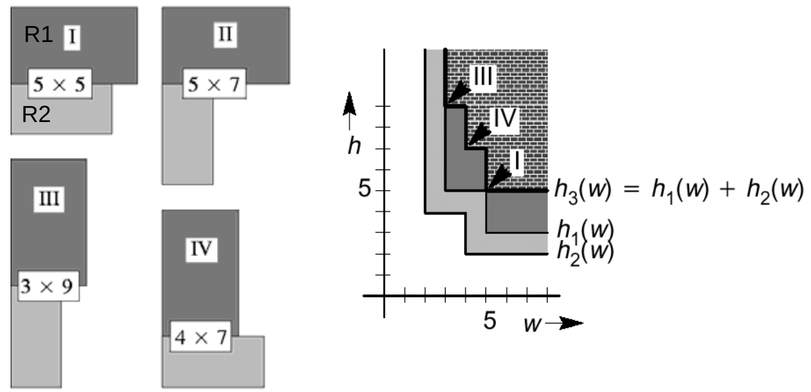
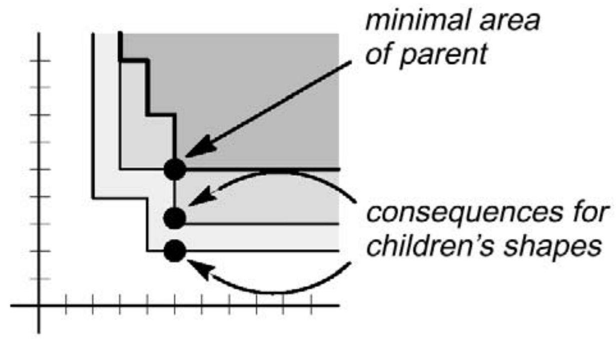
Addition for Vertical Abutment

- Composition by vertical abutment ⇒ the addition of shape functions.



Deriving Shapes of Children

- A choice for the minimal shape of composite cell fixes the shapes of the shapes of its children cells.

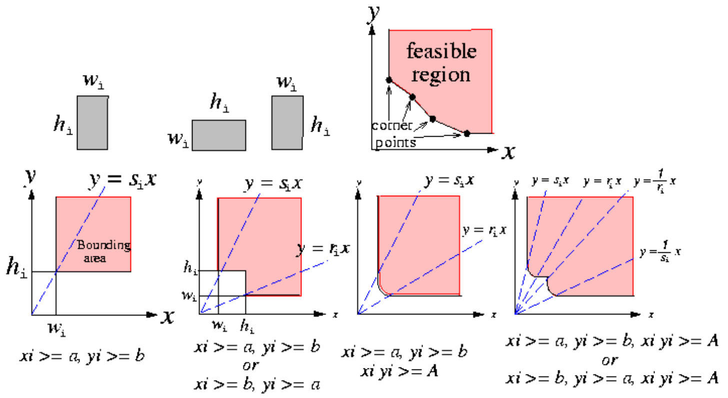


Sizing Algorithm for Slicing Floorplans

- The shape functions of all leaf cells are given as piecewise linear functions.
- Traverse the slicing tree in order to compute the shape functions of all composite cells (bottom-up composition).
- Choose the desired shape of the top-level cell; as the shape function is piecewise linear, only the break points of the function need to be evaluated, when looking for the minimal area.
- Propagate the consequences of the choice down to the leaf cells (top-down propagation).
- The sizing algorithm runs in polynomial time for slicing floorplans
  - NP-complete for non-slicing floorplans

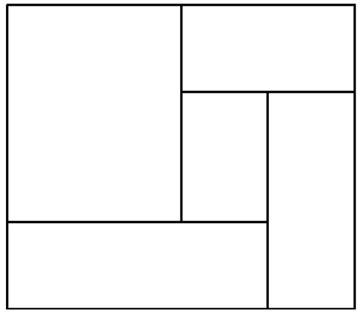
Feasible Implementations

- Shape curves correspond to different kinds of constraints where the shaded areas are feasible regions.



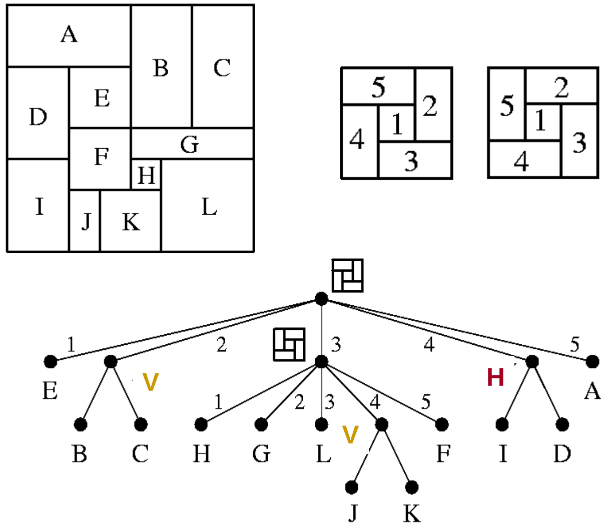
(a) rigid, fixed orientation (b) rigid, free orientation (c) flexible, fixed orientation (d) flexible, free orientation

Wheel or Spiral Floorplan



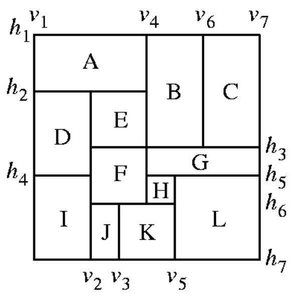
- This floorplan is not slicing!
- **Wheel** is the smallest non-slicing floorplans.
- Limiting floorplans to those that have the slicing property is reasonable: it certainly facilitates floorplanning algorithms.
- Taking the shape of a wheel floorplan and its mirror image as the basis of operators leads to hierarchical descriptions of *order 5*.

Order-5 Floorplan Examples

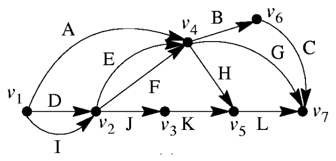


General Floorplan Representation: Polar Graphs

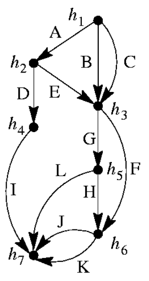
- vertex: channel segment
- edge: cell/block/module



horizontal polar graph

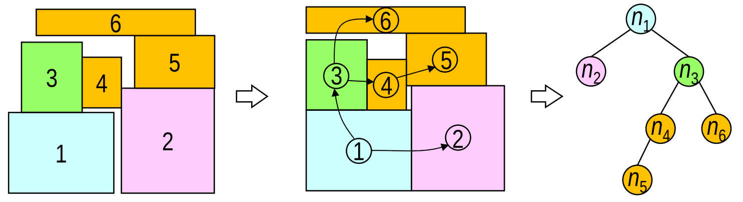


vertical polar graph



B*-Tree: Compacted Floorplan Representation

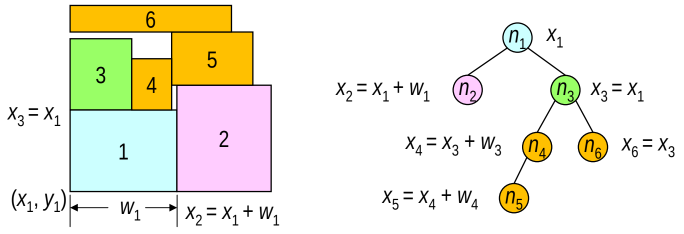
- Chang et al., "B*-tree: A new representation for non-slicing floorplans," DAC 2000.
  - Compact modules to left and bottom
  - Construct an ordered binary tree (B*-tree)
    - Left child: the lowest, adjacent block on the right ($x_j = x_i + w_i$)
    - Right child: the first block above, with the same x-coordinate ($x_j = x_i$)



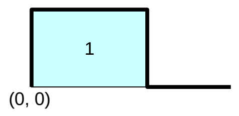
A non-slicing floorplan Compact to left and down B*-tree

B*-tree Packing

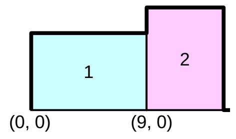
- x-coordinates can be determined by the tree structure
  - Left child: the lowest, adjacent block on the right ($x_j = x_i + w_i$)
  - Right child: the first block above, with the same x-coordinate ($x_j = x_i$)
- Y-coordinates?
  - Horizontal contour: Use a doubly linked list to record the current maximum y-coordinate for each x-range
  - Reduce the complexity of computing a y-coordinate to amortized O(1) time



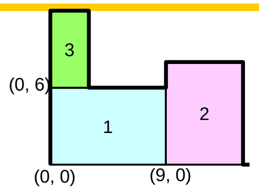
Contour Data Structure



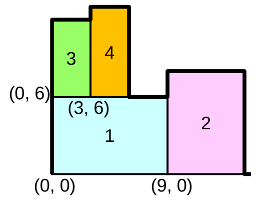
C = <(0,0), **(0,6), (9,6), (9,0),** (∞,0)>



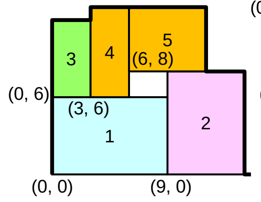
C = <(0,0), (0,6), (9,6), **(9,8)**, **(15,8), (15,0)**, (∞,0)>



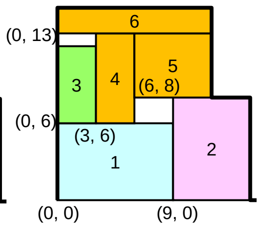
C = <(0,0), **(0,12), (3,12), (3,6),** (9,6), (9,8), (15,8), (15,0), (∞,0)>



C = <(0,0), (0,12), (3,12), **(3,13), (6,13), (6,6),** (9,6), (9,8), (15,8), (15,0), (∞,0)>



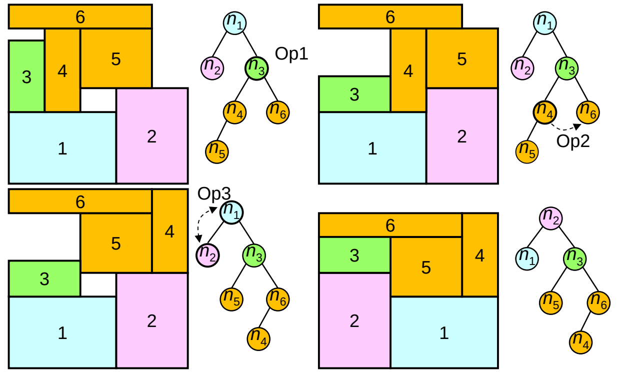
C = <(0,0), (0,12), (3,12), (3,13), **(6,13), (12,13), (12,8),** (15,8), (15,0), (∞,0)>



C = <(0,0), **(0,15), (12,15), (12,13),** (12,8)**,** (15,8), (15,0), (∞,0)>

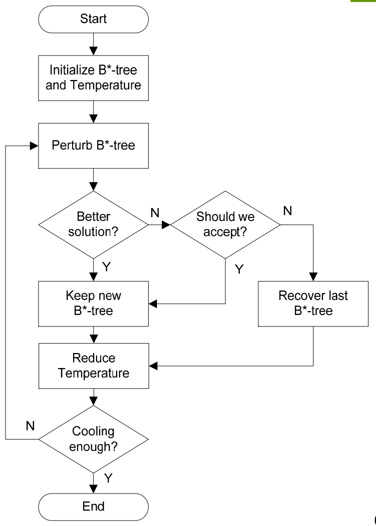
B*-tree Perturbation

- Op1: rotate a macro
- Op2: move a node to another place
- Op3: swap two nodes



Simulated Annealing Using B*-tree

- The cost function is based on problem requirements

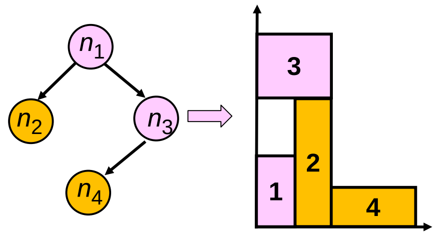


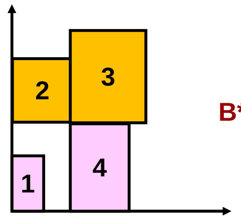
Strengths of B*-tree

- Binary tree based, efficient and easy
- Flexible to deal with various placement constraints by augmenting the B*-tree data structure (e.g., preplaced, symmetry, alignment, bus position) and rectilinear modules
- Transformation between a tree and its placement takes only linear time (vs. $O(n^2)$ or $O(n \lg \lg n)$ for sequence pair to be shown shortly)
- Operate on only one B*-tree (vs. two O-trees)
- Can evaluate area cost incrementally
- Smaller solution space: only $O(n!\, 4^n/n^{1.5})$ combinations (vs. $O((n!)^2)$ for sequence pair)
- Directly corresponds to hierarchical and multilevel frameworks for large-scale floorplan designs
- Can be extended to 3D floorplanning & related applications

Weaknesses of B*-tree

- Representation may change after packing
- Only a partially topological representation; less flexible than a fully topological representation
  - B*-tree can represent only compacted placement





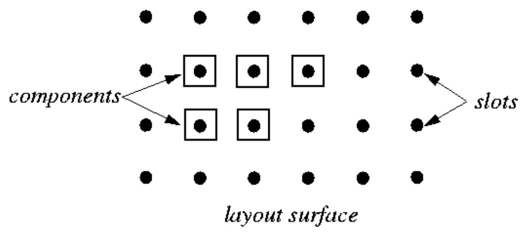
B*-tree??

Outline

- Partitioning
- Floorplanning
- Placement
- Routing
- Compaction

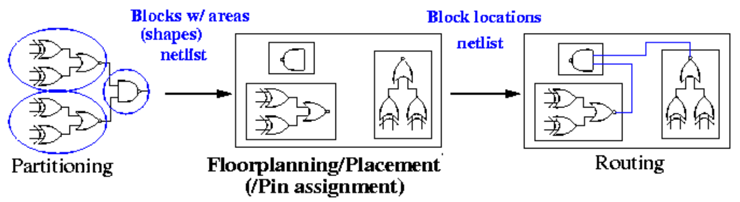
Placement

- Course contents:
  - Placement metrics
  - Constructive placement: cluster growth, min cut
  - Iterative placement: force-directed method, simulated annealing
- Reading
  - Chapter 11



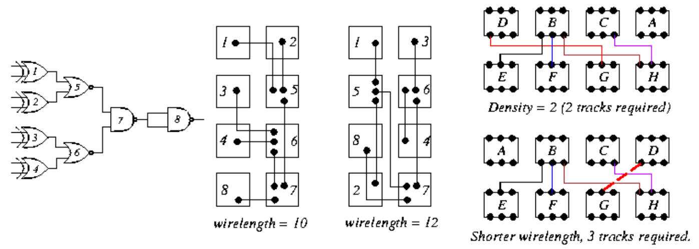
Placement

- **Placement** is the problem of automatically assigning correct positions on the chip to predesigned cells, such that some cost function is optimized.
- Inputs: A set of **fixed** cells/modules, a netlist.
- Goal: Find the best position for each cell/module on the chip according to appropriate cost functions.
  - Considerations: **routability/channel density**, **wirelength**, cut size, performance, thermal issues, I/O pads.



Placement Objectives and Constraints

- What does a placement algorithm try to optimize?
  - total area
  - total wire length
  - number of horizontal/vertical wire segments crossing a line
- Constraints:
  - placement should be routable (no cell overlaps; no density overflow).
  - timing constraints are met (some wires should always be shorter than a given length).

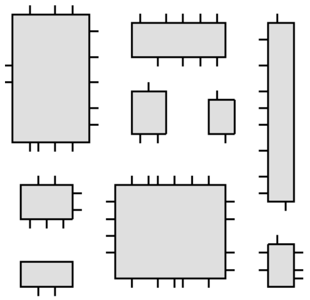


1
2
3
4
5
6
7
8
wirelength = 10
wirelength = 12
D
B
C
A
E
F
G
H
Density = 2 (2 tracks required)
Shorter wirelength, 3 tracks required.

VLSI Placement: Building Blocks

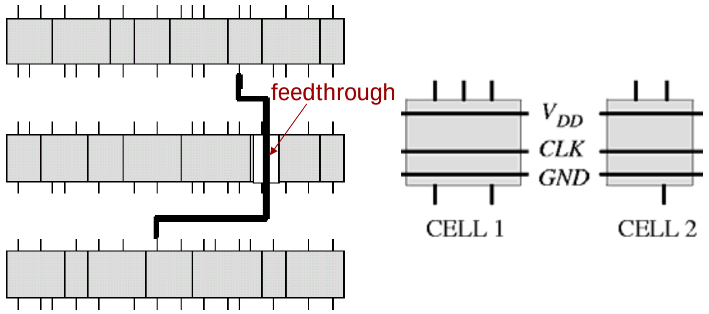
- Different design styles create different placement problems.
  - E.g., building-block, standard-cell, gate-array placement
    - Building block: The cells to be placed have arbitrary shapes.

building block example



VLSI Placement: Standard Cells

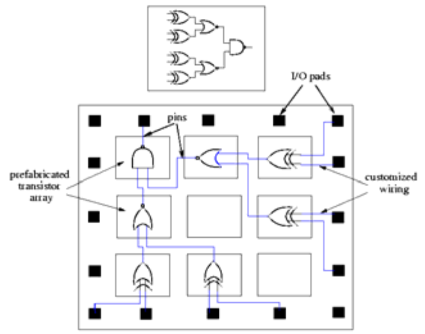
- Standard cells are designed in such a way that power and clock connections run horizontally through the cell and other I/O leaves the cell from the top or bottom sides.
- The cells are placed in rows.
- Sometimes feedthrough cells are added to ease wiring.



Consequences of Fabrication Method

- Full-custom fabrication (building block):
  - Free selection of aspect ratio (quotient of height and width).
  - Height of wiring channels can be adapted to necessity.
- Semi-custom fabrication (gate array, standard cell):
  - Placement has to deal with fixed carrier dimensions.
  - Placement should be able to deal with fixed channel capacities.

gate array



Relation with Routing

- Ideally, placement and routing should be performed simultaneously as they depend on each other’s results. This is, however, too complicated.
  - P&R: placement and routing
- In practice placement is done prior to routing. The placement algorithm estimates the wire length of a net using some *metric*.

Wirelength Estimation

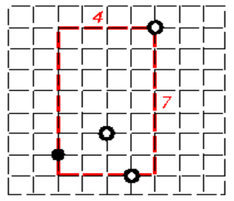
- **Semi-perimeter method:** Half the perimeter of the bounding rectangle that encloses all the pins of the net to be connected. Most widely used approximation!
- **Steiner-tree approximation:** Computationally expensive.
- **Minimum spanning tree**: Good approximation to Steiner trees.
- **Squared Euclidean distance:** Squares of all pairwise terminal distances in a net using a quadratic cost function

$$\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\gamma_{ij}[(x_i - x_j)^2 + (y_i - y_j)^2]$$

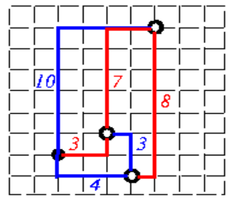
- **Complete graph:** Since #edges in a complete graph is $\binom{n(n-1)}{2}$,

wirelength ≈ $\frac{2}{n}\sum_{(i,\ j)\ \in\ net} dist(i,\ j)$.

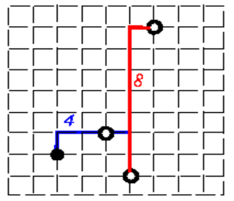
Wirelength Estimation (cont'd)



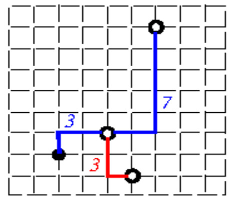
semi–perimeter len = 11



*complete graph len * 2/n = 17.5*



Steiner tree len = 12



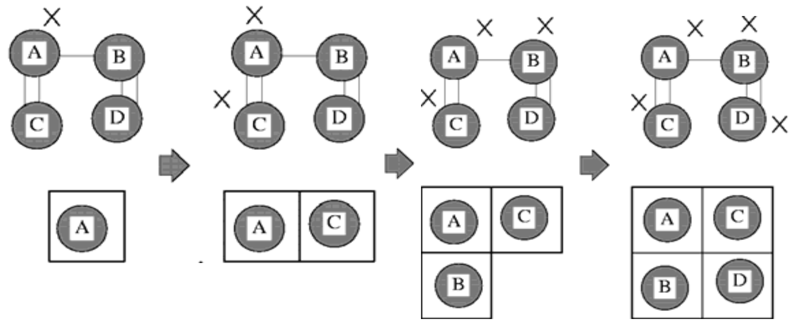
Spanning tree len = 13

Placement Algorithms

- The placement problem is NP-complete
- Popular placement algorithms:
  - **Constructive algorithms:** once the position of a cell is fixed, it is not modified anymore.
    - Cluster growth, min cut, etc.
  - **Iterative algorithms:** intermediate placements are modified in an attempt to improve the cost function.
    - Force-directed method, etc
  - **Nondeterministic approaches:** simulated annealing, genetic algorithm, etc.
- Most approaches combine multiple elements:
  - Constructive algorithms are used to obtain an initial placement.
  - The initial placement is followed by an iterative improvement phase.
  - The results can further be improved by simulated annealing.

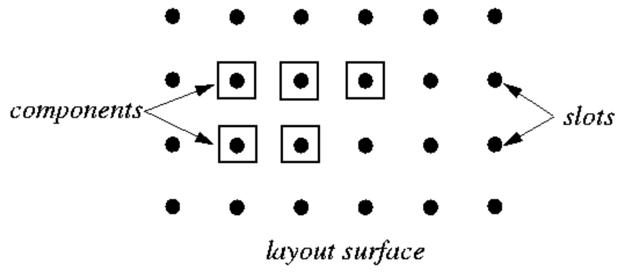
Bottom-Up Placement: Clustering

- Starts with a single cell and finds more cells that share nets with it.



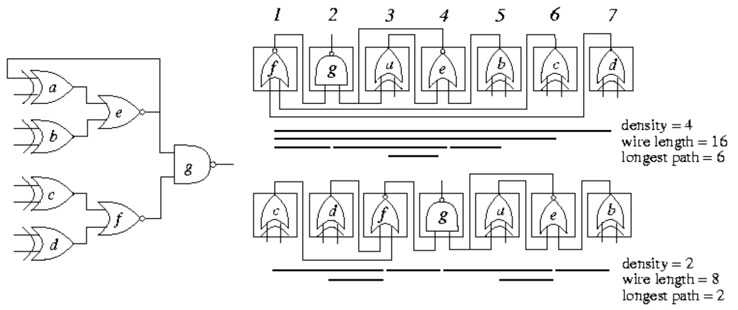
Placement by Cluster Growth

- Greedy method: Selects unplaced components and places them in available slots.
  - SELECT: Choose the unplaced component that is most strongly connected to all of the placed components (or most strongly connected to any single placed component).
  - PLACE: Place the selected component at a slot such that a certain “cost” of the partial placement is minimized.



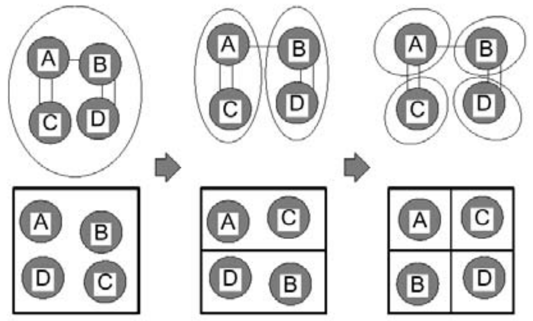
Cluster Growth Example

- # of other terminals connected: c_a=3, c_b=1, c_c=1, c_d =1, c_e=4, c_f=3, and c_g=3 ⇒ *e* has the most connectivity.
- Place *e* in the center, slot 4. *a, b, g* are connected to *e*, and ⇒ Place *a* next to *e* (say, slot 3). Continue until all cells are placed.
- Further improve the placement by swapping the gates.



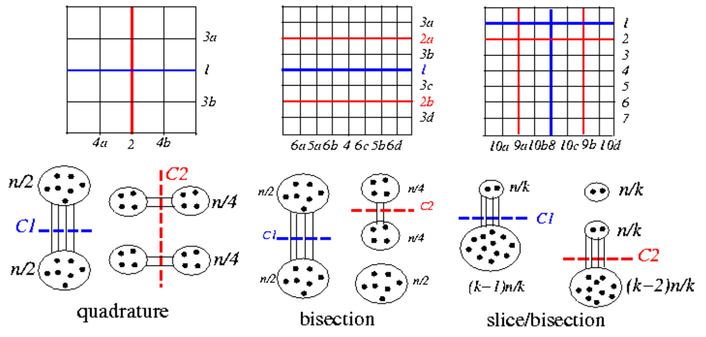
Top-down Placement: Min Cut

- Starts with the whole circuit and ends with small circuits.
- Recursive bipartitioning of a circuit (e.g., K&L) leads to a min-cut placement.



Min-Cut Placement

- Breuer, “A class of min-cut placement algorithms,” DAC, 1977.
- **Quadrature:** suitable for circuits with high density in the center.
- **Bisection:** good for standard-cell placement.
- **Slice / Bisection:** good for cells with high interconnection on the periphery.

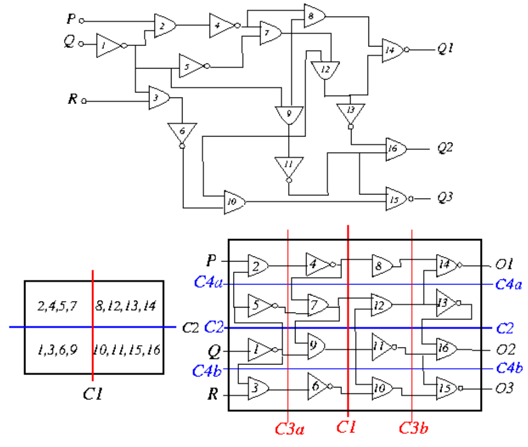


Algorithm for Min-Cut Placement

```
Algorithm: Min_Cut_Placement(N, n, C)
/* N: the layout surface */
/* n : # of cells to be placed */
/* n0: # of cells in a slot */
/* C: the connectivity matrix */

1 begin
2 if (n ≤ n0) then PlaceCells(N, n, C)
3 else
4     (N1, N2) ← CutSurface(N);
5     (n1, C1), (n2, C2) ← Partition(n, C);
6  Call Min_Cut_Placement(N1, n1, C1);
7  Call Min_Cut_Placement(N2, n2, C2);
8 end
```

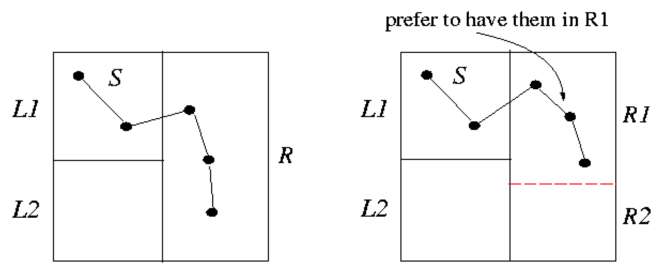
Quadrature Placement Example

- Apply the K-L heuristic to partition + Quadrature Placement: Cost $C_1 = 4$, $C_{2L} = C_{2R} = 2$, etc.



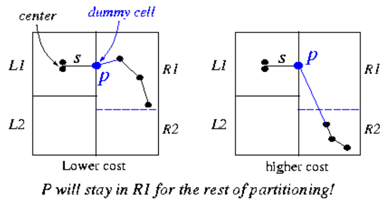
Min-Cut Placement with Terminal Propagation

- Dunlop & Kernighan, “A procedure for placement of standard-cell VLSI circuits,” *IEEE TCAD*, Jan. 1985.
- Drawback of the original min-cut placement: Does not consider the positions of terminal pins that enter a region.
  - What happens if we swap {1, 3, 6, 9} and {2, 4, 5, 7} in the previous example?

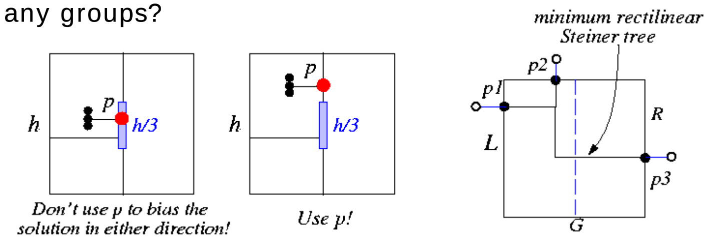


Terminal Propagation

- We should use the fact that *s* is in L_1!

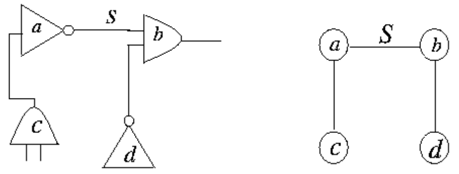


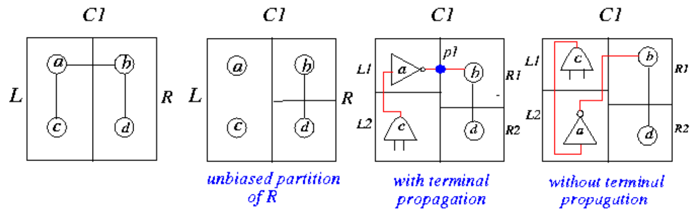
- When not to use *p* to bias partitioning? Net *s* has cells in many groups?


p
h
h/3
Don't use p to bias the solution in either direction!
p
h
h/3
Use p!
minimum rectilinear Steiner tree
p2
p1
R
L
p3
G

Terminal Propagation Example

- Partitioning must be done breadth-first, not depth-first.



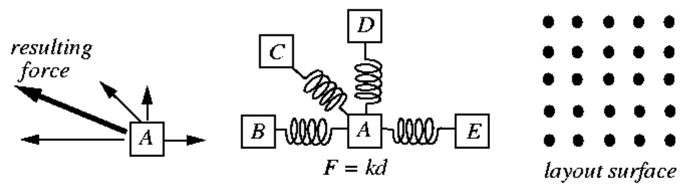


General Procedure for Iterative Improvement

```
Algorithm: Iterative_Improvement()
begin
s ← initial_configuration();
c ← cost(s);
while (not stop()) do
    s′ ← perturb(s);
    c′ ← cost(s′);
    if (accept(c, c′))
  then   s ← s′;
end
```

Placement by the Force-Directed Method

- Hanan & Kurtzberg, “Placement techniques,” in *Design Automation of Digital Systems*, Breuer, Ed, 1972.
- Quinn, Jr. & Breuer, “A force directed component placement procedure for printed circuit boards,” *IEEE Trans. Circuits and Systems*, June 1979.
- Reduce the placement problem to solving a set of simultaneous linear equations to determine equilibrium locations for cells.
- Analogy to Hooke's law: $F = kd$, F: force, k: spring constant, d: distance.
- Goal: Map cells to the layout surface.



Finding the Zero-Force Target Location

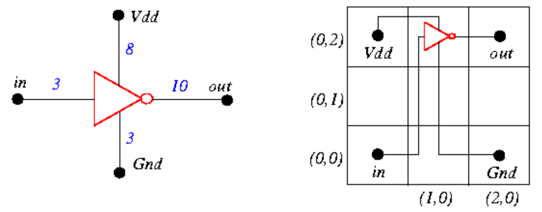
- Cell i connects to several cells j's at distances d_{ij}'s by wires of weights w_{ij}'s. Total force: $F_i = \sum_j w_{ij} d_{ij}$
- The zero-force target location ($\hat{x}_i$, $\hat{y}_i$) can be determined by equating the x- and y-components of the forces to zero:

$$\sum_j w_{ij} \cdot (x_j - \hat{x}_i) = 0 \quad \Rightarrow \quad \hat{x}_i = \frac{\sum_j w_{ij} x_j}{\sum_j w_{ij}}$$

$$\sum_j w_{ij} \cdot (y_j - \hat{y}_i) = 0 \quad \Rightarrow \quad \hat{y}_i = \frac{\sum_j w_{ij} y_j}{\sum_j w_{ij}}$$

- In the example, and $\hat{y}_i$ = 1.50.

$$\hat{x}_i = \frac{8\times0+10\times2+3\times0+3\times2}{8+10+3+3} = 1.083$$



Force-Directed Placement

- Can be constructive or iterative:
  - Start with an initial placement.
  - Select a “most profitable” cell p (e.g., maximum F, critical cells) and place it in its zero-force location.
  - “Fix” placement if the zero-location has been occupied by another cell q.
    - Popular options to fix:
      - **Ripple move:** place p in the occupied location, compute a new zero-force location for q, …
      - **Chain move:** place p in the occupied location, move q to an adjacent location, …
      - Move p to a free location close to q.

Force-Directed Placement

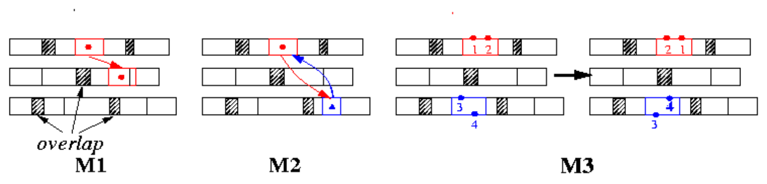
```
Algorithm: Force-Directed_Placement

1 begin
2 Compute the connectivity for each cell;
3 Sort the cells in decreasing order of their connectivities into list $L$;
4 while ($IterationCount < IterationLimit$) do
5     Seed ← next module from $L$;
6     Declare the position of the seed vacant;
7         while ($EndRipple = FALSE$) do
8             Compute target location of the seed;
9             case the target location
10            VACANT:
11                Move seed to the target location and lock;
12                $EndRipple \leftarrow TRUE$; $AbortCount \leftarrow 0$;
13            SAME AS PRESENT LOCATION:
14                $EndRipple \leftarrow TRUE$; $AbortCount \leftarrow 0$;
15            LOCKED:
16                Move selected cell to the nearest vacant location;
17                $EndRipple \leftarrow TRUE$; $AbortCount \leftarrow AbortCount + 1$;
18                if ($AbortCount > AbortLimit$) then
19                    Unlock all cell locations;
19                    $IterationCount \leftarrow IterationCount + 1$;
20            OCCUPIED AND NOT LOCKED:
21                Select cell as the target location for next move;
22                Move seed cell to target location and lock the target location;
23                $EndRipple \leftarrow FALSE$; $AbortCount \leftarrow 0$;
26 end
```

Placement by Simulated Annealing

- Sechen and Sangiovanni-Vincentelli, “The TimberWolf placement and routing package,” *IEEE J. Solid-State Circuits*, Feb. 1985; “TimberWolf 3.2: A new standard cell placement and global routing package,” DAC-86.
- TimberWolf: Stage 1
  - Modules are moved between different rows as well as within the same row.
  - Modules overlaps are allowed.
  - When the temperature is reached below a certain value, stage 2 begins.
- TimberWolf: Stage 2
  - Remove overlaps.
  - Annealing process continues, but only interchanges adjacent modules within the same row.

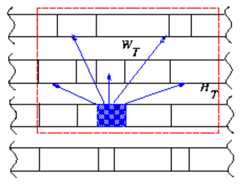
Solution Space & Neighborhood Structure

- **Solution Space:** All possible arrangements of the modules into rows, possibly with overlaps.
- **Neighborhood Structure:** 3 types of moves
  - M_1: Displace a module to a new location.
  - M_2: Interchange two modules.
  - M_3: Change the orientation of a module.



Neighborhood Structure

- TimberWolf first tries to select a move between M_1 and M_2: $Prob(M_1) = 0.8$, $Prob(M_2) = 0.2$.
- If a move of type M_1 is chosen and it is rejected, then a move of type M_3 for the same module will be chosen with probability 0.1.
- Restrictions: (1) what row for a module can be displaced? (2) what pairs of modules can be interchanged?
- **Key: Range Limiter**
  - At the beginning, (W_T, H_T) is big enough to contain the whole chip.
  - Window size shrinks as temperature decreases. Height & width $\propto$ $log(T)$.
  - Stage 2 begins when window size is so small that no inter-row module interchanges are possible.



Cost Function

- Cost function: $C = C_1 + C_2 + C_3$.
- C_1: total estimated wirelength.
  - $C_1 = \sum_{i \in Nets}(\alpha_i\, w_i + \beta_i\, h_i)$
  - α_i, β_i are horizontal and vertical weights, respectively. (α_i=1, β_i=1 $\Rightarrow$ half perimeter of the bounding box of Net *i*.)
  - Critical nets: Increase both α_i and β_i .
  - If vertical wirings are “cheaper” than horizontal wirings, use smaller vertical weights: $\beta_i < \alpha_i$.
- C_2: penalty function for module overlaps.
  - $C_2 = \gamma \sum_{i \neq j} O^2_{ij}$, γ: penalty weight.
  - O_{ij}: amount of overlaps in the *x*-dimension between modules *i* and *j*.
- C_3: penalty function that controls the row length.
  - $C_2 = \delta \sum_{r \in Rows}|L_r - D_r|$, δ : penalty weight.
  - D_r: desired row length.
  - L_r: sum of the widths of the modules in row *r*.

Annealing Schedule

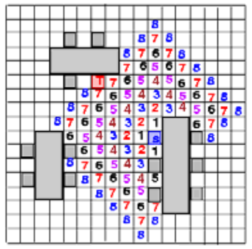
- $T_k = r_k T_{k-1}$, k = 1, 2, 3, …
- r_k increases from 0.8 to max value 0.94 and then decreases to 0.8.
- At each temperature, a total # of *nP* attempts is made.
- *n*: # of modules; *P*: user specified constant.
- Termination: $T < 0.1$.

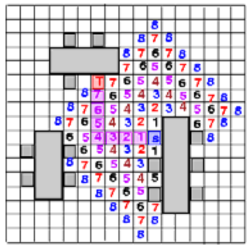
Outline

- Partitioning
- Floorplanning
- Placement
- Routing
  - Global rounting
  - Detailed routing
- Compaction

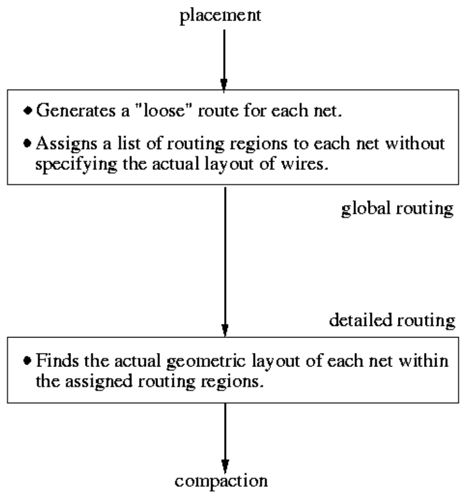
Routing

- Course contents:
  - Global routing
  - Detail rounting
- Reading
  - Chapter 12

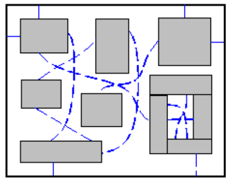

Filling

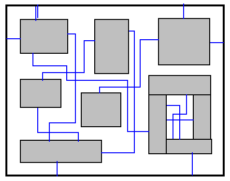

Retrace

Routing



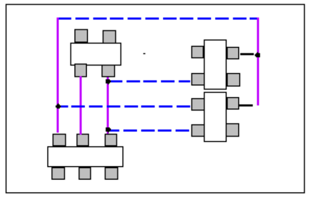
placement
• Generates a "loose" route for each net.
• Assigns a list of routing regions to each net without specifying the actual layout of wires.
global routing
detailed routing
• Finds the actual geometric layout of each net within the assigned routing regions.
compaction

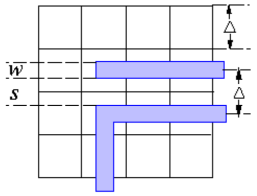

Global routing


Detailed routing

Routing Constraints

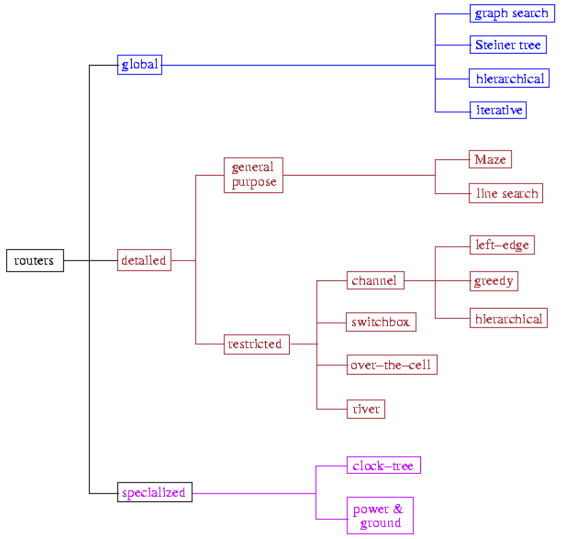
- 100% routing completion + area minimization, under a set of constraints:
  - Placement constraint: usually based on fixed placement
  - Number of routing layers
  - Geometrical constraints: must satisfy design rules
  - Timing constraints (performance-driven routing): must satisfy delay constraints
  - Crosstalk?
  - Process variations?


Two–layer routing



Geometrical constraint

Classification of Routing

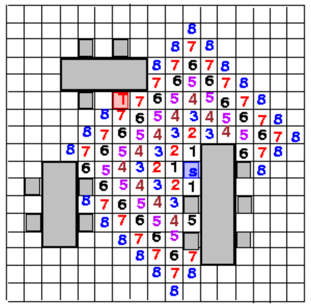


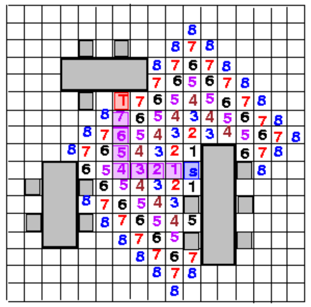
Maze Router: Lee Algorithm

- Lee, "An algorithm for path connection and its application," *IRE Trans. Electronic Computer*, EC-10, 1961.
- Discussion mainly on single-layer routing
- **Strengths**
  - Guarantee to find connection between 2 terminals if it exists.
  - Guarantee minimum path.
- **Weaknesses**
  - Requires large memory for dense layout.
  - Slow.
- Applications: global routing, detailed routing

Lee Algorithm

- Find a path from *S* to *T* by "wave propagation".


Filling


Retrace

- Time & space complexity for an $M \times N$ grid: $O(MN)$ (**huge!**)