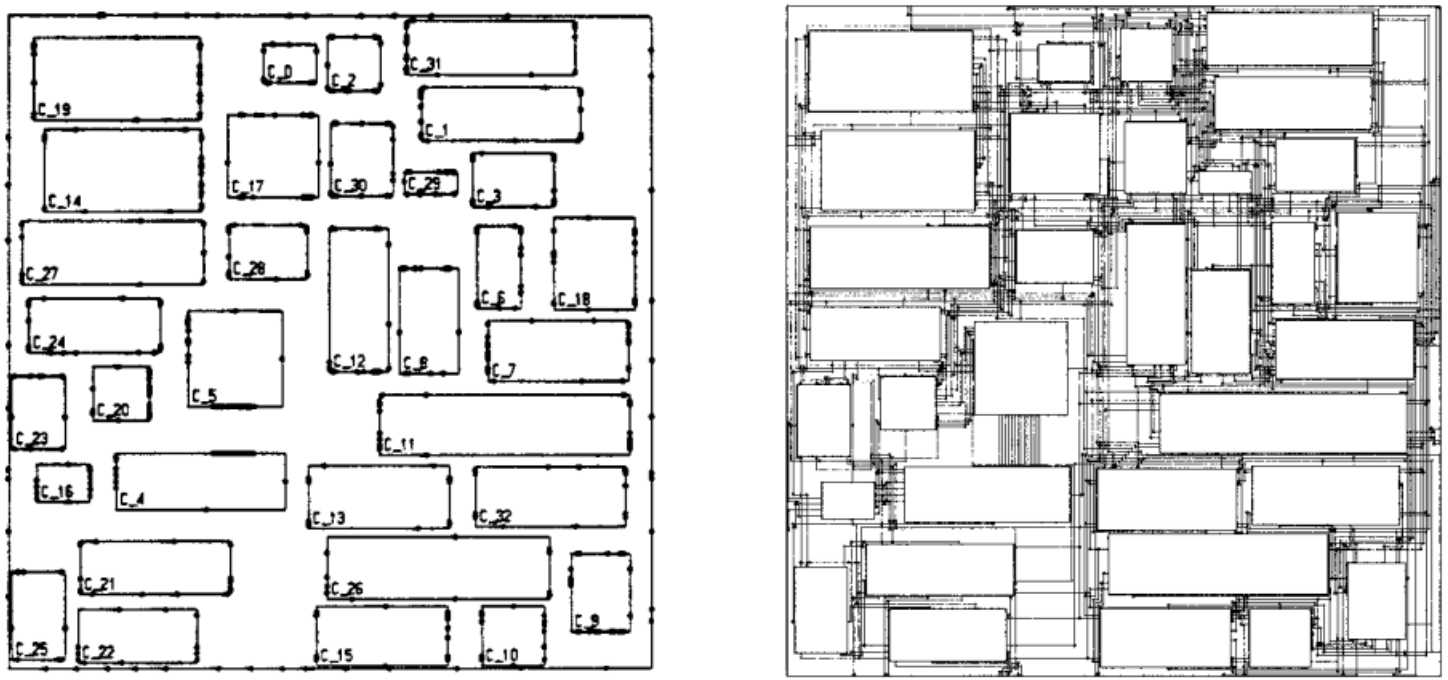


Figure 7: *Ami33*(BBL2) benchmark example.
Left; after placement. Right; after detailed routing.

The Bellman-Ford Algorithm for Longest Paths

- An alternative to the Liao-Wong algorithm is Bellman-Ford algorithm
- Solves the case where edge weights can be negative.
- Returns FALSE if there exists a cycle reachable from the source; TRUE otherwise.
- Time complexity: $O(VE)$.

https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

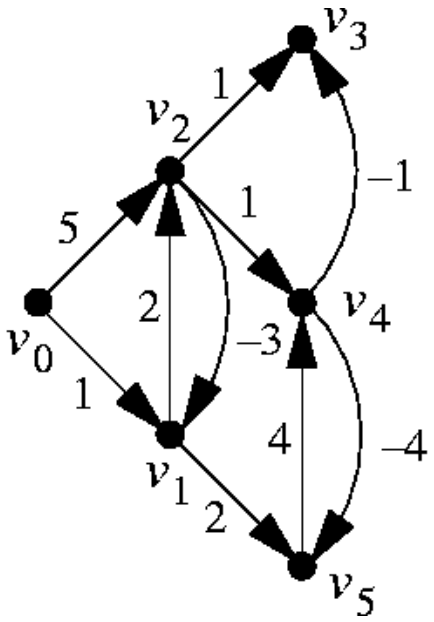
Bellman–Ford proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution.

The Bellman-Ford Algorithm for Longest Paths

```
for (i ← 1; i ≤ n; i ← i + 1)
  x_i ← −∞;
x_0 ← 0;
count ← 0;
S_1 ← {v_0};
S_2 ← ∅;
while (count ≤ n && S_1 ≠ ∅) {
    for each v_i ∈ S_1
      for each v_j "such that" (v_i, v_j) ∈ 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 ← ∅;
    count ← count + 1;
}
if (count > n)
  error("positive cycle");
```

Maintaining two sets

Example of Bellman-Ford for Longest Paths



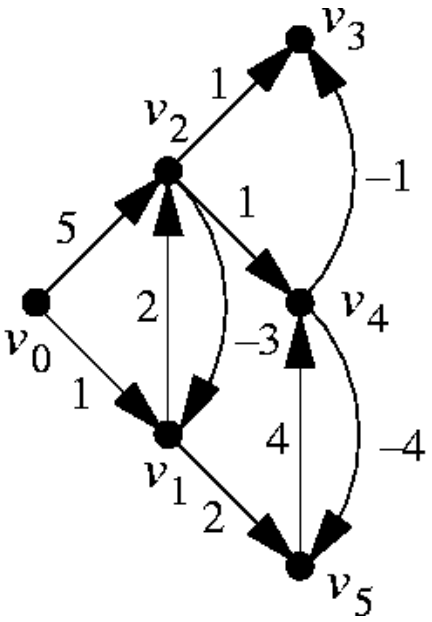
- Repeated "wave front propagation."
- S_1: the current wave front.
- x_i: longest-path length from v_0 to v_i.
- After k iterations, it computes the longest-path values for paths going through k-1 intermediate vertices.
- Time complexity: $O(VE)$.

S_1	x_1	x_2	x_3	x_4	x_5
"not initialized"	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$\{v_0\}$	1	5	$-\infty$	$-\infty$	$-\infty$

S1=(v0), v0->v2 (5), v0->v1 (1)
S2=(v1, v2)
S1 <- S2, S2=empty

$\{v_3\}$	2	5	7	8	4

Example of Bellman-Ford for Longest Paths



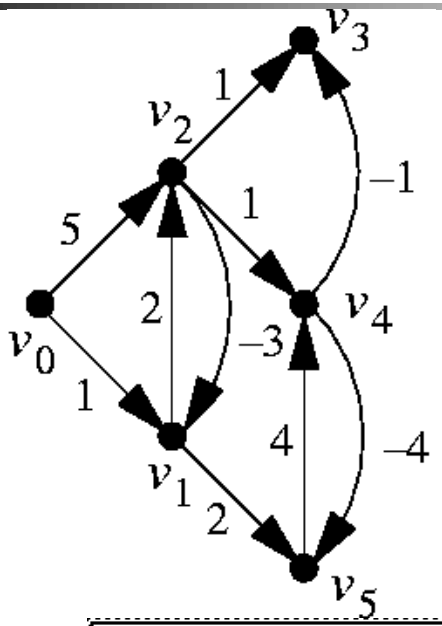
- Repeated "wave front propagation."
- S_1: the current wave front.
- x_i: longest-path length from v_0 to v_i.
- After k iterations, it computes the longest-path values for paths going through k-1 intermediate vertices.
- Time complexity: $O(VE)$.

Two sets

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

S1=(v1,v2),
from v1, v1->v2(=1+2<5), v1->v5 (=1+2, add to S2)
from v2, v2->v1 (=5-3 >1, add to S2), v2->v3(6), v2->v4(6),
S2=(v1, v3, v5, v4)
S1 <- S2, S2=empty

Example of Bellman-Ford for Longest Paths

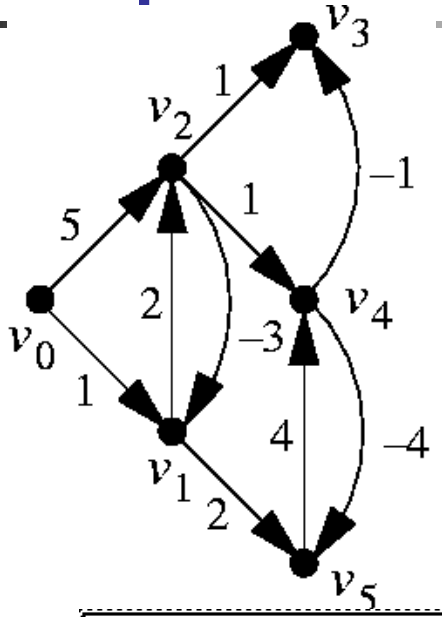


- Repeated "wave front propagation."
- S_1: the current wave front.
- x_i: longest-path length from v_0 to v_i.
- After k iterations, it computes the longest-path values for paths going through k-1 intermediate vertices.
- Time complexity: $O(VE)$.

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

S1=(v1,v3, v4,v5),
from v5, v5->v4(=3+4=7>6, add v4 to S2)
from v4, v4->v3(=7-1=6), v4->v5(=7-4)
from v1, v1->v5(=2+2=4>3, add v5 to S2), v1->v2(=2+2<5),
from v3, none; S2=(v5, v4), S1<- S2, S2=empty

Example of Bellman-Ford for Longest Paths

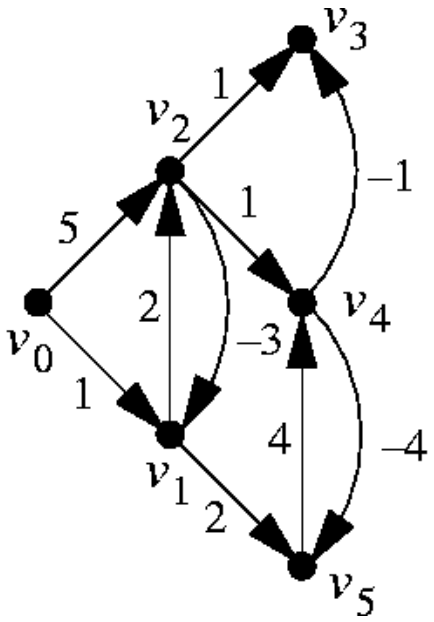


- Repeated "wave front propagation."
- S_1: the current wave front.
- x_i: longest-path length from v_0 to v_i.
- After k iterations, it computes the longest-path values for paths going through k-1 intermediate vertices.
- Time complexity: $O(VE)$.

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

S1=(v4,v5), from v4, v4->v3(=7-1), v4->v5(=7-4)
from v5, v5->v4(=4+4=8>7, add v4 to S2)
S1<-S2 (v4), S2 empty

Example of Bellman-Ford for Longest Paths

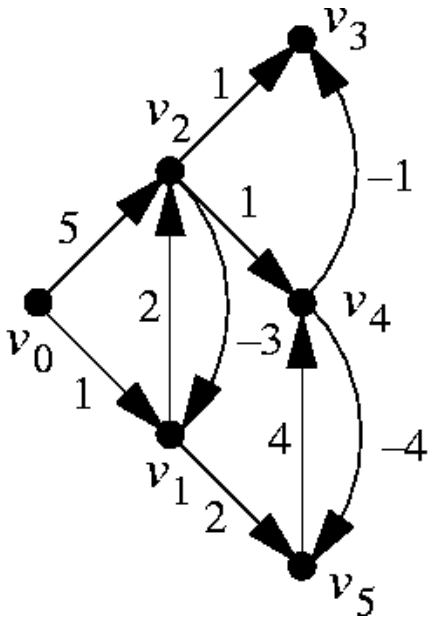


- Repeated "wave front propagation."
- S_1: the current wave front.
- x_i: longest-path length from v_0 to v_i.
- After k iterations, it computes the longest-path values for paths going through k-1 intermediate vertices.
- Time complexity: $O(VE)$.

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

S1=(v4), v4->v3(=8-1=7 >6, add v3 to S2),

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

S1=(v3), none

The Bellman-Ford Algorithm for Longest Paths

```
for (i ← 1; i ≤ n; i ← i + 1)
  x_i ← −∞;
x_0 ← 0;
count ← 0;
S_1 ← {v_0};
S_2 ← ∅;
while (count ≤ n && S_1 ≠ ∅) {
    for each v_i ∈ S_1
      for each v_j "such that" (v_i, v_j) ∈ 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 ← ∅;
    count ← count + 1;
}
if (count > n)
  error("positive cycle");
```

Maintaining two sets

Able to detect cycle

The Bellman-Ford Algorithm for Shortest Paths

https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

https://www.youtube.com/watch?v=lyw4FaxrwHg

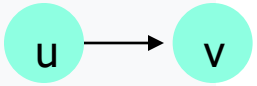
A good video to explain this shortest path and detect cycle

```
function BellmanFord(list vertices, list edges, vertex source) is

    // This implementation takes in a graph, represented as
    // lists of vertices (represented as integers [0..n-1]) and edges,
    // and fills two arrays (distance and predecessor) holding
    // the shortest path from the source to each vertex

    distance := list of size n
    predecessor := list of size n

    // Step 1: initialize graph
    for each vertex v in vertices do

        distance[v] := inf             // Initialize the distance to all vertices to infinity
        predecessor[v] := null         // And having a null predecessor

    distance[source] := 0              // The distance from the source to itself is, of course, zero
```

The Bellman-Ford Algorithm for Shortest Paths

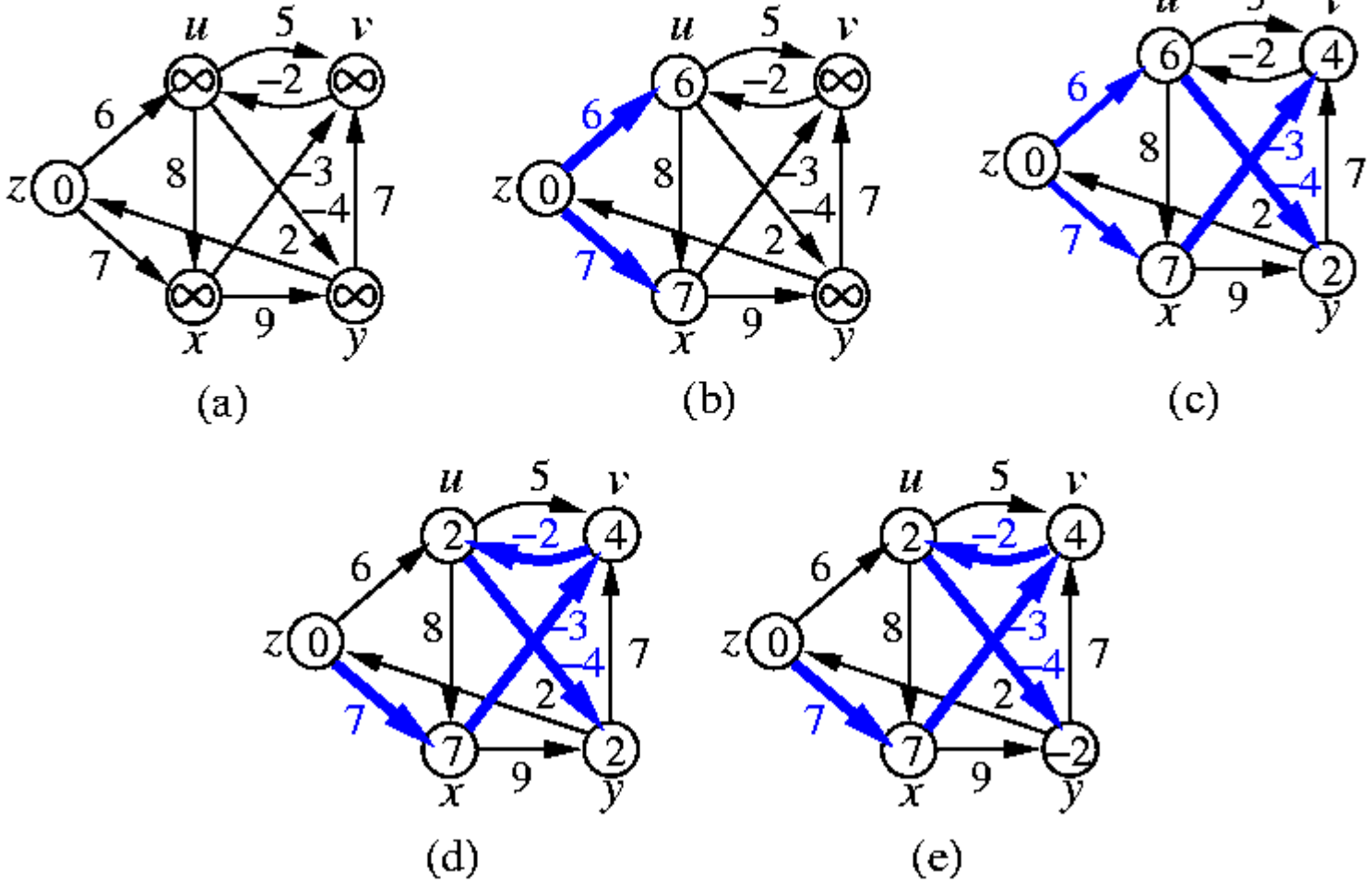
https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

```
// Step 2: relax edges repeatedly

repeat |V|-1 times:
     for each edge (u, v) with weight w in edges do
         if distance[u] + w < distance[v] then
             distance[v] := distance[u] + w
             predecessor[v] := u

// Step 3: check for negative-weight cycles
for each edge (u, v) with weight w in edges do
    if distance[u] + w < distance[v] then
        // Step 4: find a negative-weight cycle
        negativeloop := [v, u]
        repeat |V|-1 times:
            u := negativeloop[0]
            for each edge (u, v) with weight w in edges do
                if distance[u] + w < distance[v] then
                    negativeloop := concatenate([v], negativeloop)
        find a cycle in negativeloop, let it be ncycle
        // use any cycle detection algorithm here
        error "Graph contains a negative-weight cycle", ncycle

return distance, predecessor
```



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 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
```

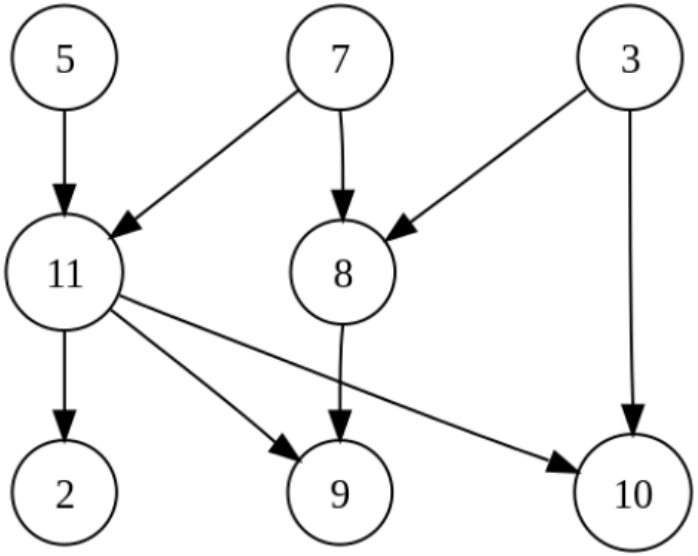
- An alternative to the Liao-Wong algorithm is the Bellman-Ford **longest** path algorithm
- Solves the case where edge weights can be negative**.**
- Returns FALSE if there exists a cycle reachable from the source; TRUE otherwise.
- Time complexity: $O(VE)$.

https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm
https://www.youtube.com/watch?v=lyw4FaxrwHg
(Worthwhile to watch)

Another Type Of Ordering

Topological Order

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. Precisely, a topological sort is a graph traversal in which each node *v* is visited only after all its dependencies are visited. A topological ordering is possible **if and only if** the graph has **no** directed cycles, that is, if it is a directed **acyclic** graph (DAG). Topological sorting has many applications especially in ranking problems such as feedback arc set. Topological sorting is possible even when the DAG has disconnected components.



The graph shown to the left has many valid topological sorts, including:

- 5, 7, 3, 11, 8, 2, 9, 10 (visual top-to-bottom, left-to-right)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

https://en.wikipedia.org/wiki/Topological_sorting

Topological Sort (Using Depth-First Search)

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

```
L ← Empty list that will contain the sorted nodes
while exists nodes without a permanent mark do
    select an unmarked node n
    visit(n)

function visit(node n)
    if n has a permanent mark then
        return
    if n has a temporary mark then
        stop   (not a DAG)

    mark n with a temporary mark

    for each node m with an edge from n to m do
        visit(m)

    remove temporary mark from n
    mark n with a permanent mark
    add n to head of L
```

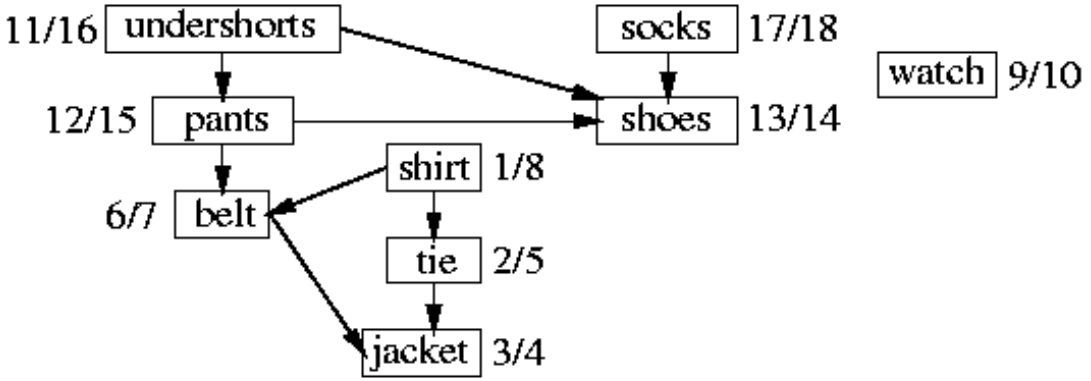
The topological ordering can also be used to quickly compute shortest paths through a weighted directed acyclic graph.

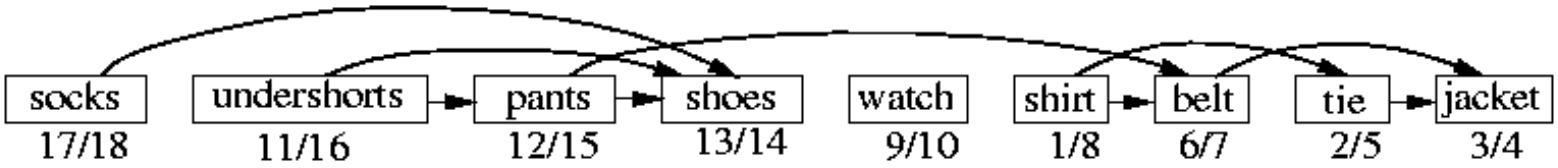
Topological Sort

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





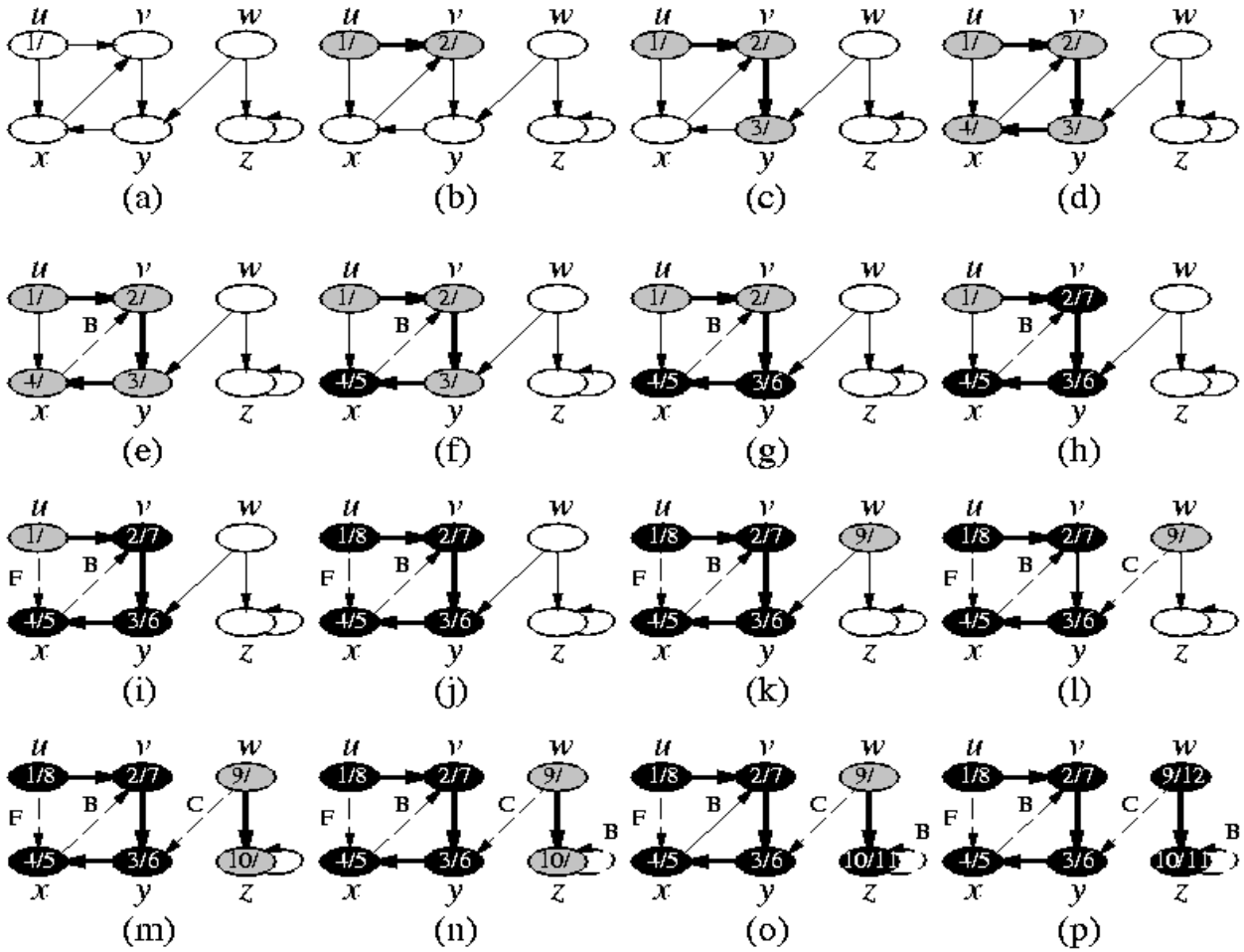
Vertices are arranged from left to right in order of decreasing finishing times.

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) → gray (discovered) → 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**: $G_\pi = (V, E_\pi)$, $E_\pi = \{(\pi[v], v) \in E \mid v \in V, \pi[v] \neq \text{NIL}\}$.

https://www.youtube.com/watch?v=7J3GadLzydI

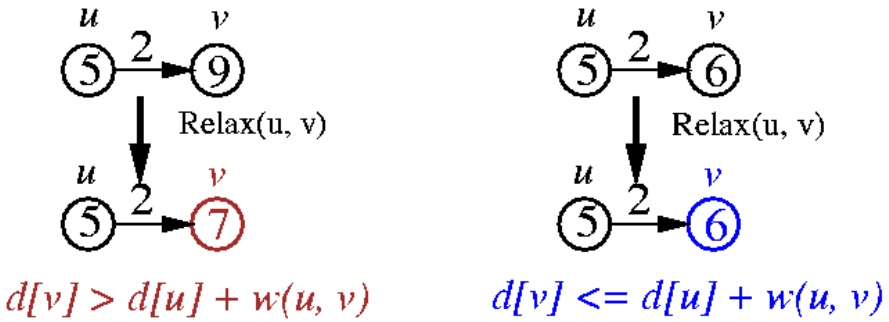
Topological Sort Visualized and Explained (easier)

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. $\pi[v] \leftarrow$ NIL; /* predecessor of v */
4. $d[s] \leftarrow 0$;

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

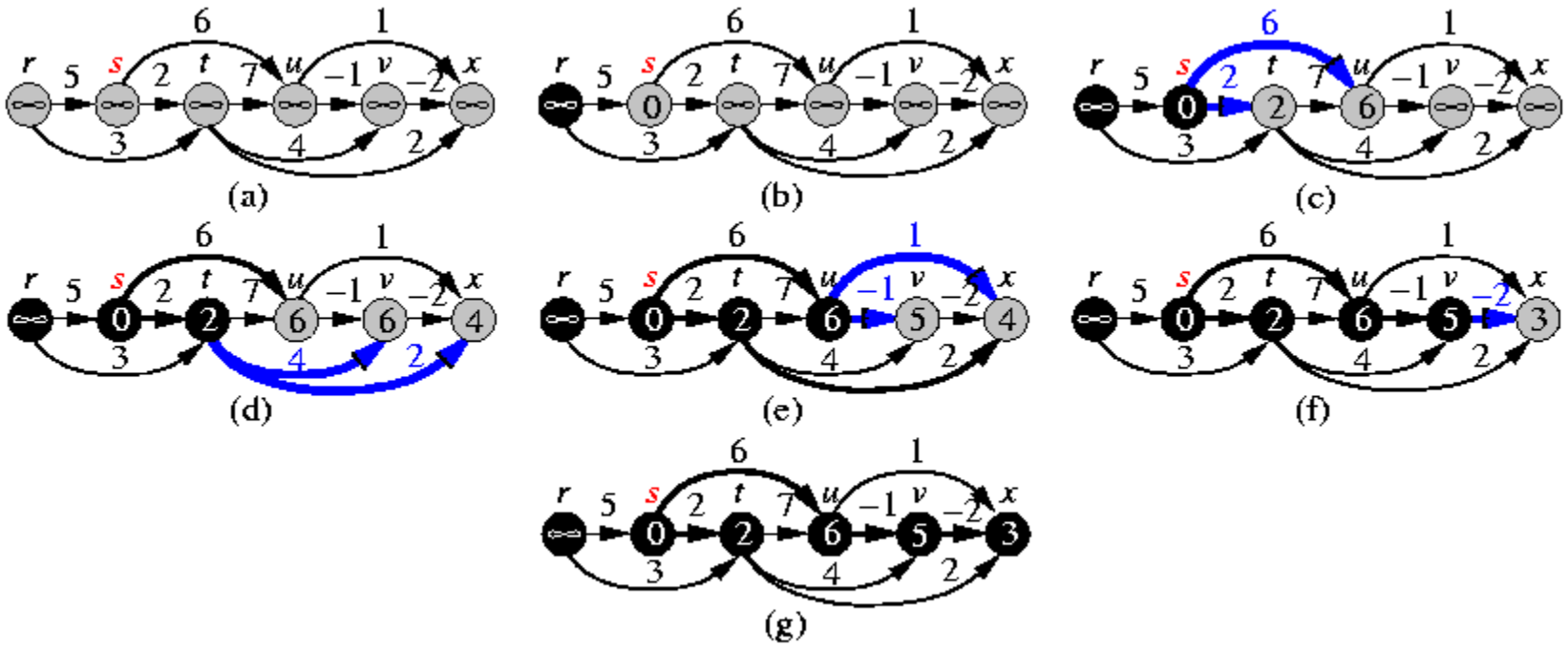
- $d[v] \leq d[u] + w(u, v)$ after calling Relax(u, v, w).
- $d[v] \geq \delta(s, v)$ during the relaxation steps; once $d[v]$ achieves its lower bound $\delta(s, v)$, it never changes.
- Let $s \rightsquigarrow u \rightarrow v$ be a shortest path. If $d[u] = \delta(s, u)$ prior to the call Relax(u,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



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).

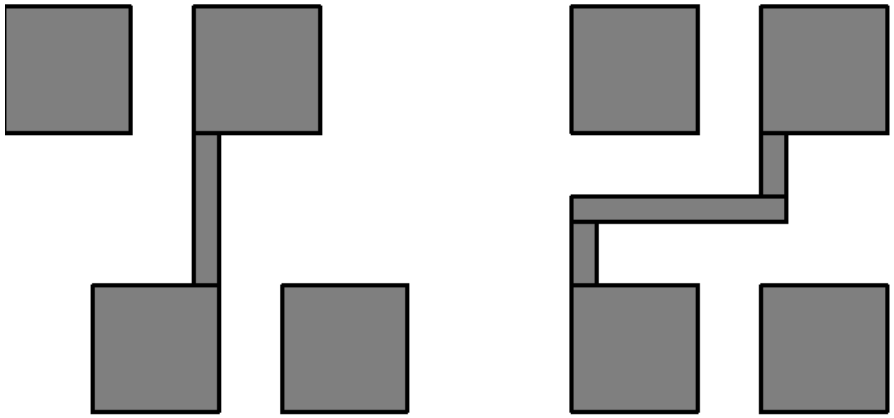


Summary: Longest and Shortest Paths

- 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

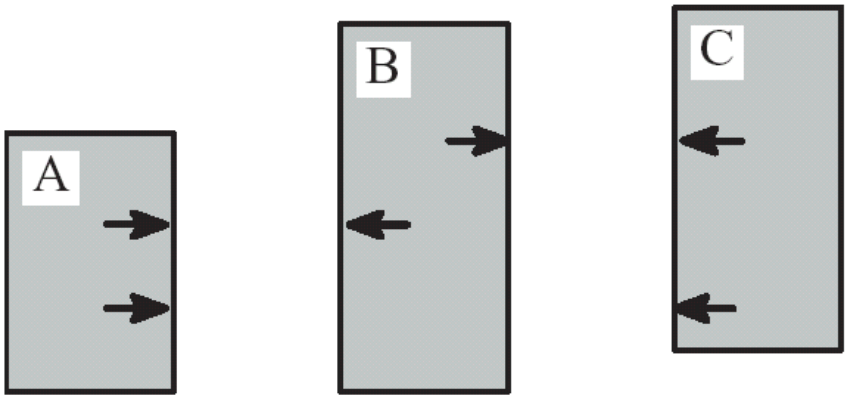
- Noncritical layout elements: Every element outside the critical paths has freedom on its best position => may use this freedom to optimize some cost function.
- Automatic jog insertion: The quality of the layout can further be improved by automatic jog insertion.



- 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.

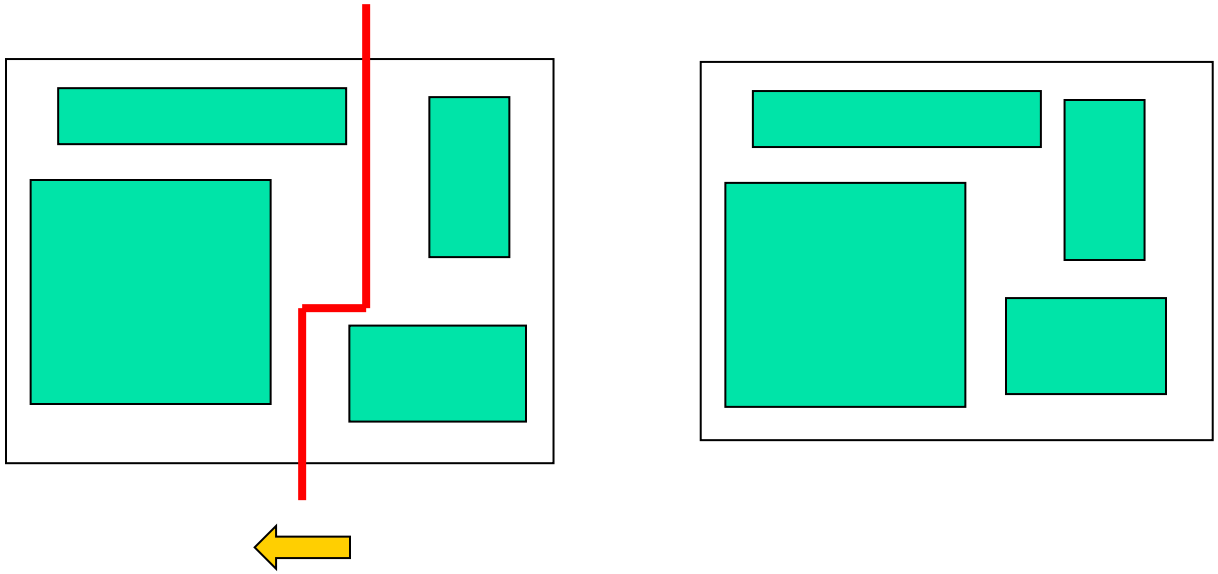


- Doenhardt and Lengauer have proposed a method for irredundant constraint generation with complexity $O(n \log n)$.

Summary So Far

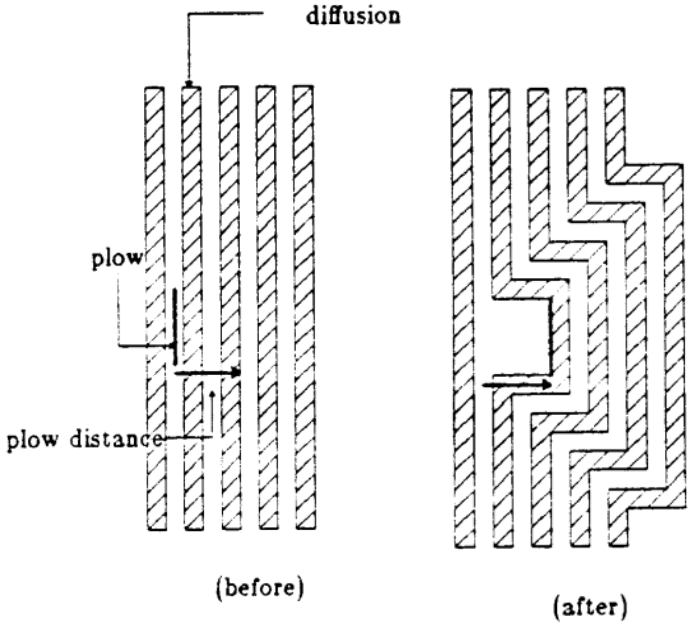
- 1D compaction (and 2D compaction – need heuristics)
- Constraint graphs → longest paths
- Forward edge graph, Backward edge graph
  - Solving compaction problem: Liao-Wong algorithm
- Bellman-Ford: considering both types of edges together

Interactive compaction; but maintaining connectivity is not easy



Some Layout Results After Compaction

Interactive compaction; but maintaining connectivity is not easy



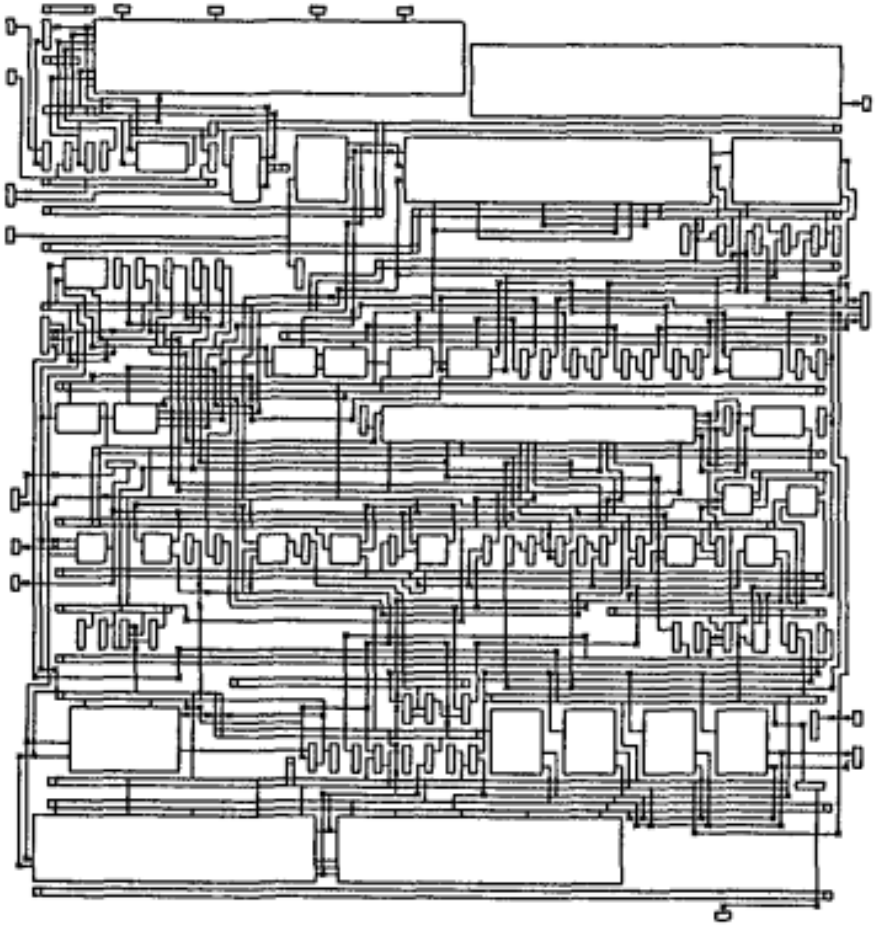


Fig.18 Layout compacted by using interactive compaction router

Course Ordering Of The Major Topics

- Digital design flow
- CMOS logic gates and Boolean equations
- Algorithms and complexity
- Basic logic synthesis, technology mapping (next)
- Compaction
- Partitioning (next)
- Floorplanning
- Placement
- Routing
- Clock/PG routing & Elmore delay
- Simulation
- High level synthesis