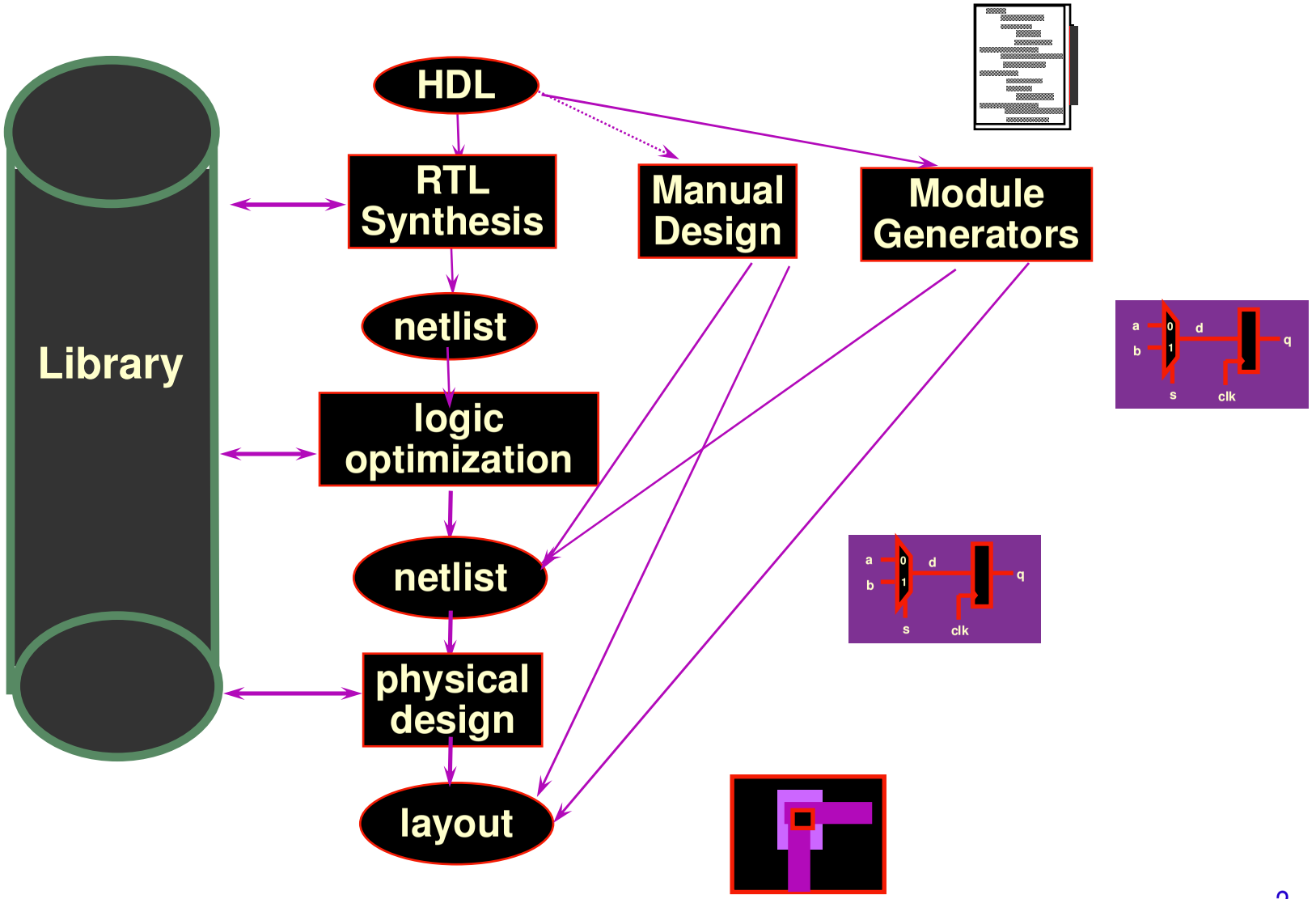
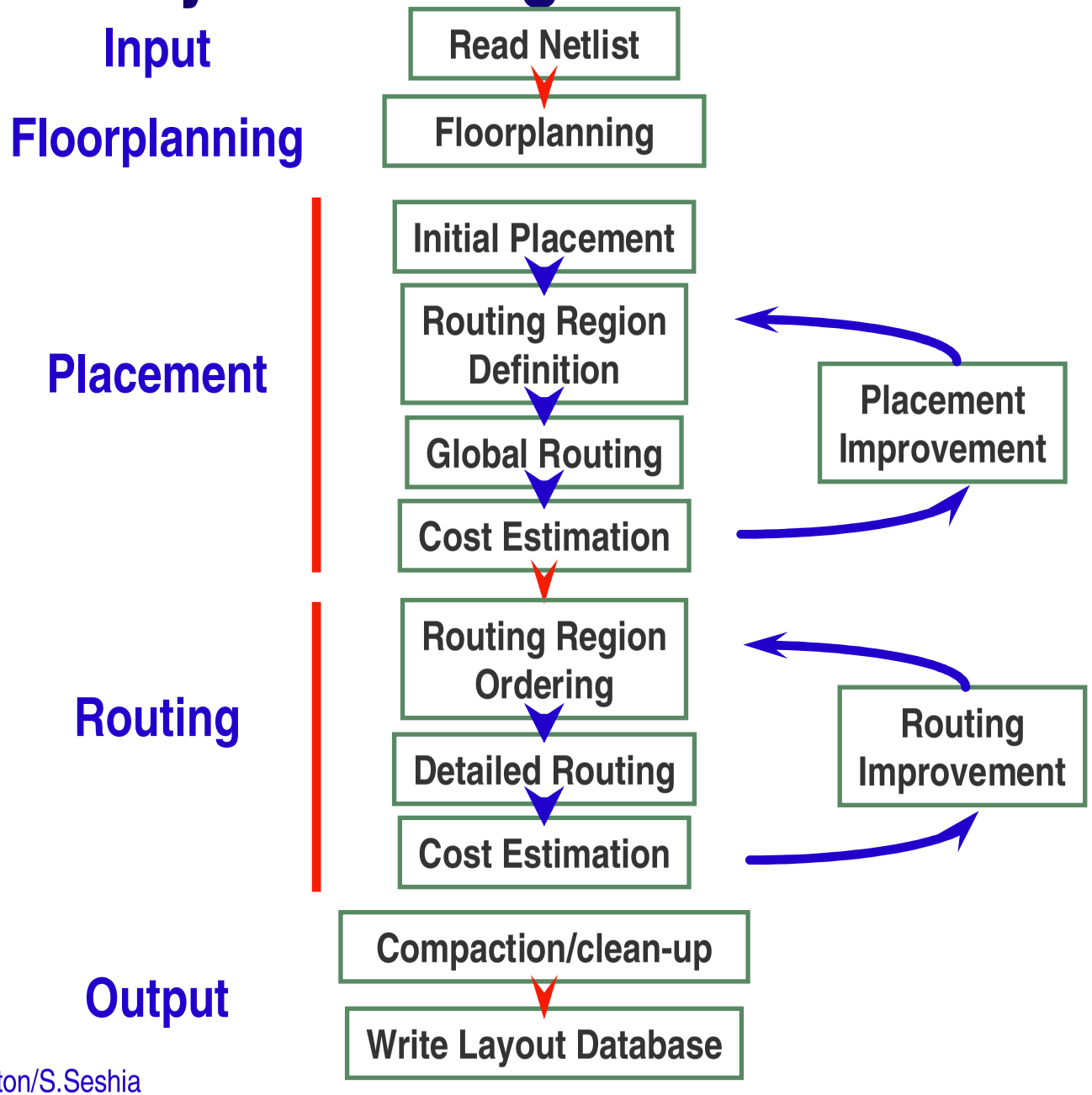
Layout Compaction

Prof. Kurt Keutzer
Prof. A. R. Newton
Prof. Sanjit Seshia
UC Berkeley
Prof. M. Orshansky
UCB → U of Texas

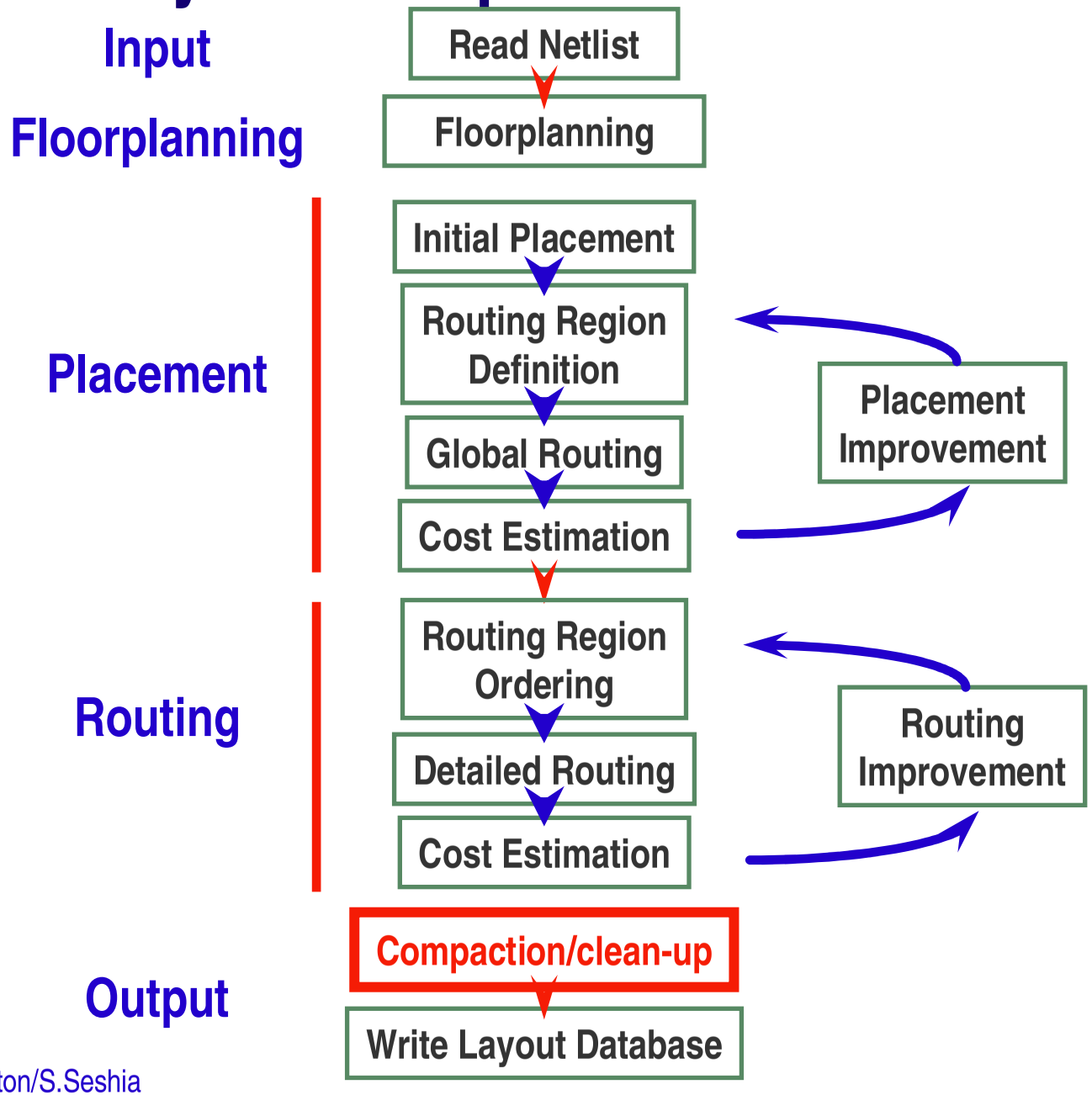
RTL Design Flow



Physical Design: Overall Flow



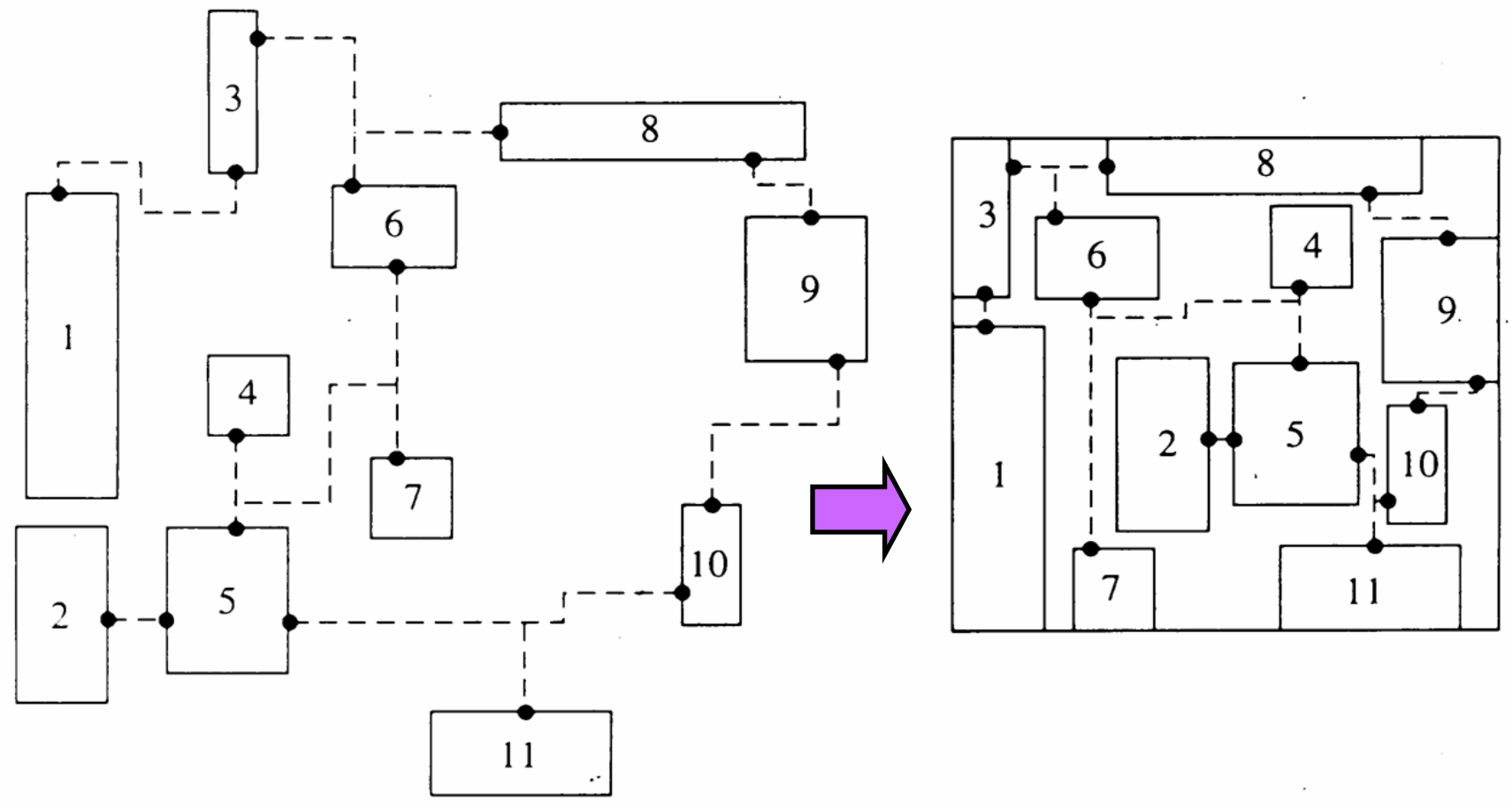
Layout Compaction



Compaction: Introduction

- After P&R, the layout is functionally complete
- Some vacant space may still be present in the layout
  - Due to non-optimality of P&R
- Compaction = removing vacant space
  - Improves cost, performance, and yield
- Key for high-performance full-custom layouts
- Standard cells – only channel heights may be minimized
  - But channel compactors are near-optimal

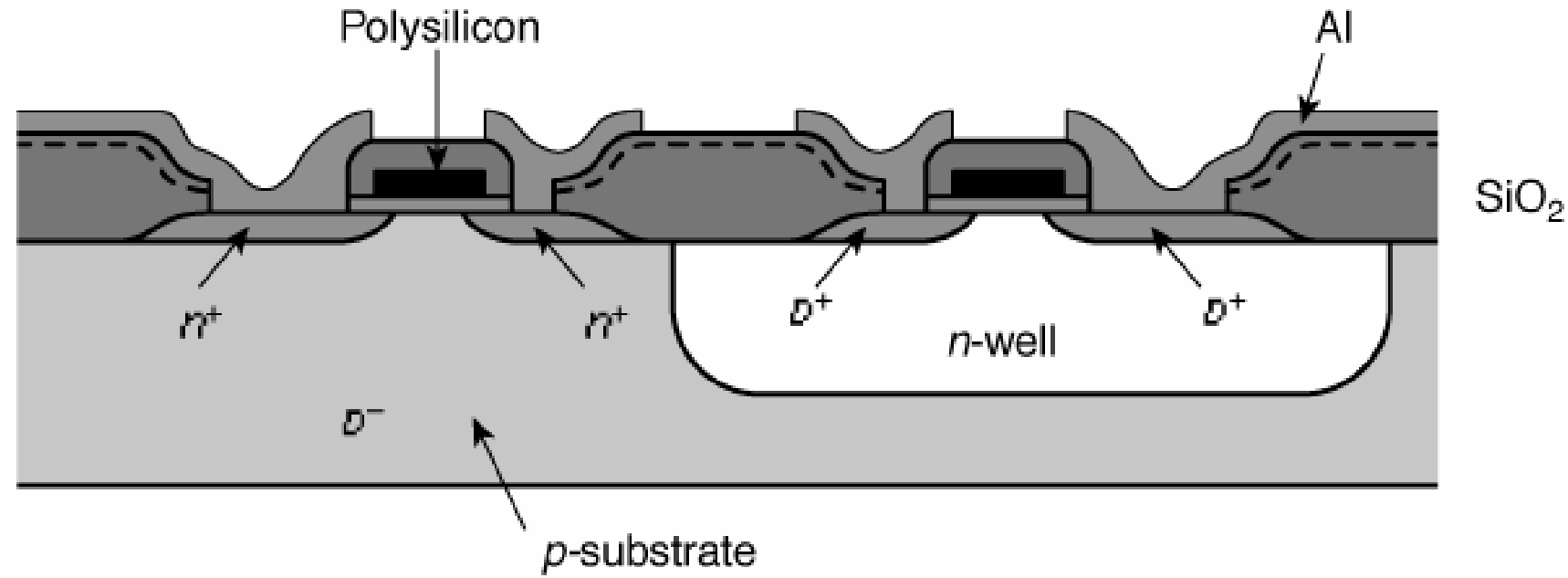
Layout Compaction



Compaction: Introduction

- Compaction tries to minimize total layout area while
  - Respecting design rules and designer-specified constraints
- Three ways to minimize the layout area
  - Reducing inter-feature space
    - Check spacing design rules
  - Reducing feature size
    - Check size rules
  - Reshaping features
    - Electrical connectivity must be preserved

Cross-Section of CMOS Technology



Jan Rabaey

Design Rules

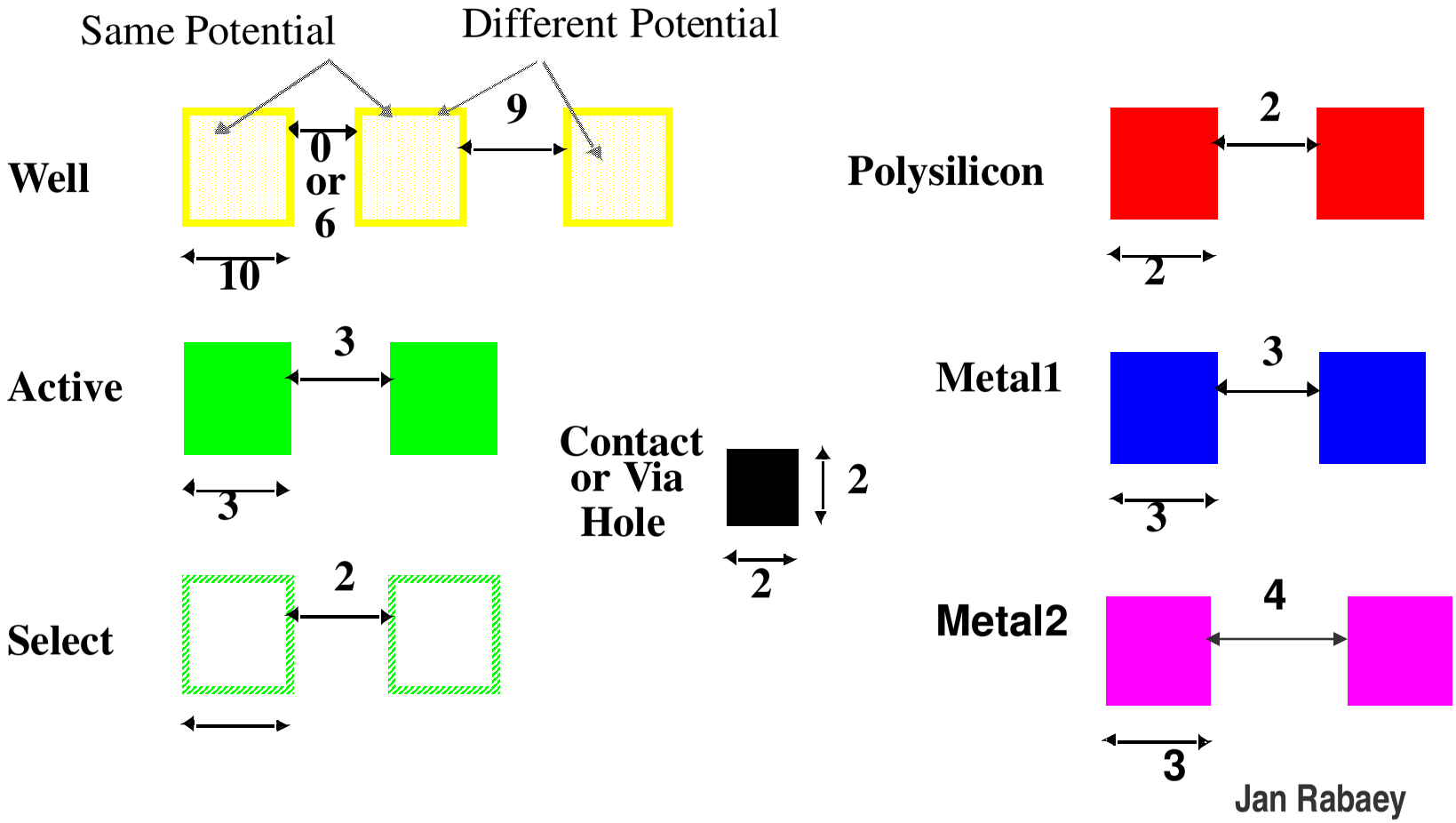
- Interface between designer and process engineer
- Guidelines for constructing process masks
- Unit dimension: Minimum line width
  - scalable design rules: lambda parameter
  - absolute dimensions (micron rules)

CMOS Process Layers

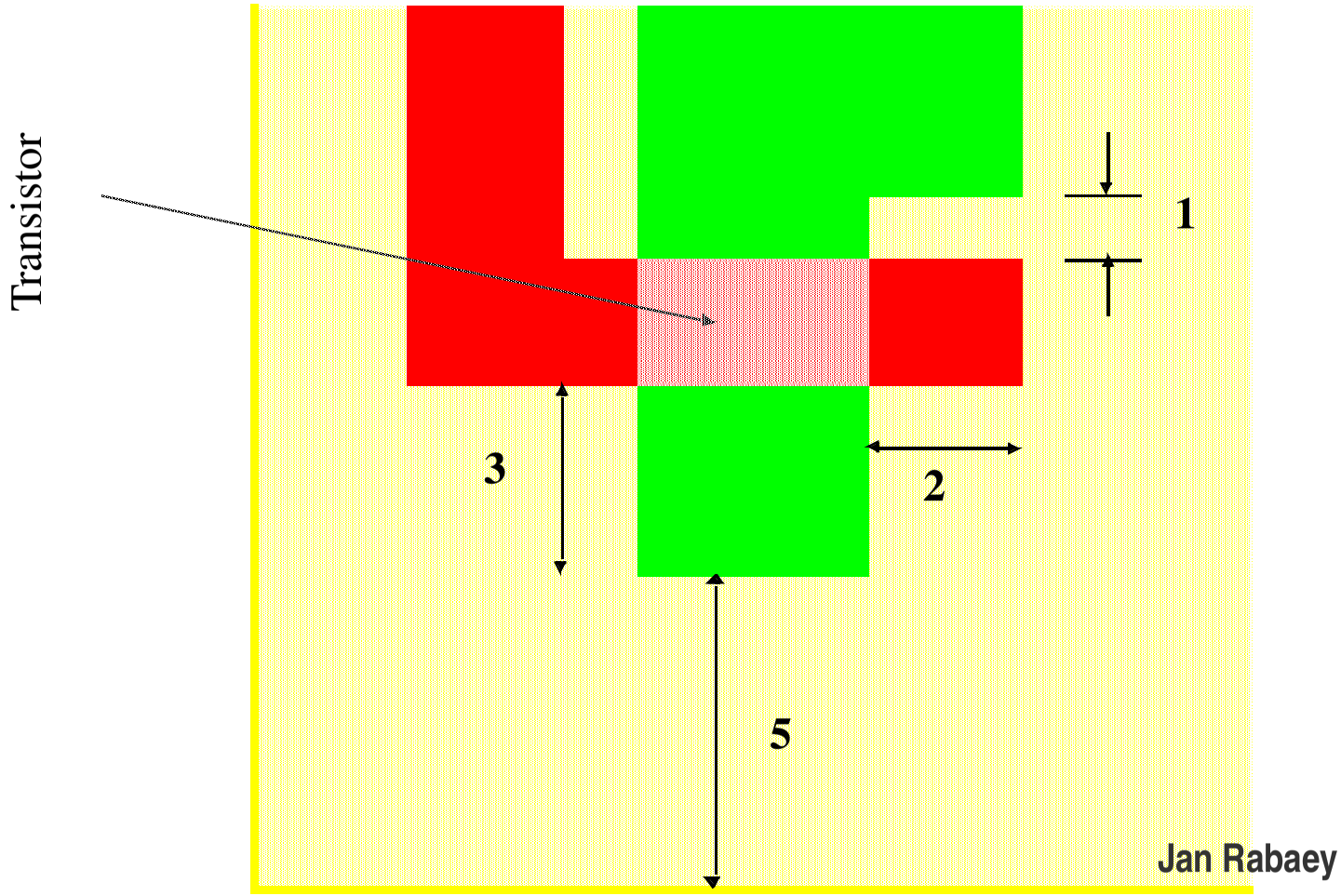
Layer	Color	Representation
Well (p,n)	Yellow	
Active Area (n+,p+)	Green	
Select (p+,n+)	Green	
Polysilicon	Red	
Metal1	Blue	
Metal2	Magenta	
Contact To Poly	Black	
Contact To Diffusion	Black	
Via	Black	

Jan Rabaey

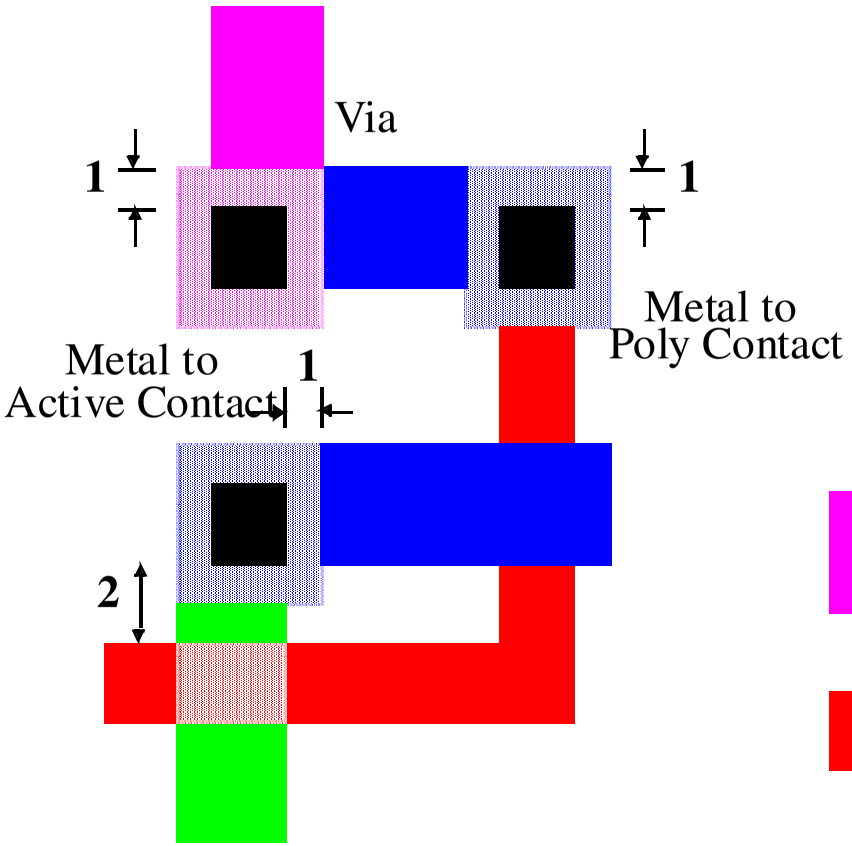
Intra-Layer Design Rules

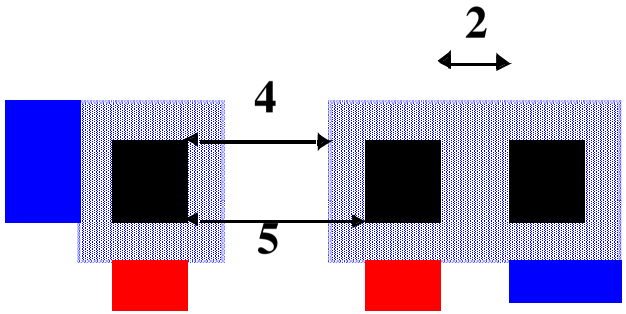


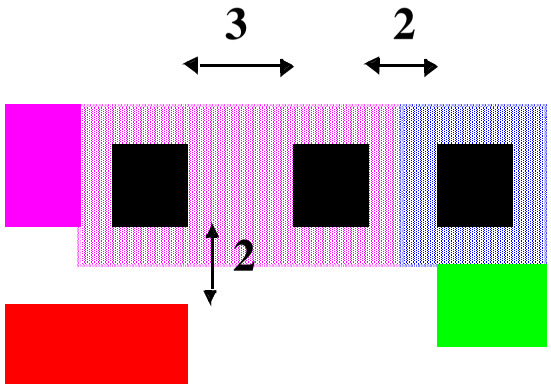
Transistor Layout – Inter Layer Rules



Via's and Contacts





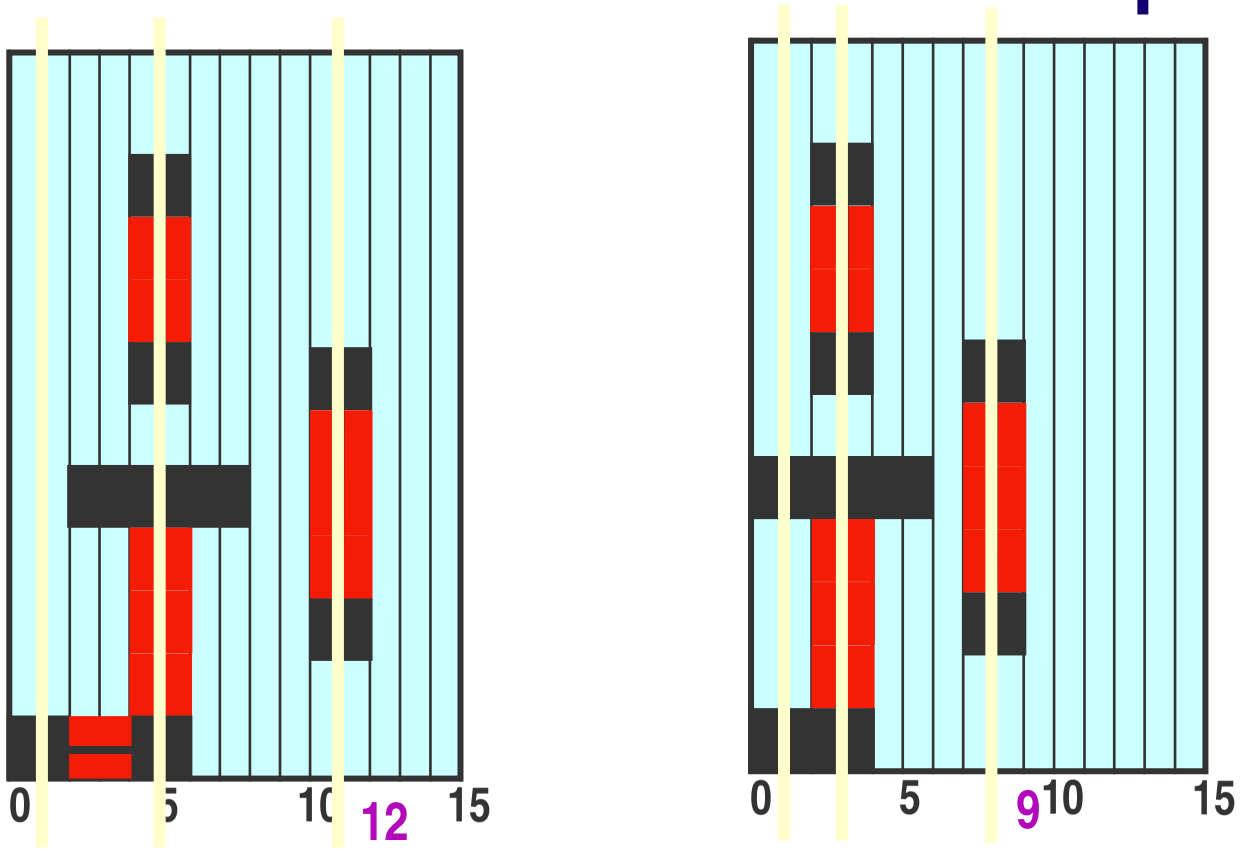


Jan Rabaey

Different Approaches to Compaction

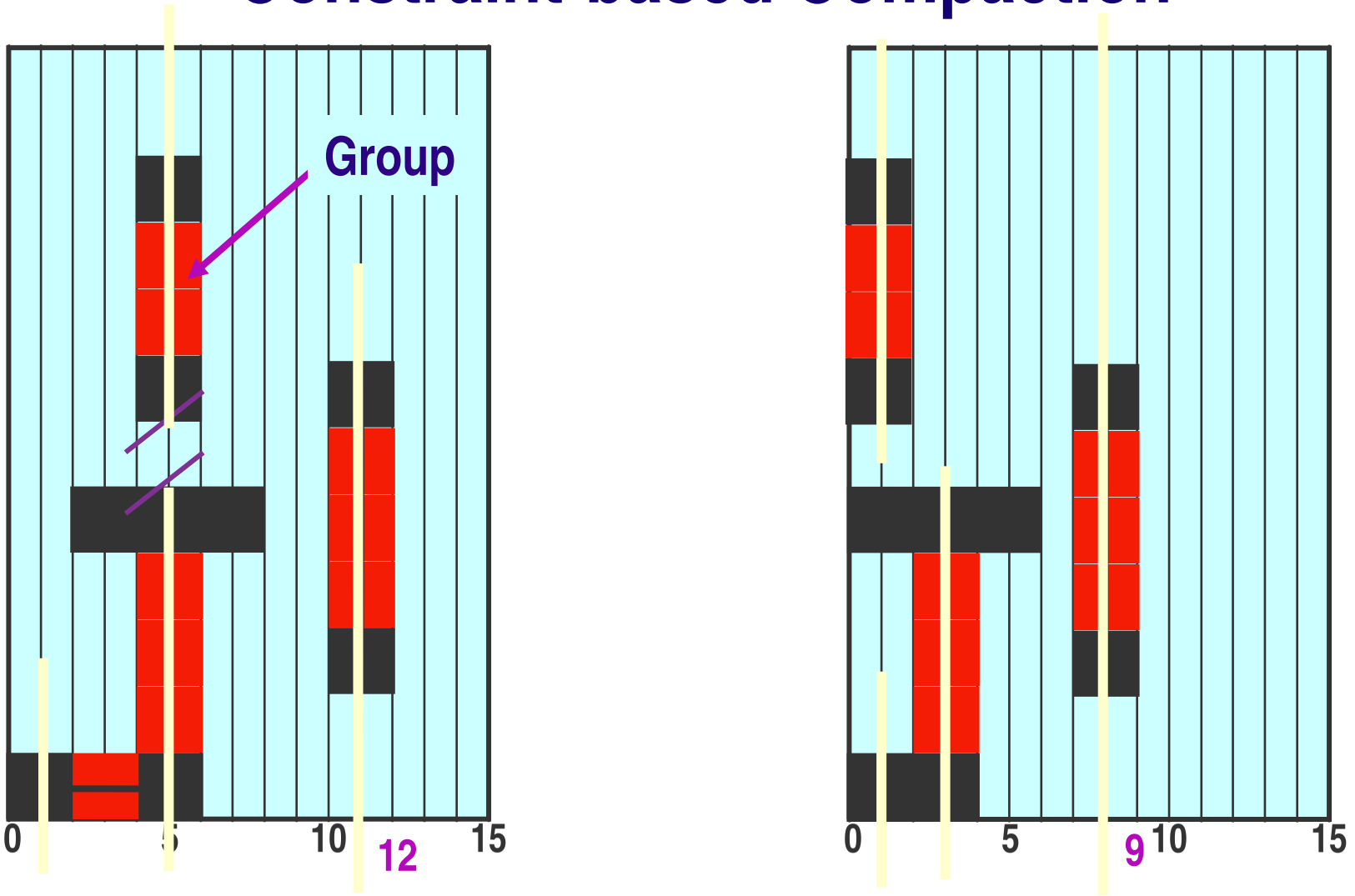
- One dimensional vs. two-dimensional compaction
- 1-D compaction
  - Components moved only in x- or y-direction
  - Efficient (nearly linear-time) algorithms available
- 2-D compaction
  - Components may be moved both in x- and y-direction
  - More effective compaction
  - NP-hard
    - Determining how x and y should interact to reduce area is hard!
- Historical interest: Constraint-graph based compaction vs. virtual grid based compaction
  - Virtual grid methods are fast and simple. Results in larger area.

Historical interest: Virtual Grid Compaction



- Every features is assumed to lie on a virtual grid line
- Required to stay at grid locations during compaction – no distortion of topology
- Compact by finding min possible spacing between each adjacent pair of grids
  - Min spacing is given by worst-case design-rule for any feature on the grid
- Advantage: algorithm is simple and fast

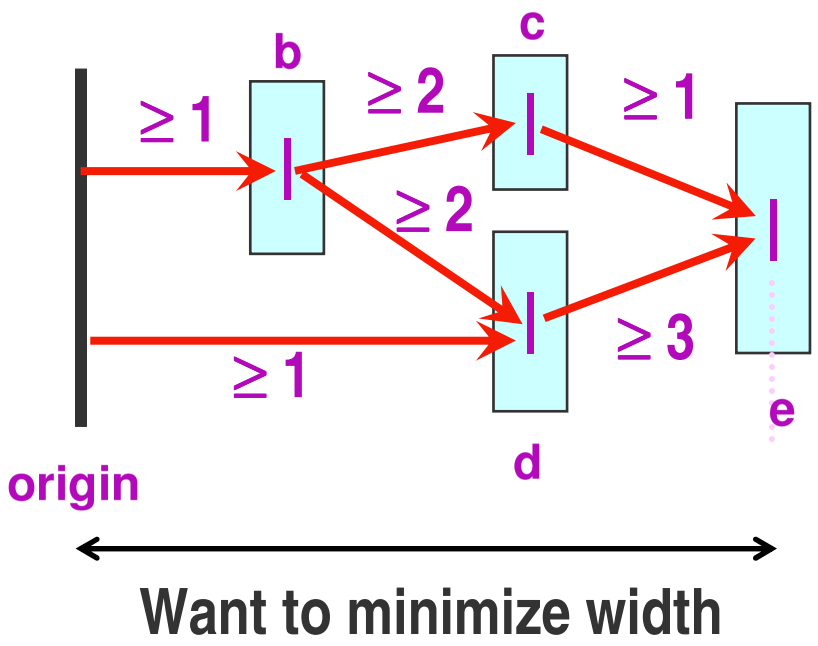
Need something different: Constraint-based Compaction



Early tools: Floss, Cabbage, SLIP, SLIM, Sticks

Constraint-based Compaction

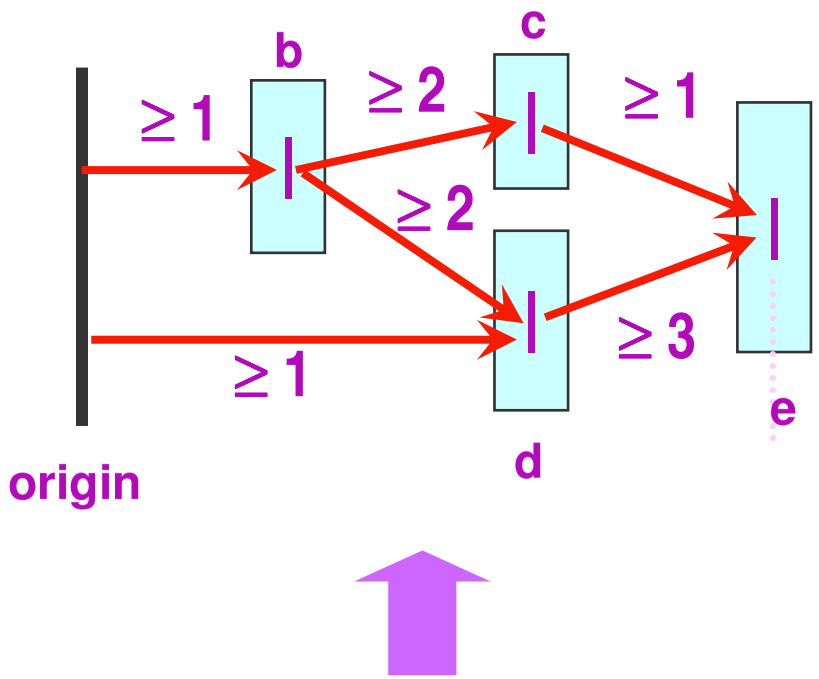
- Features represented as rectangular shapes
  - Left-most edge becomes “origin” rectangle
- Layout constraints are imposed by design rules
  - Min spacing (separation) design rules
- Want to compact to the left



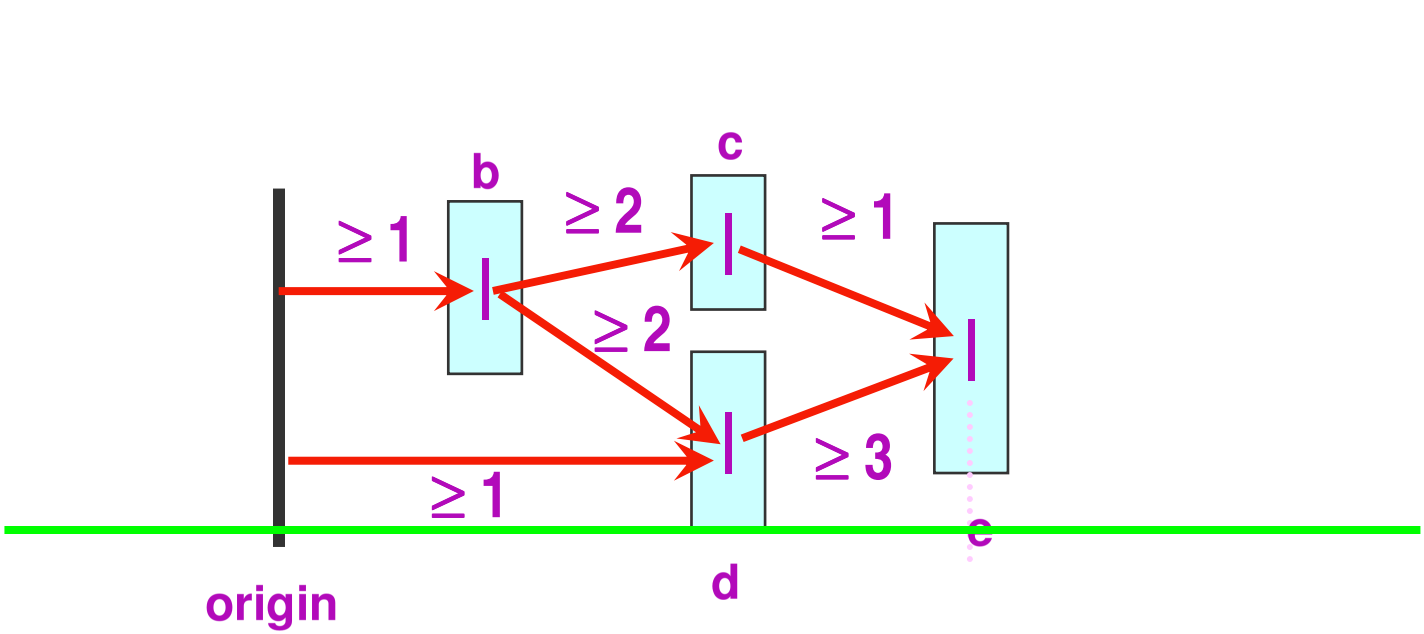
Two Steps in Constraint-based Compaction

1. Constraint generation
   - Constraints are naturally represented as inequalities
     - **Example: if *c* is to the right of *b*, and they have to be at least 2 units apart: $c \geq b + 2$**
   - These can be represented in a graph
2. Constraint solving
   - Use graph algorithms

Constraint Generation: A Scan-Line Approach

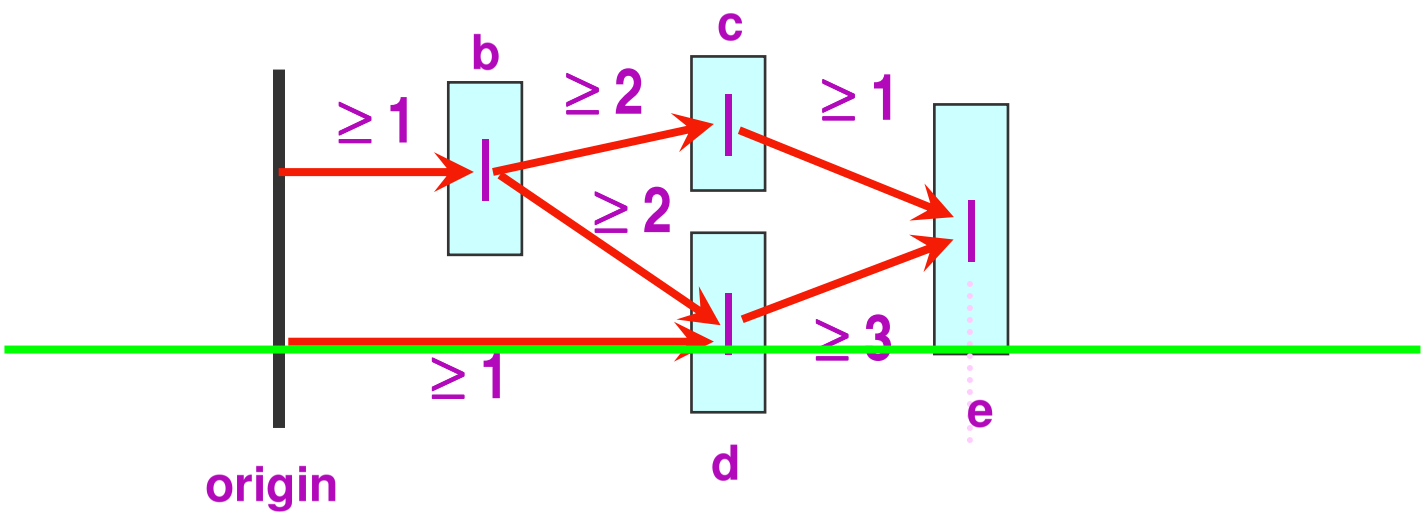


Constraint Generation: A Scan-Line Approach



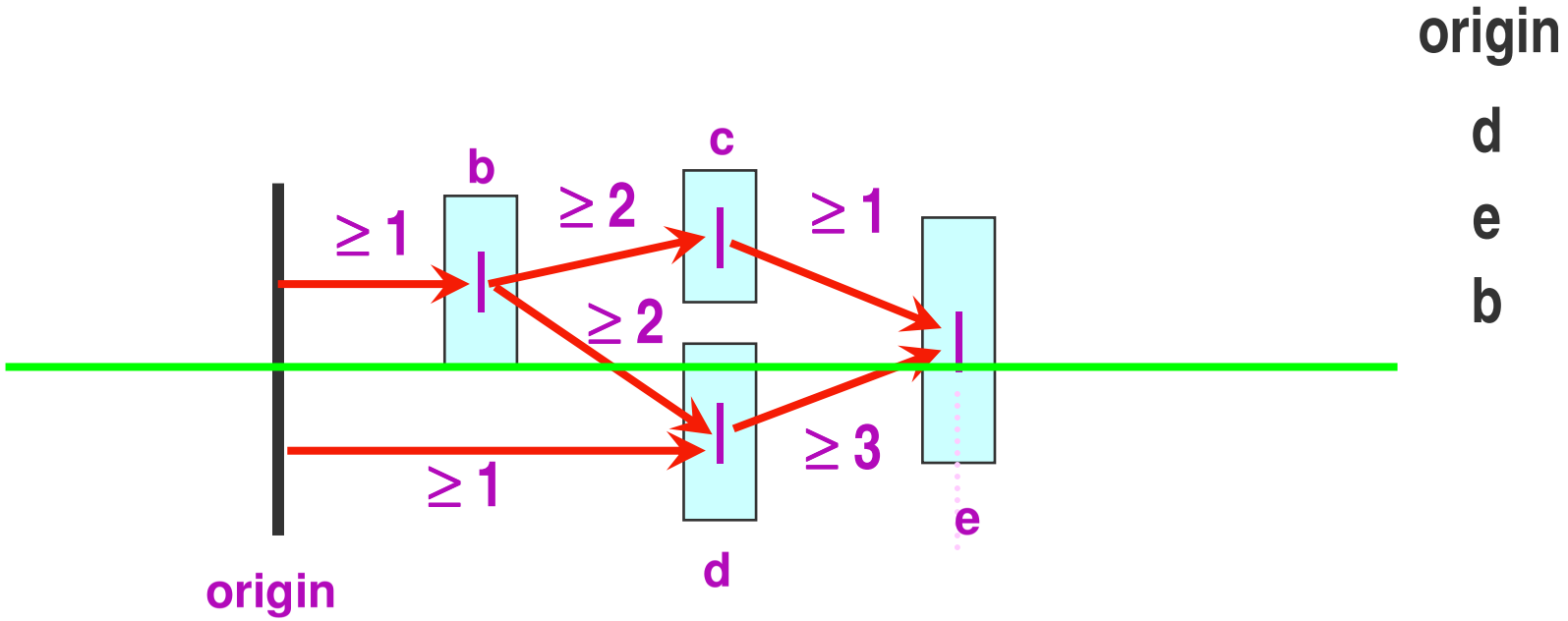
$d \geq origin + 1$

Constraint Generation: A Scan-Line Approach



$d \geq origin + 1$ $e \geq d + 3$

Constraint Generation: A Scan-Line Approach



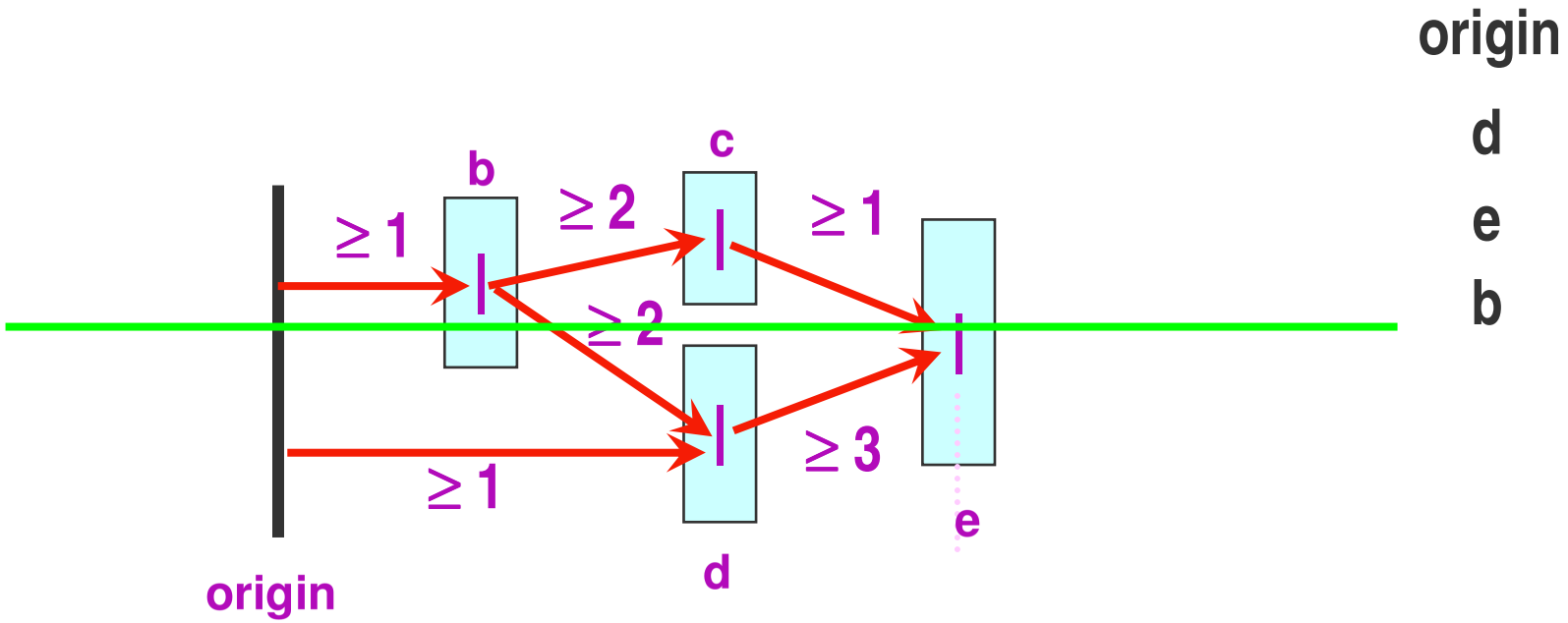
$d \geq origin + 1$

$e \geq d + 3$

$d \geq b + 2$

$b \geq origin + 1$

Constraint Generation: A Scan-Line Approach

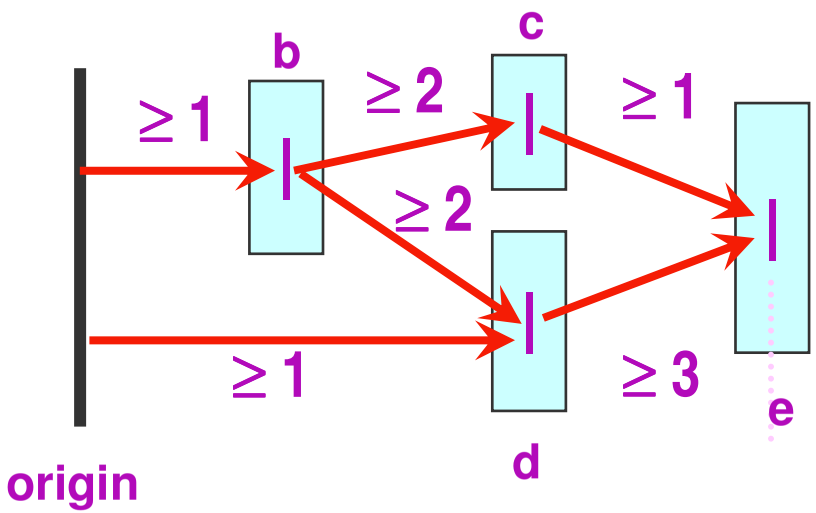


$d \geq origin + 1$ $\quad$ $e \geq d + 3$

$d \geq b + 2$ $\quad$ $b \geq origin + 1$

No more constraints involving d

All Generated Constraints



$b \geq origin + 1$ $\quad c \geq b + 2$

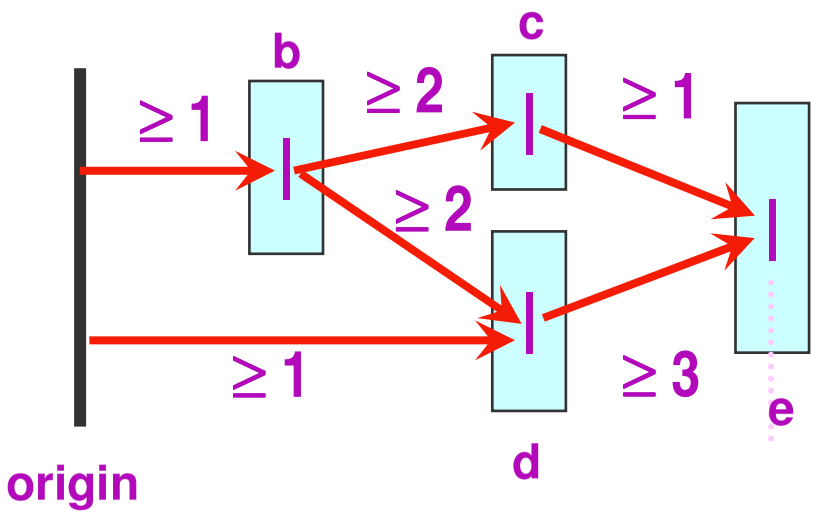
$d \geq origin + 1$ $\quad d \geq b + 2$

$e \geq c + 1$ $\quad e \geq d + 3$

Constraint Generation Summary

- Constraint generation is the most time-consuming part of constraint-based compaction
  - Getting harder in current technologies
  - In the worst case, there may be a design rule between every shape of layout
    - In reality only a small local subset of constraints are needed
  - Approached naively $O(n^2)$
- Approach:
  - First generate connectivity (grouping) constraints
  - In generating separation constraints, need to define a set of non-redundant constraints
- Scan-line algorithm
- Shadow-propagation algorithm

How do we solve these constraints?



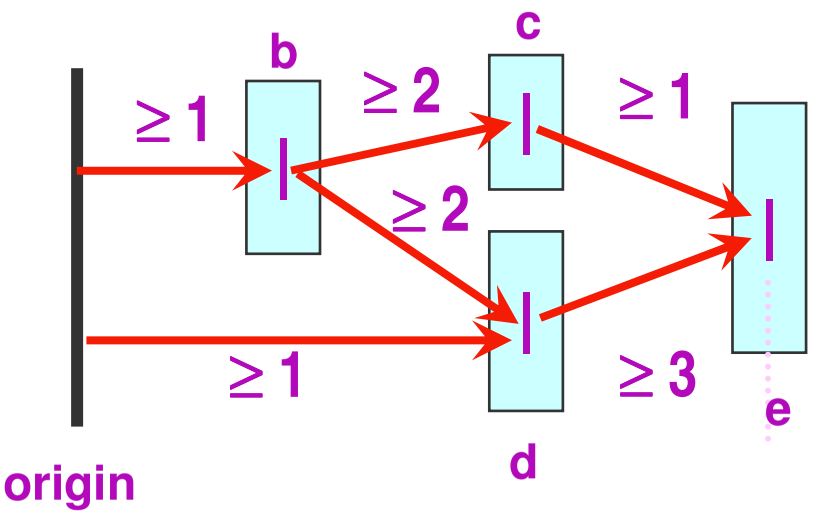
$b \geq origin + 1$ $\quad$ $c \geq b + 2$

$d \geq origin + 1$ $\quad$ $d \geq b + 2$

$e \geq c + 1$ $\quad$ $e \geq d + 3$

- For now, assume we only have separation constraints
- What is a mathematically efficient way to solve these constraints?

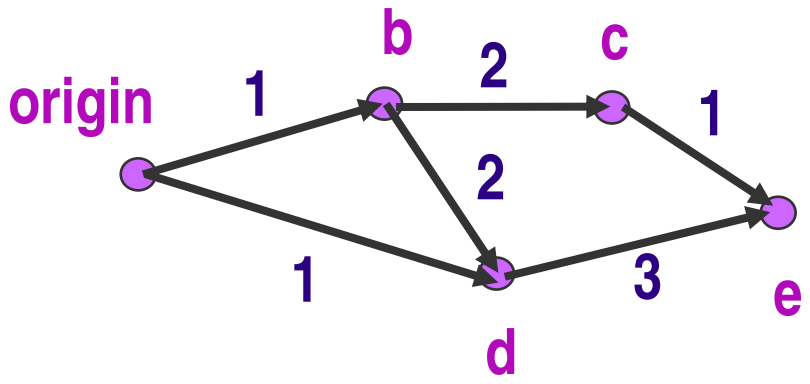
Use graphical structure



$b \geq origin + 1$ $\quad c \geq b + 2$

$d \geq origin + 1$ $\quad d \geq b + 2$

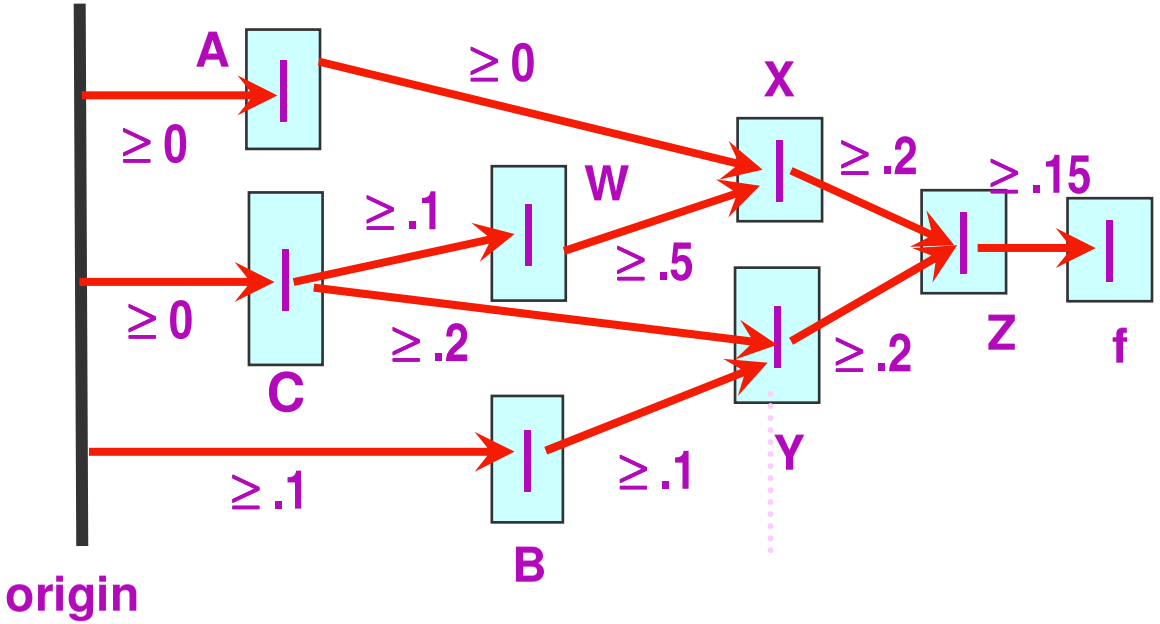
$e \geq c + 1$ $\quad e \geq d + 3$



- Constraint graph represents all constraints

Problem formulation - 1

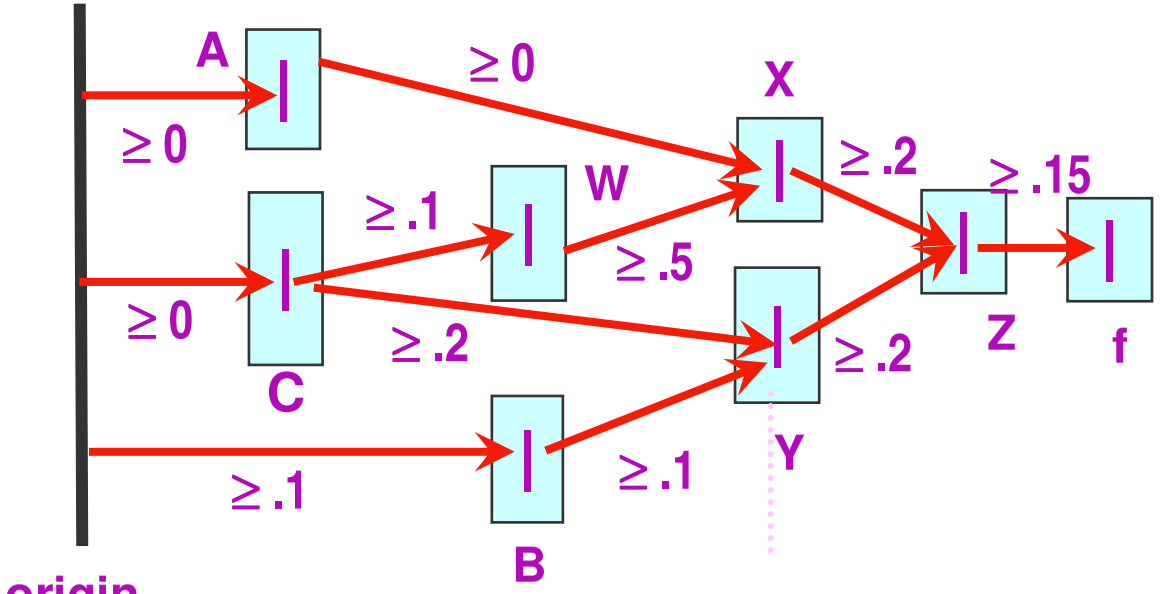
- Use a labeled *directed* graph
- *G = <V,E>*
- *Vertices* layout objects
- *Edges* represent constraints between vertices
- *Labels* represent constraint values
- Now what do we do with this?

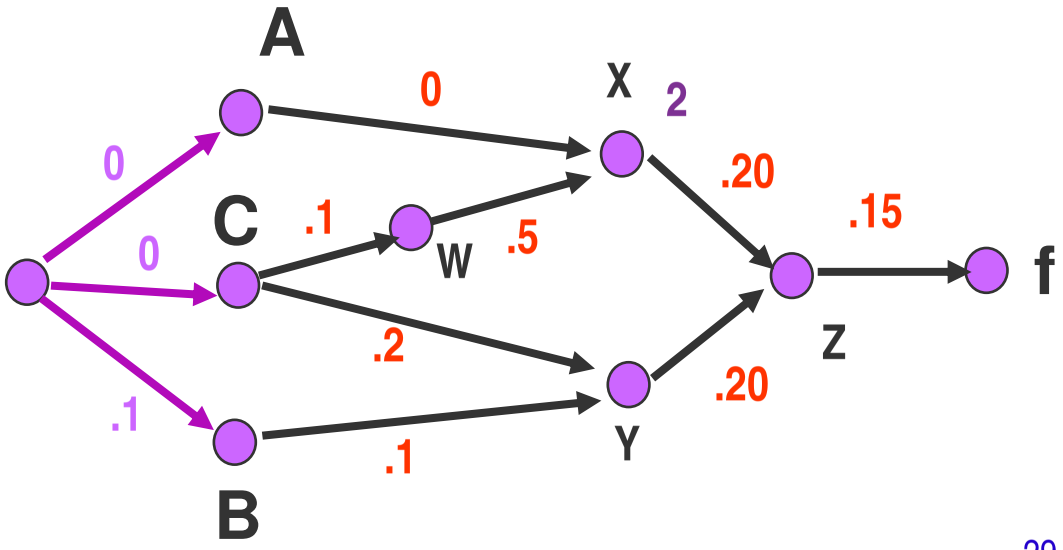


Hint: What problem does this remind you of?

Problem formulation - 1

- Use a labeled *directed* graph
- *G = <V,E>*
- *Vertices* layout objects
- *Edges* represent constraints between vertices
- *Labels* represent constraint values
- Now what do we do with this?

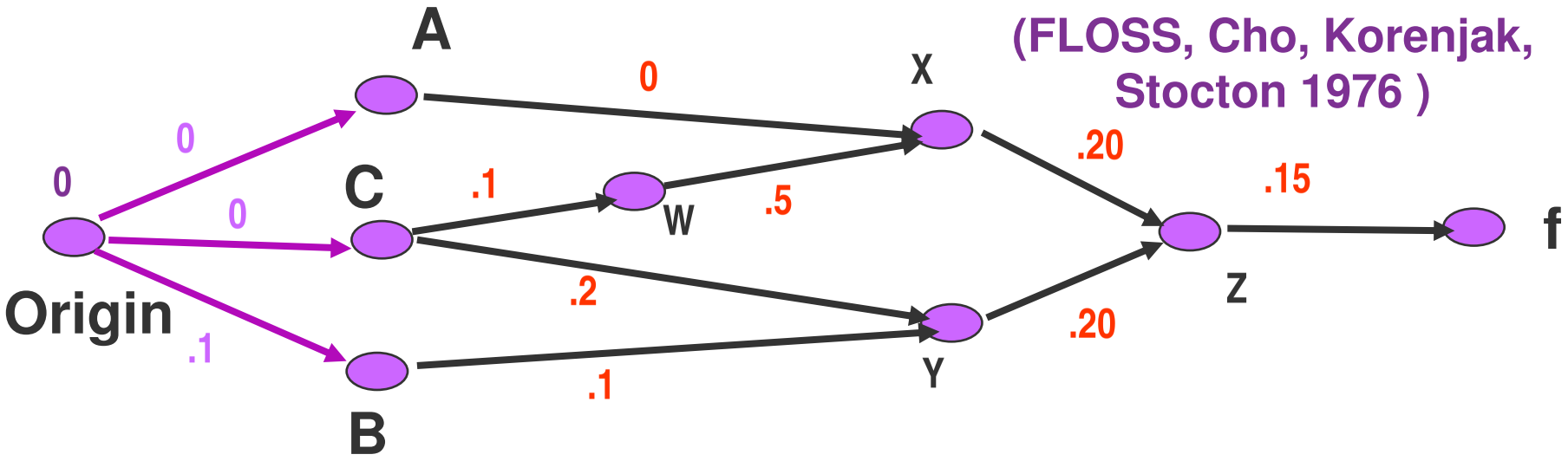




Problem formulation - 2

- Goal of 1-D compaction is to generate a minimum width layout
- Determination of minimum width is equivalent to solving a longest path problem
  - Calculate longest path from origin to each vertex
    - Longest path to a vertex is its x-coordinate
  - Longest of these is the width of the layout
- Theoretically, run time is O(|V|+|E|)
  - Why?
- Practically, run time is close to linear in |V|, the size of the layout!

Problem formulation - 3



Compute the longest path in a graph *G = <V,E,constraints,Origin>* (*constraints* is a set of labels, *Origin* is the super-source of the DAG)

Forward-prop(*W*){
  for each vertex *v* in *W*
    for each edge *<v,w>* from *v*
      value(w) = max(value(w), value(v) + value(w) + constraint(<v,w>))
      if all incoming edges of *w* have been traversed
        add *w* to *W*
}

Longest path(G)
  Forward_prop(Origin)
}

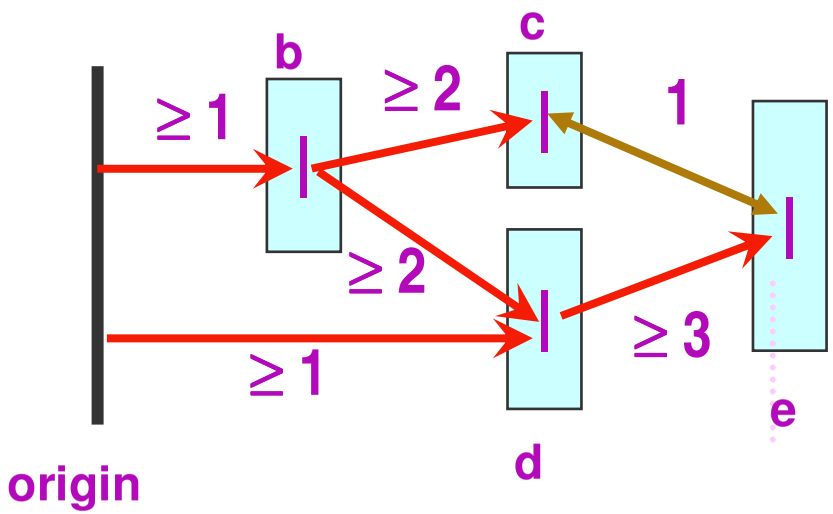
Are we done?

DAG → Arbitrary Graph

- Separation constraints yield inequalities, all in the same direction
  - Corresponding graph is a DAG
- What about grouping constraints?
  - These are typically equalities
- Sliding ports
  - Yield inequalities in both directions (upper and lower bounds)
- O(|V| + |E|) algorithm only works on DAGs, but the constraint graph can contain cycles

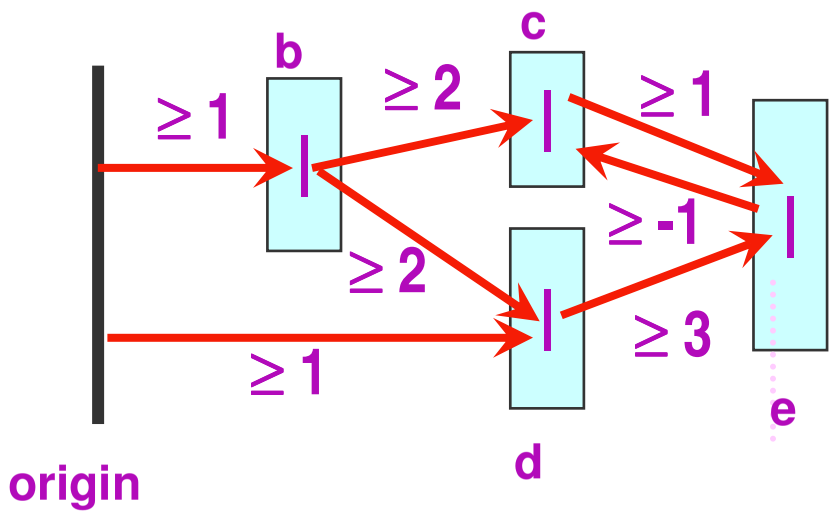
First Elaboration - Equality Constraints -1

- *Grouping constraints*
  - Features from same circuit component
  - Need to be moved together
- Described by *equality constraints*



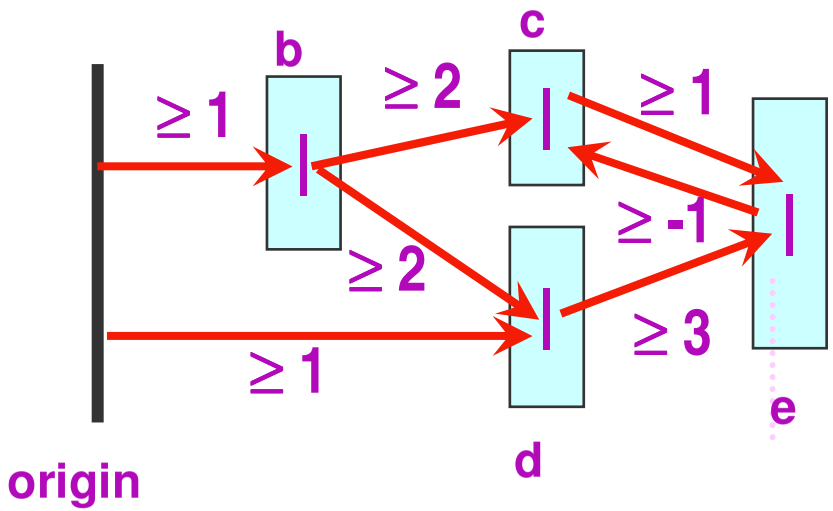
e = c + 1

First Elaboration - Equality Constraints -2



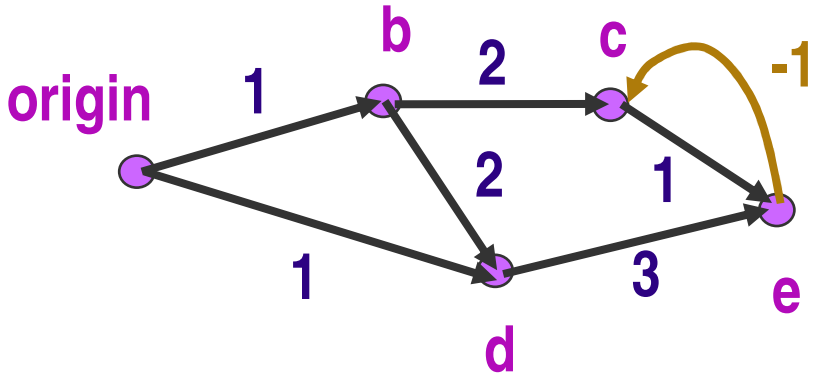
$e = c + 1$ == $e \geq c + 1$, $c \geq e - 1$

Reflecting Equality Constraints



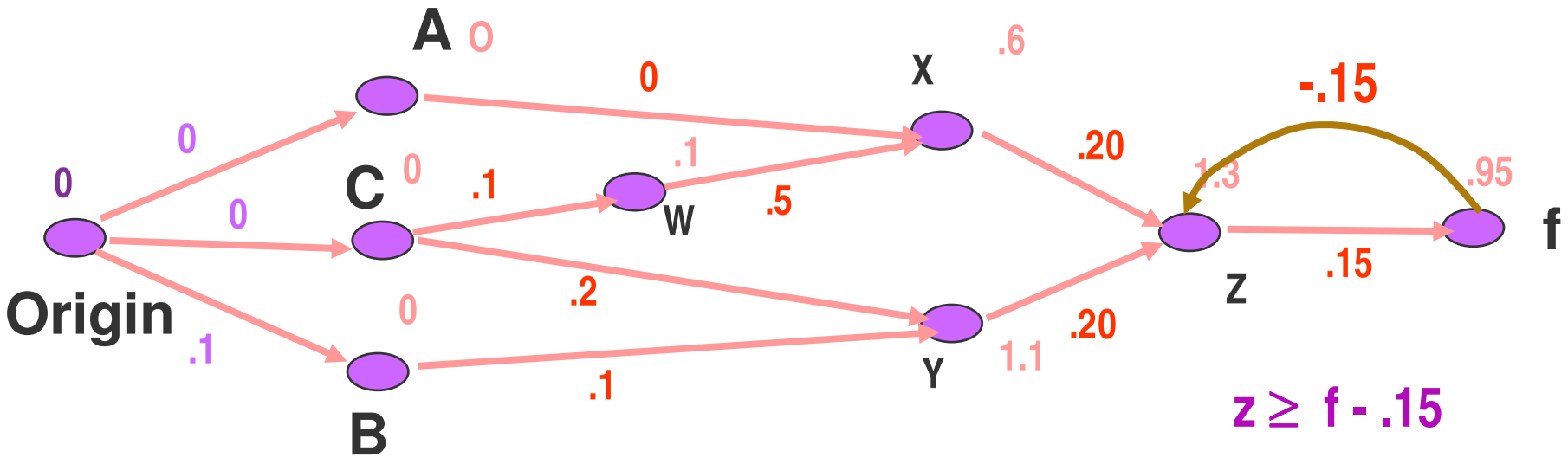
$e \geq c + 1$

$c \geq e - 1$



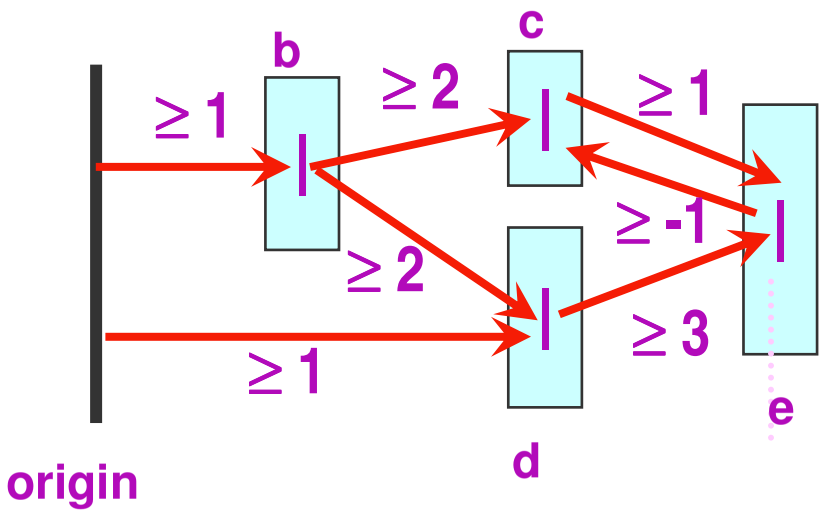
What challenges does this pose to our algorithm?

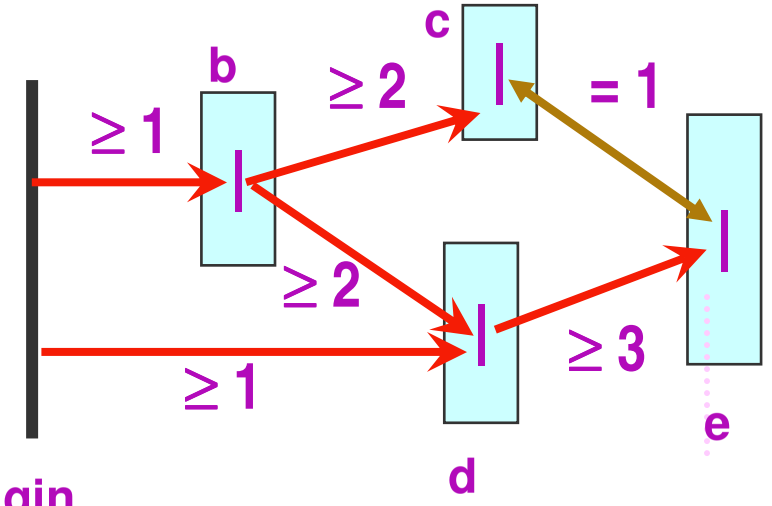
Dealing With Legitimate Cycles



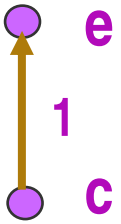
- Topological longest path algorithm can't handle graphs with cycles, only DAGs
- Can use a Bellman / Moore algorithm for general graphs
  - Run time is O(|V|*|E|)
  - Run time of longest path algorithm on DAG is O(|V|+|E|)

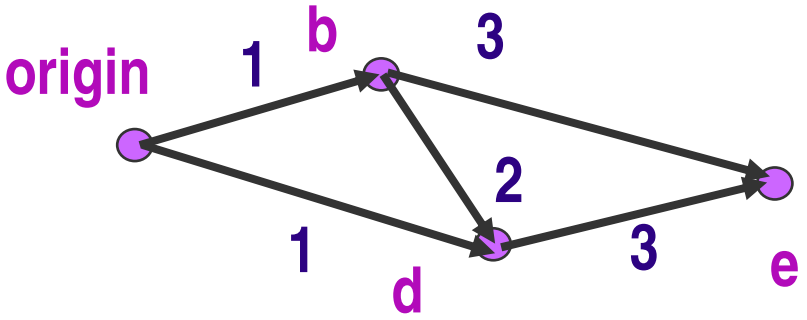
Alternative Approach to Equality Constraints





e = c + 1



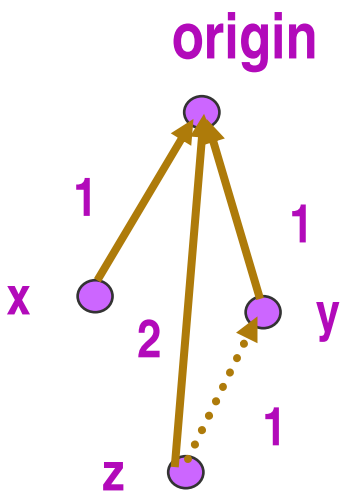


Handle equality constraints independently

Build independent graph for equality constraints

Express inequality constraints between *designated representatives*

Handling Equality Constraints

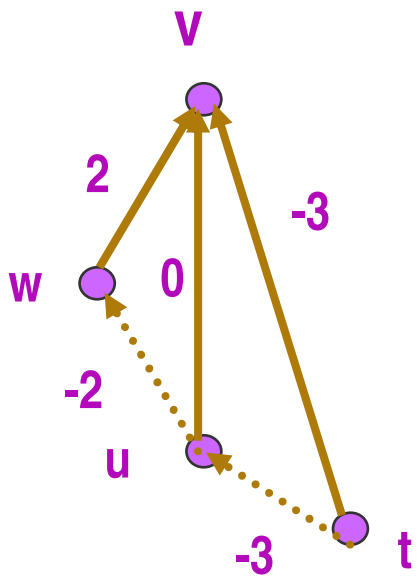


origin = x + 1

origin = y + 1

z = y + 1

origin = z + 2



v = w + 2

w = u – 2 i.e. u = w + 2

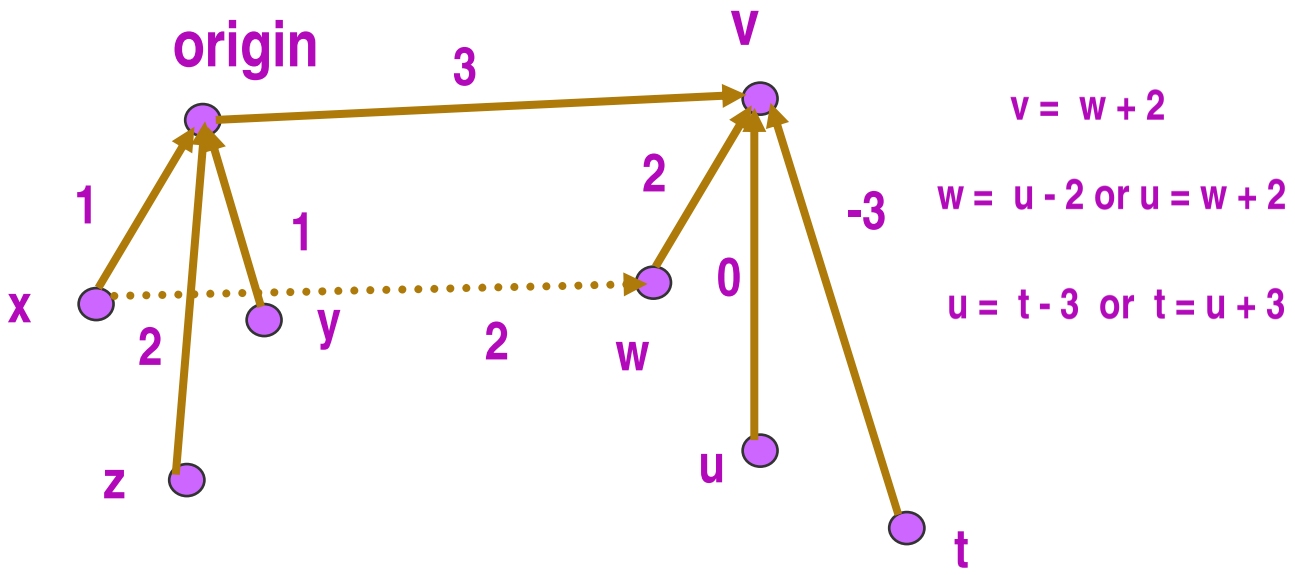
u = t - 3 i.e. or t = u + 3

Handling Equality Constraints

origin = x + 1

origin = y + 1

y = z + 1



v = w + 2

w = u - 2 or u = w + 2

u = t - 3 or t = u + 3

Generally, this is known as the union-find algorithm

$$w = x + 2$$

$$w\,(v - 2) = x\,(origin - 1) + 2$$

$$v - 2 = origin - 1 + 2$$

$$v = origin + 3$$

Constraint-Based Compaction: Constraint Solving Summary

Build constraint graph

- if equality constraint
  - add to equality constraint graph
- if inequality constraint
  - find *distinguished representatives* of each vertex in constraint
  - add constraint between *distinguished representatives*

Check for cycles in inequality constraint graph

- If cycles exist terminate with error

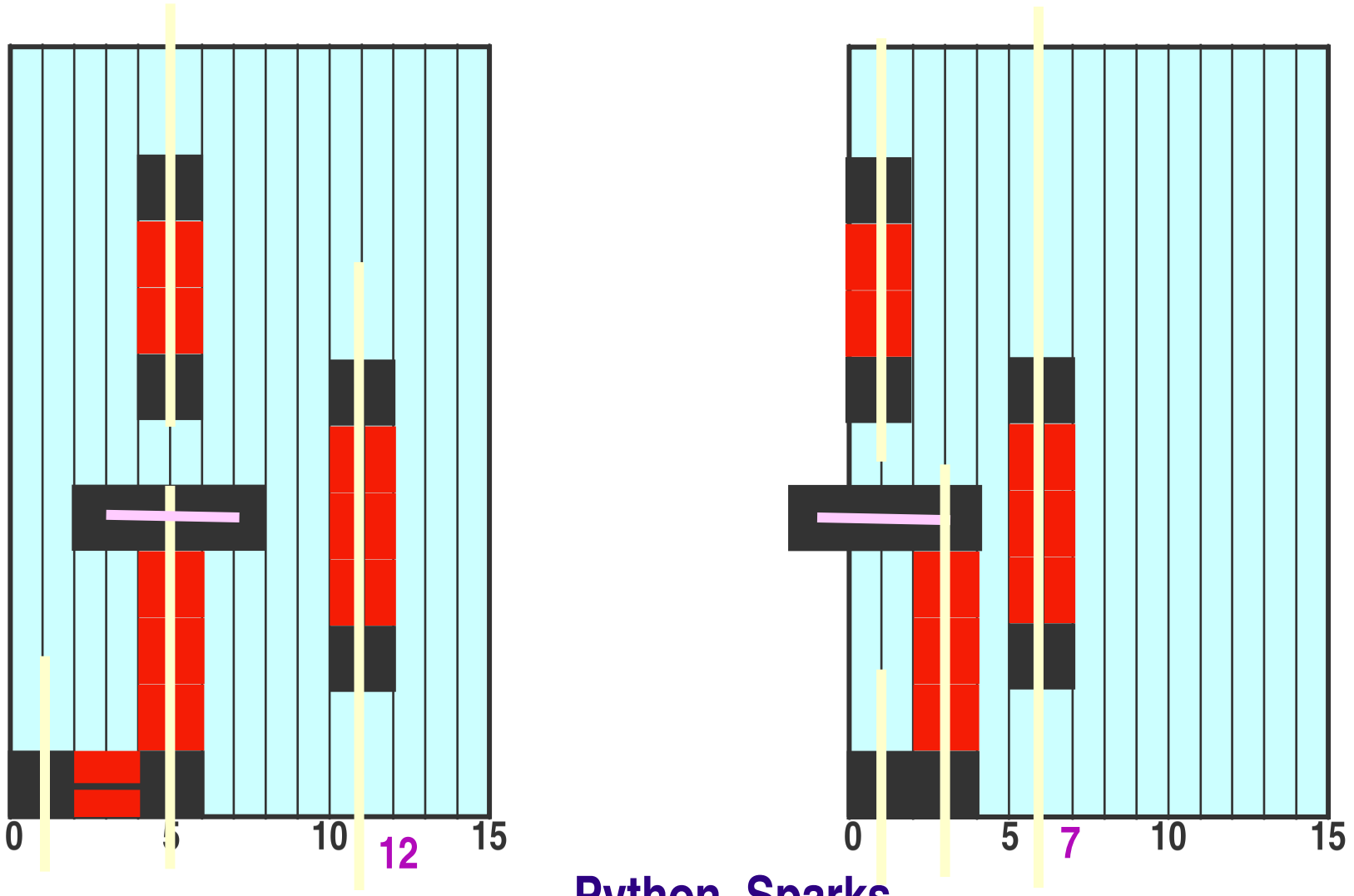
Solve inequality constraint graph

Solve equality constraint graph

Compaction Enhancements

- 1-D constraint-based compaction problem can be formulated optimally and computationally efficiently
- In real circuits what we want is often more complex than can be captured in simple linear inequalities of the form:
  - $e \geq c + 1$
- Or equalities of the form:
  - $u = t - 3$
- For example:
  - Spacing, slack distribution
  - Wirelength minimization
  - Jog introduction
- Improving area minimization using:
  - 1 ½ D compaction
  - 2D compaction

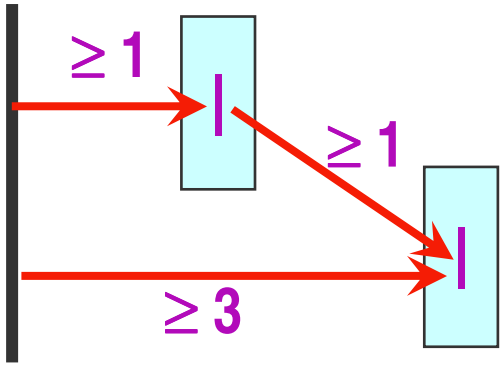
Enhancements: Sliding/Spacing Terminals

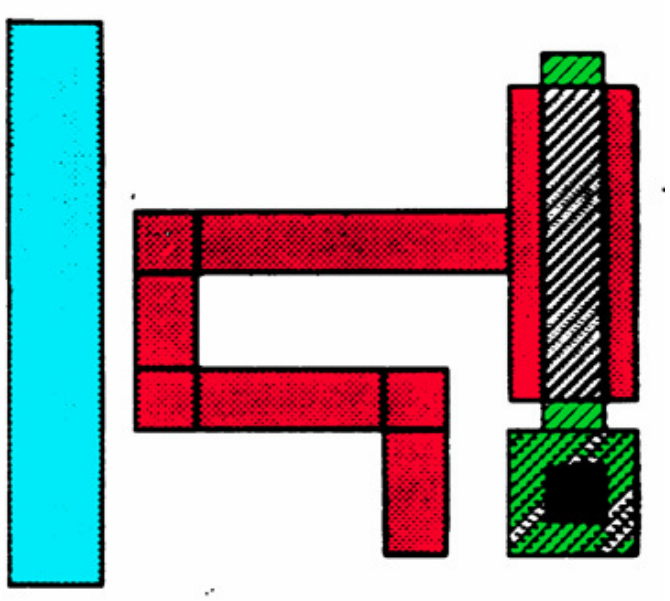


Python, Sparks

(requires use of upper as well as lower-bound constraints)

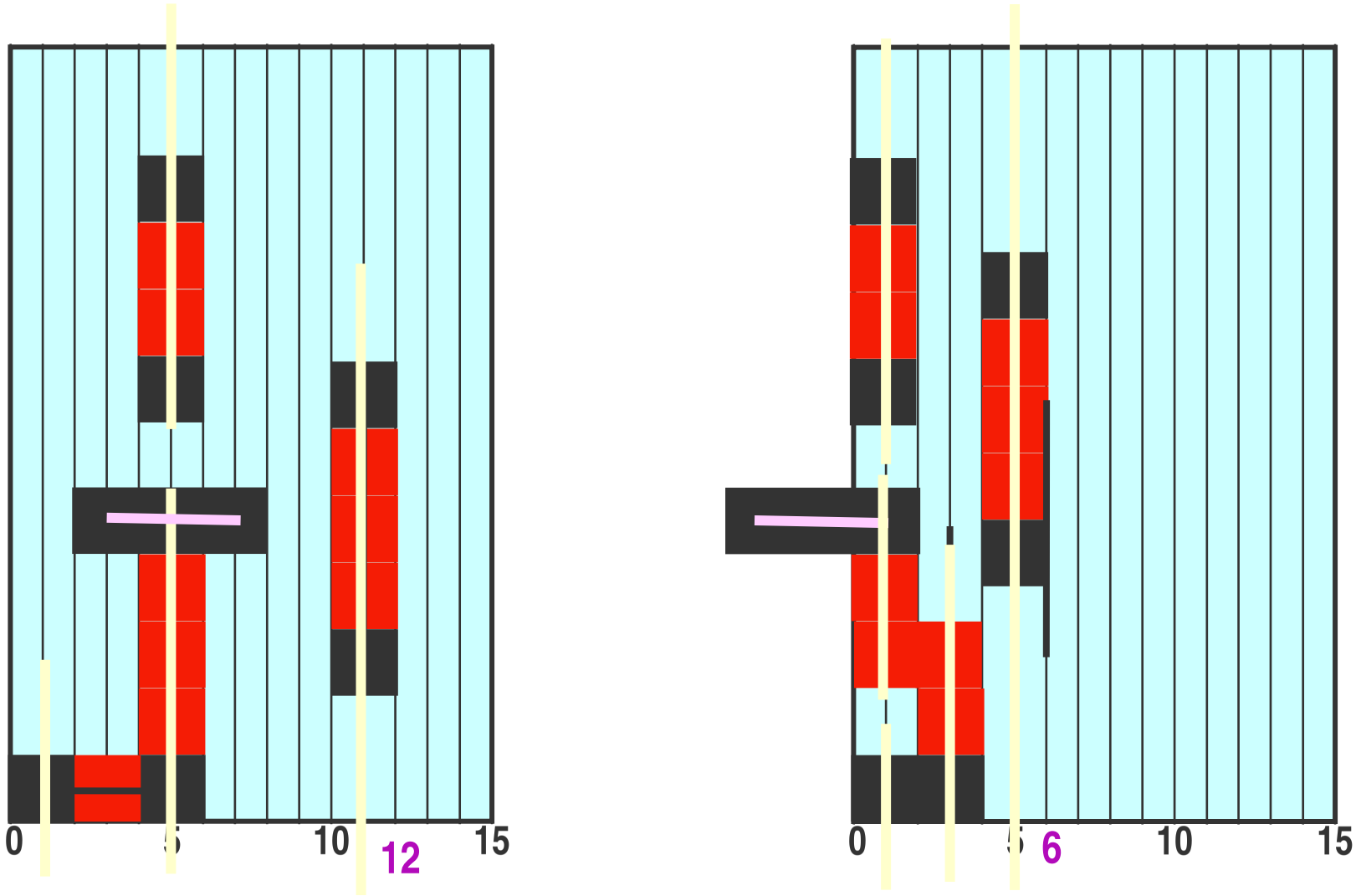
Wire-Length Minimization





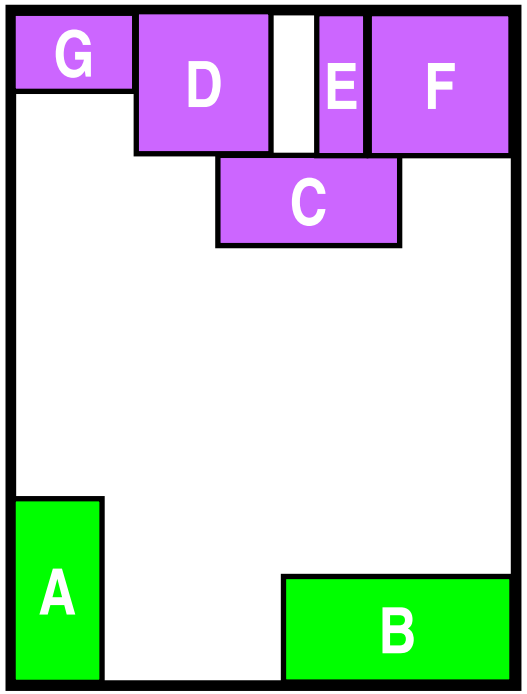
- Features not on the critical path will be pulled towards a layout edge because they are given their minimal legal spacing
- May lead to increased wire length and slower circuit performance
- Can re-distribute ‘slack’ (the available empty space) to the features not on the critical path

Jogs in “Wires”

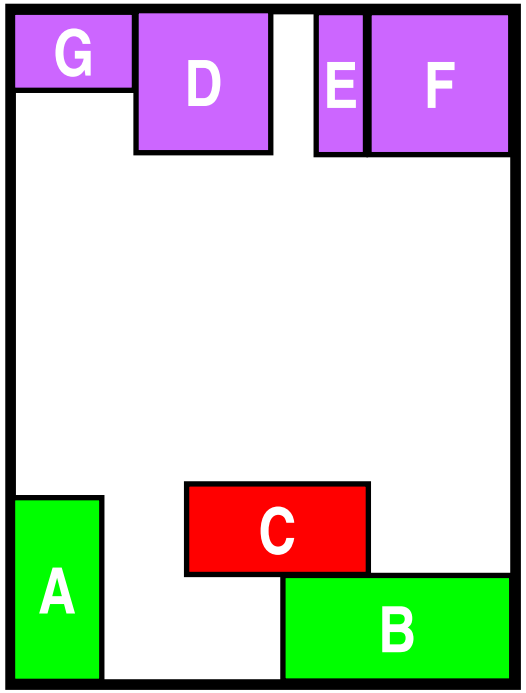


Cabbage, SLIP, Dumbo

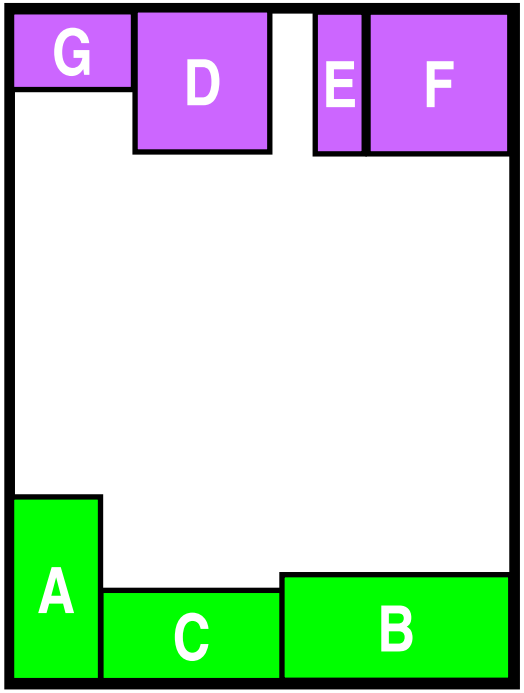
One-and-a-half Dimensional Compaction



C compacted up



C compacted down

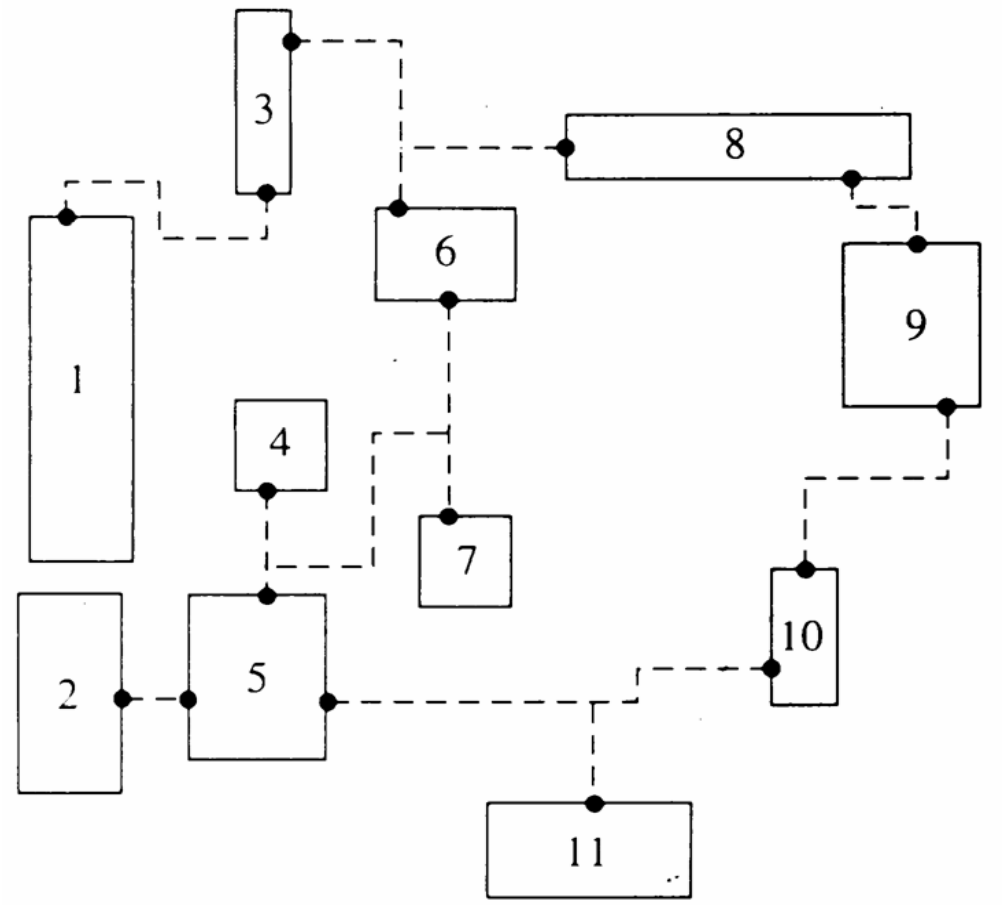


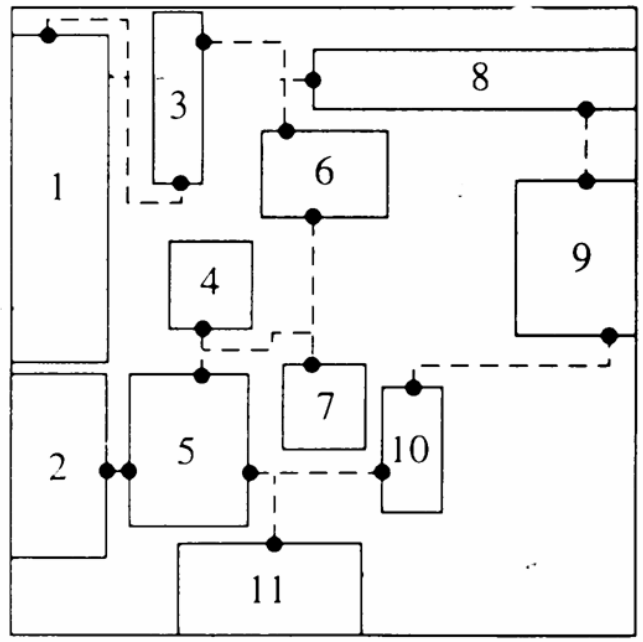
If we could just bump C over

One-and-a-half Dimensional Compaction

- Key idea: provide enough lateral movements to blocks during compaction to resolve interferences
- Algorithm starts with a partially compacted layout (two applications of 1-D compaction)
- Maintain two lists – floor and ceiling
- Floor is a list of blocks visible from the top, ceiling is the list of blocks visible from the bottom
- Select the lowest block in the ceiling and move it to the floor *maximizing the gap between floor and ceiling.*
- Continue until all blocks have been moved from ceiling to floor.

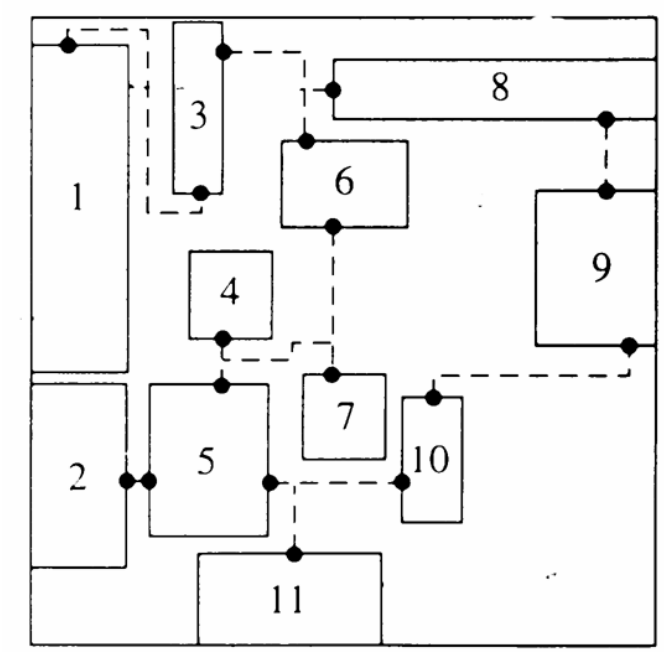
1-Dimensional Compaction in 2D



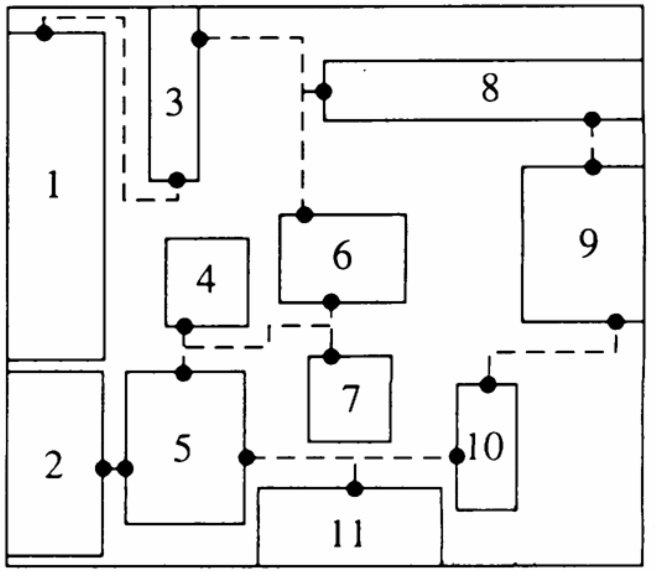


X then Y 1D Compaction

1-Dimensional Compaction in 2D

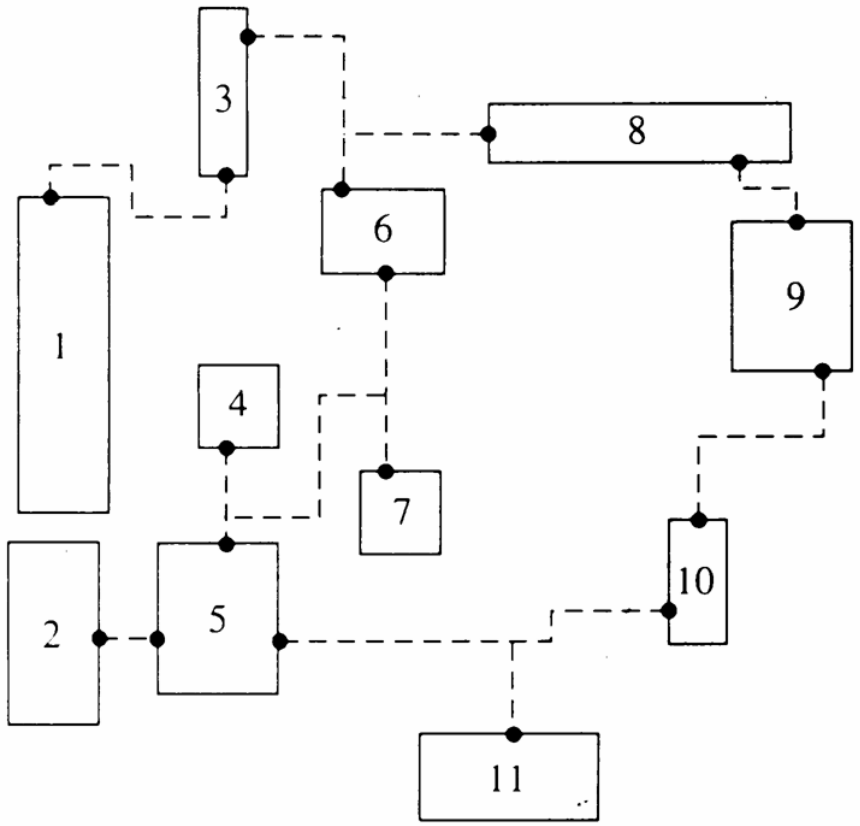


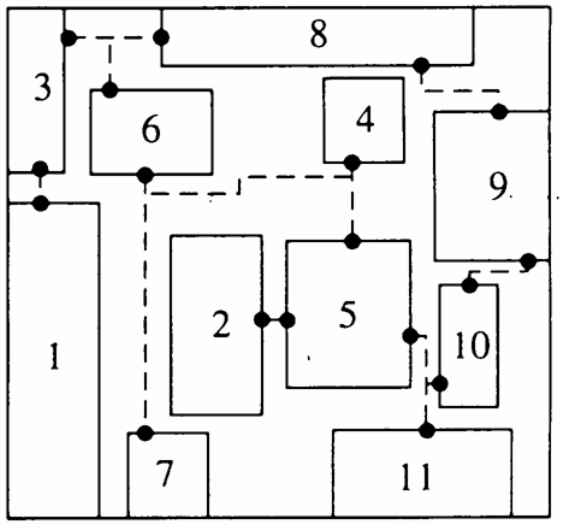
X then Y 1D Compaction



Y then X 1D Compaction

True Two-Dimensional Compaction

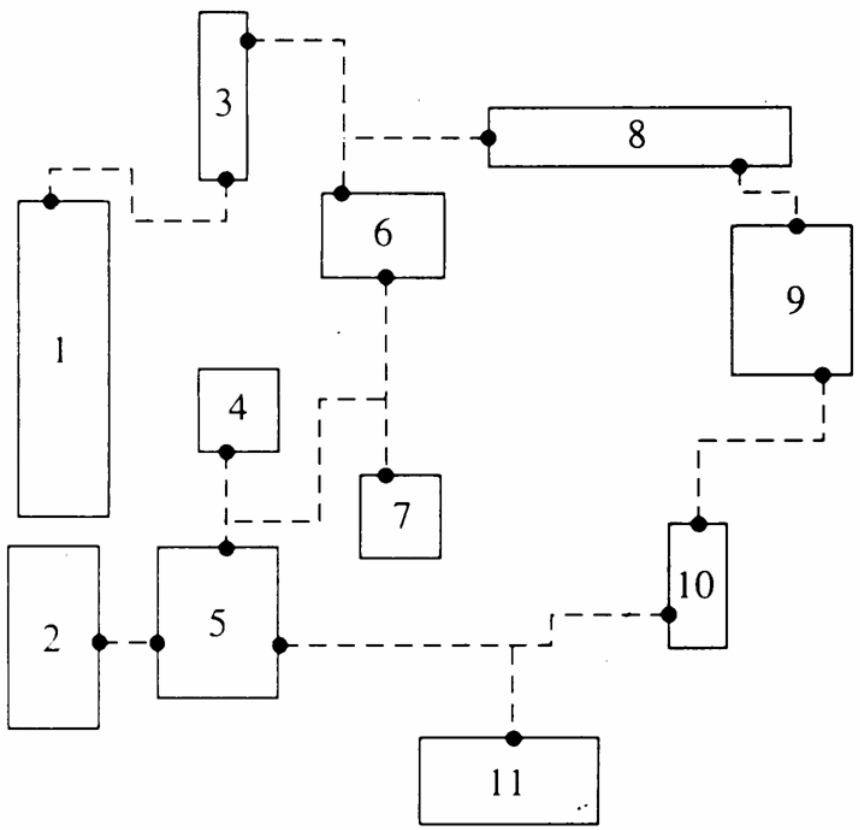


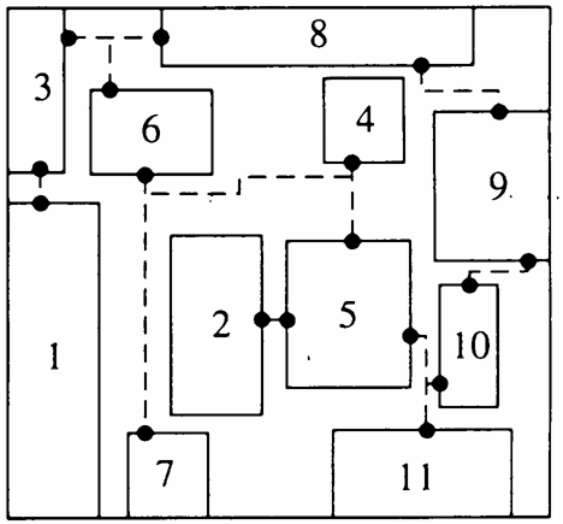


2D Compaction

- Two dimensional compaction is NP Hard (C. K. Wong, 1984)
- Choosing how two dimensions should interact to produce optimal is hard
- Can formulate as integer-linear programming problem
  - Worst-case complexity is exponential

What makes 2-D compaction hard?

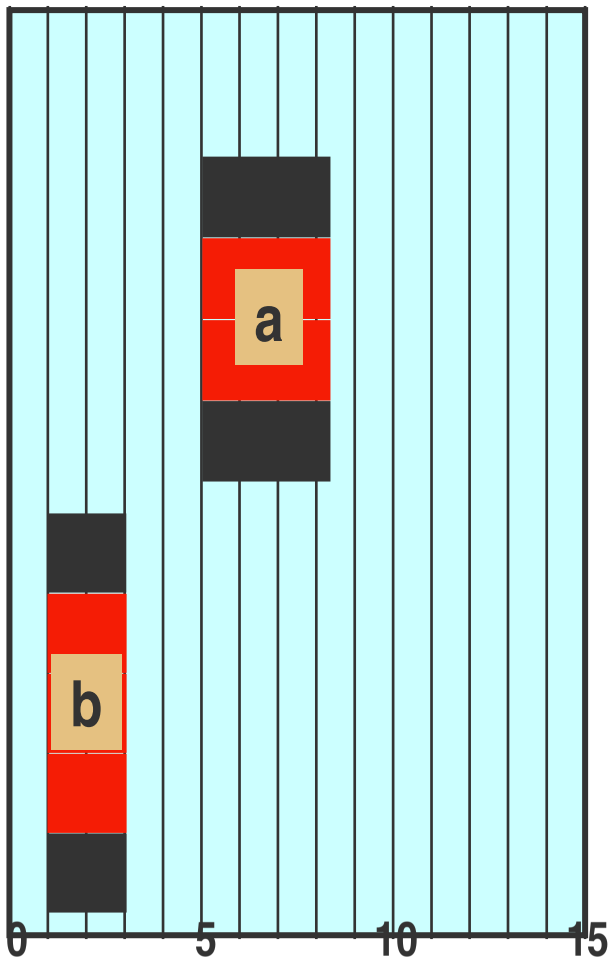




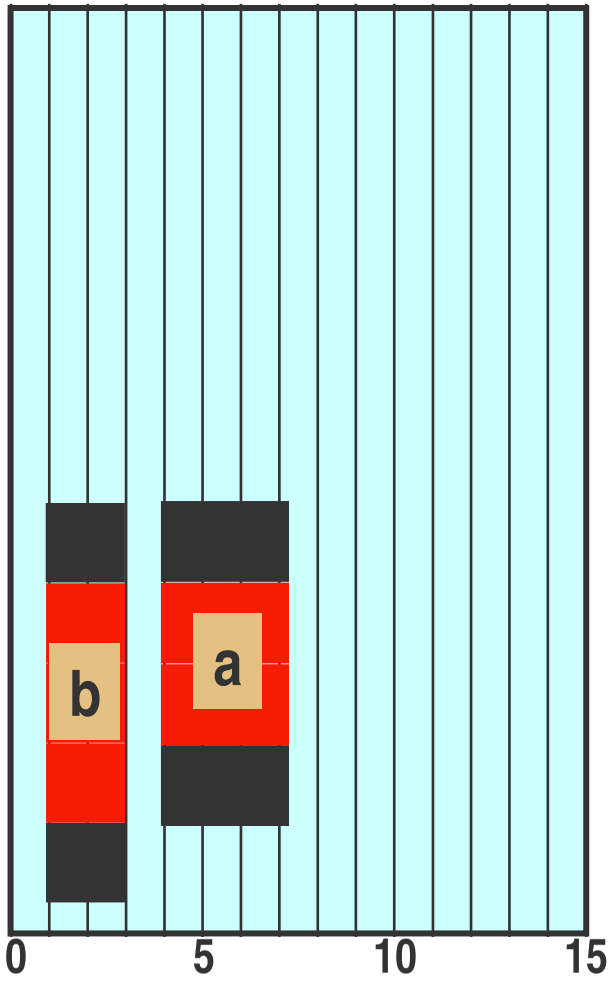
2D Compaction

- Two dimensional compaction is NP Hard (C. K. Wong, 1984)
- Choosing how two dimensions should interact to produce optimal is hard
- Can formulate as integer-linear programming problem
  - Worst-case complexity is exponential

Choices -- Vertical or Horizontal Overlap?

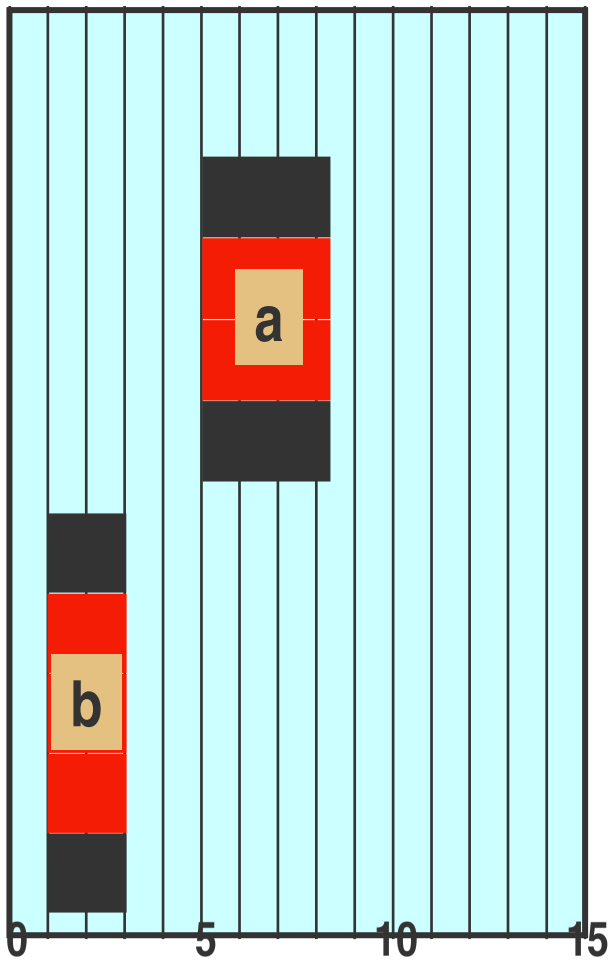


$a \geq b + 1$

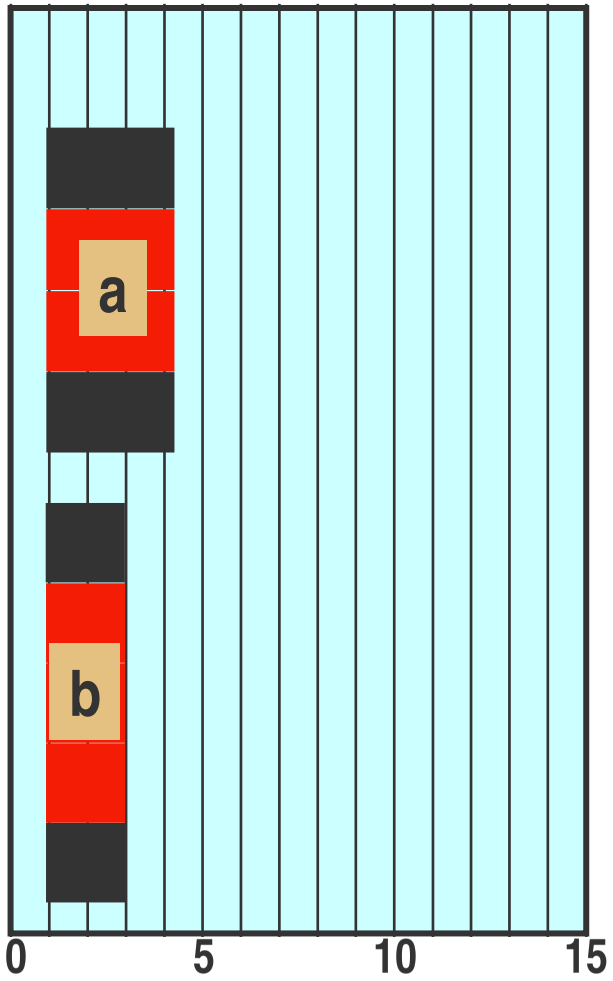


$a \geq b + 1$

Choices -- Vertical or Horizontal Overlap?

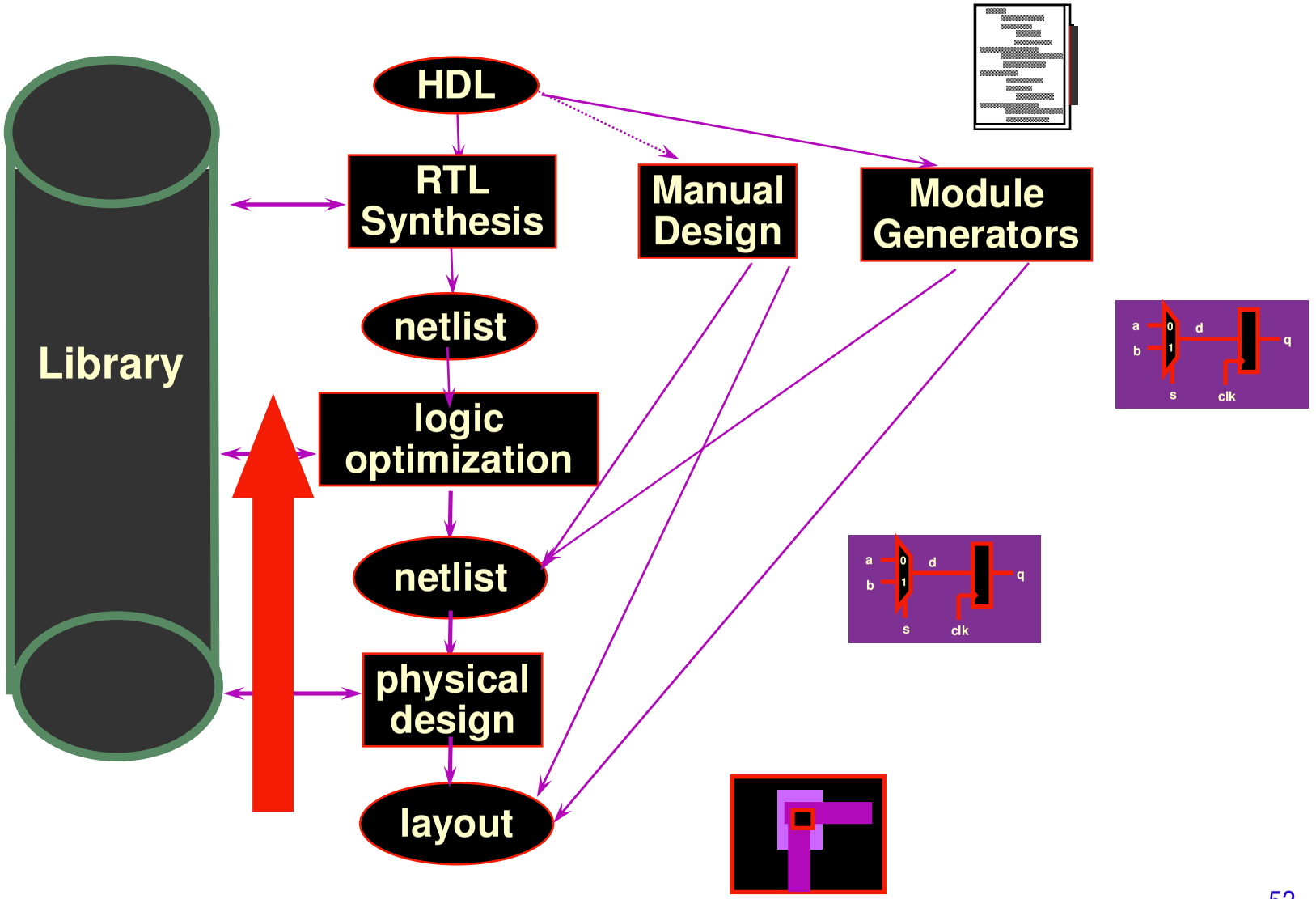


$a \geq b + 1$



$a \geq b + 1$

Next Month: Optimization, Testing, Verification



Background Material

Handout in Reader:

Chapter 10 in *Combinatorial Algorithms for IC Layout,* by Thomas Lengauer, Wiley and Teubner, 1990.

Algorithms background:

Introduction to Algorithms, T. Cormen, C. Lesierson, R. Rivest (& Stern), The MIT Press, Second or later printing.

- shortest paths (read selectively)
- union-find algorithm (disjoint forest implementation)