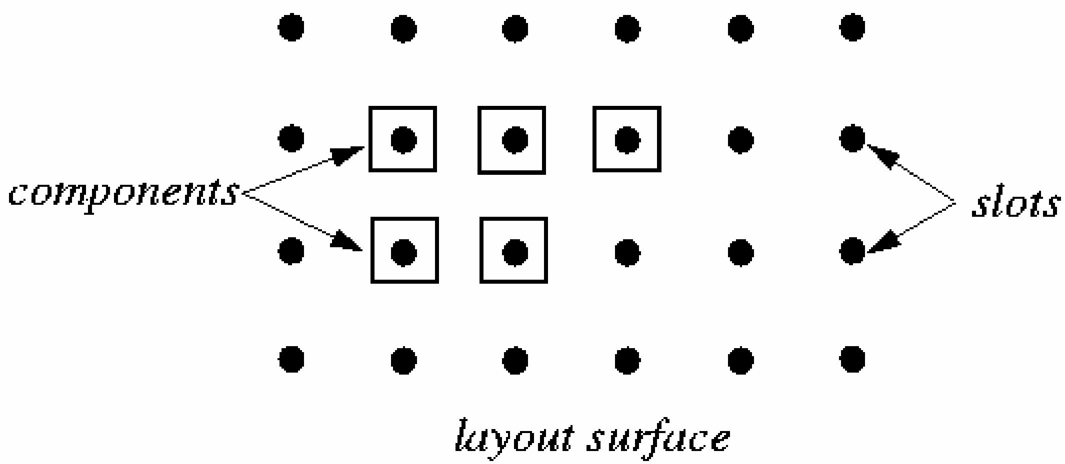
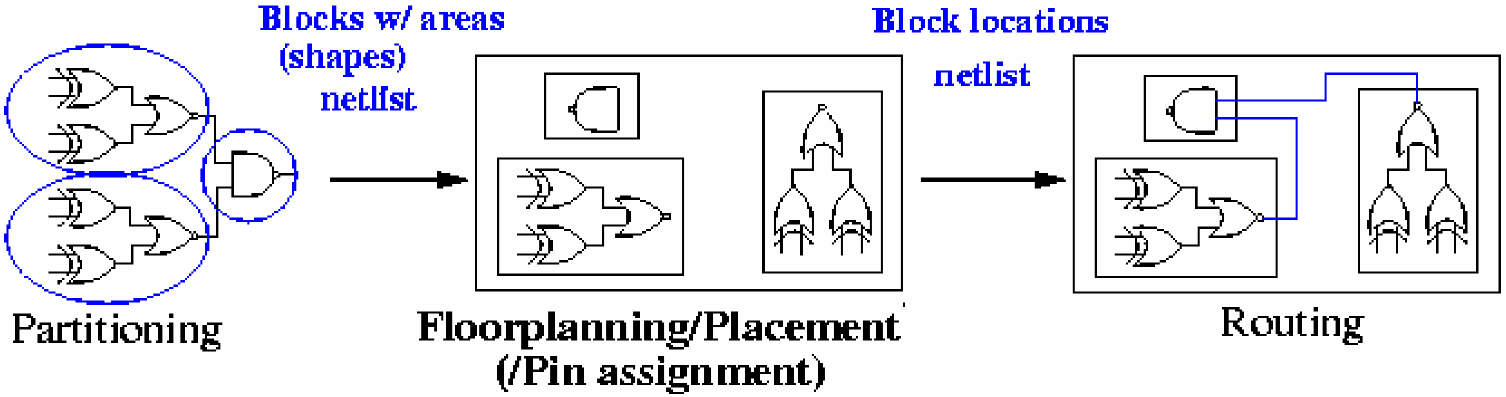
# Unit 5C: Placement

- Course contents:
  - Placement metrics
  - Constructive placement: cluster growth, min cut
  - Iterative placement: force-directed method, simulated annealing, genetic algorithm
- Readings
  - Chapter 7.1--7.4
  - Chapter 5.8



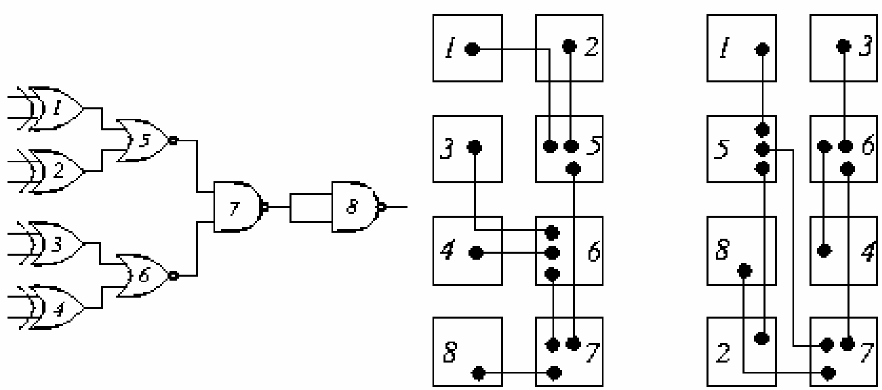
# Placement

- **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.

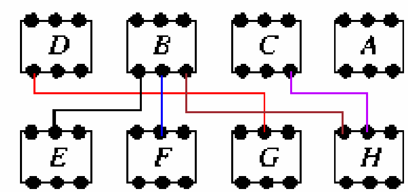


# Placement Objectives and Constraints

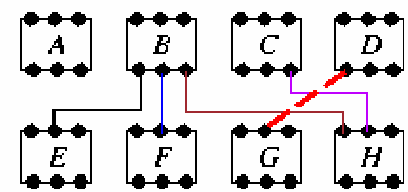
- What does a placement algorithm try to optimize?
  - the total area
  - the total wire length
  - the number of horizontal/vertical wire segments crossing a line
- Constraints:
  - the placement should be routable (no cell overlaps; no density overflow).
  - timing constraints are met (some wires should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



wirelength = 10 wirelength = 12



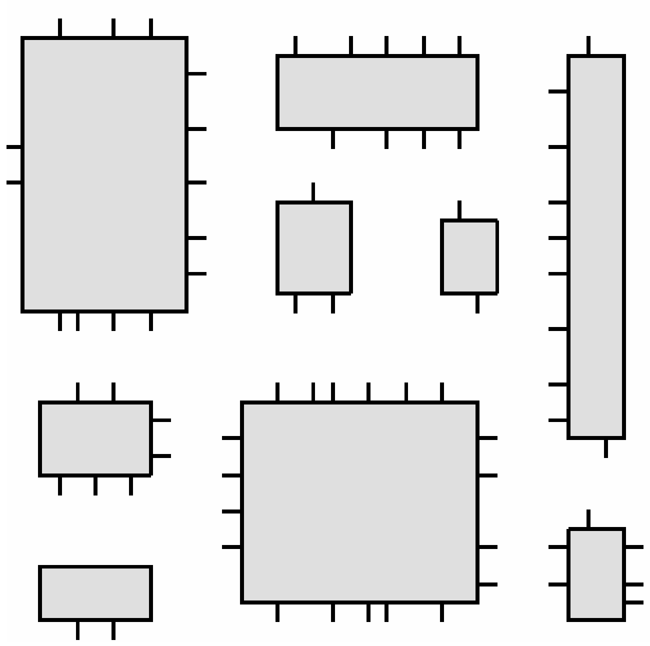

Density = 2 (2 tracks required)




Shorter wirelength, 3 tracks required.

# VLSI Placement: Building Blocks

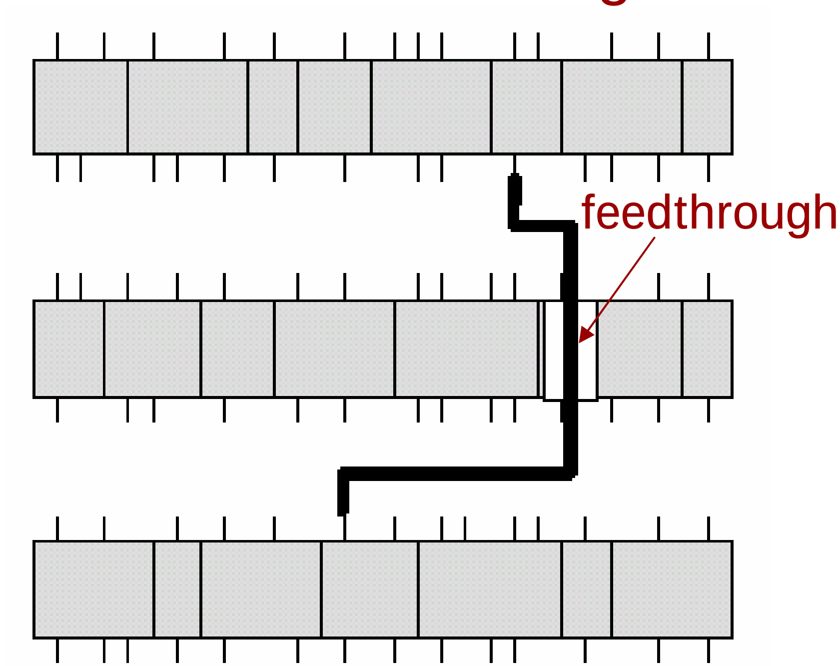
- Different design styles create different placement problems.
  – E.g., building-block, standard-cell, gate-array placement
- Building block: The cells to be placed have arbitrary shapes.

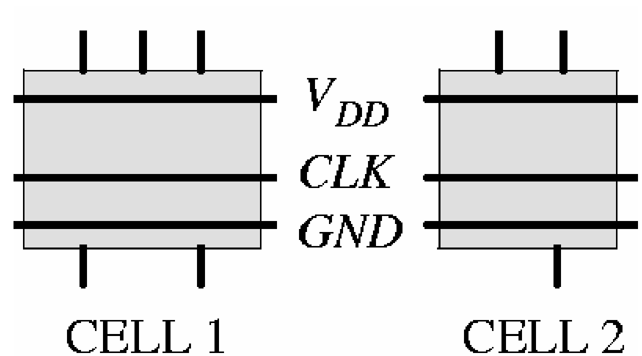


Building block

# 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.

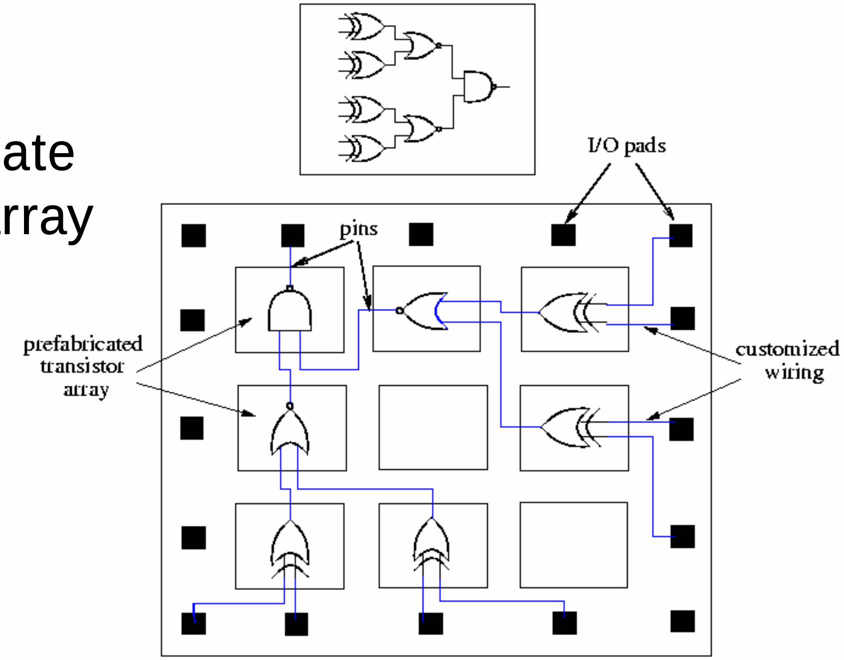


# Consequences of Fabrication Method

- 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*.

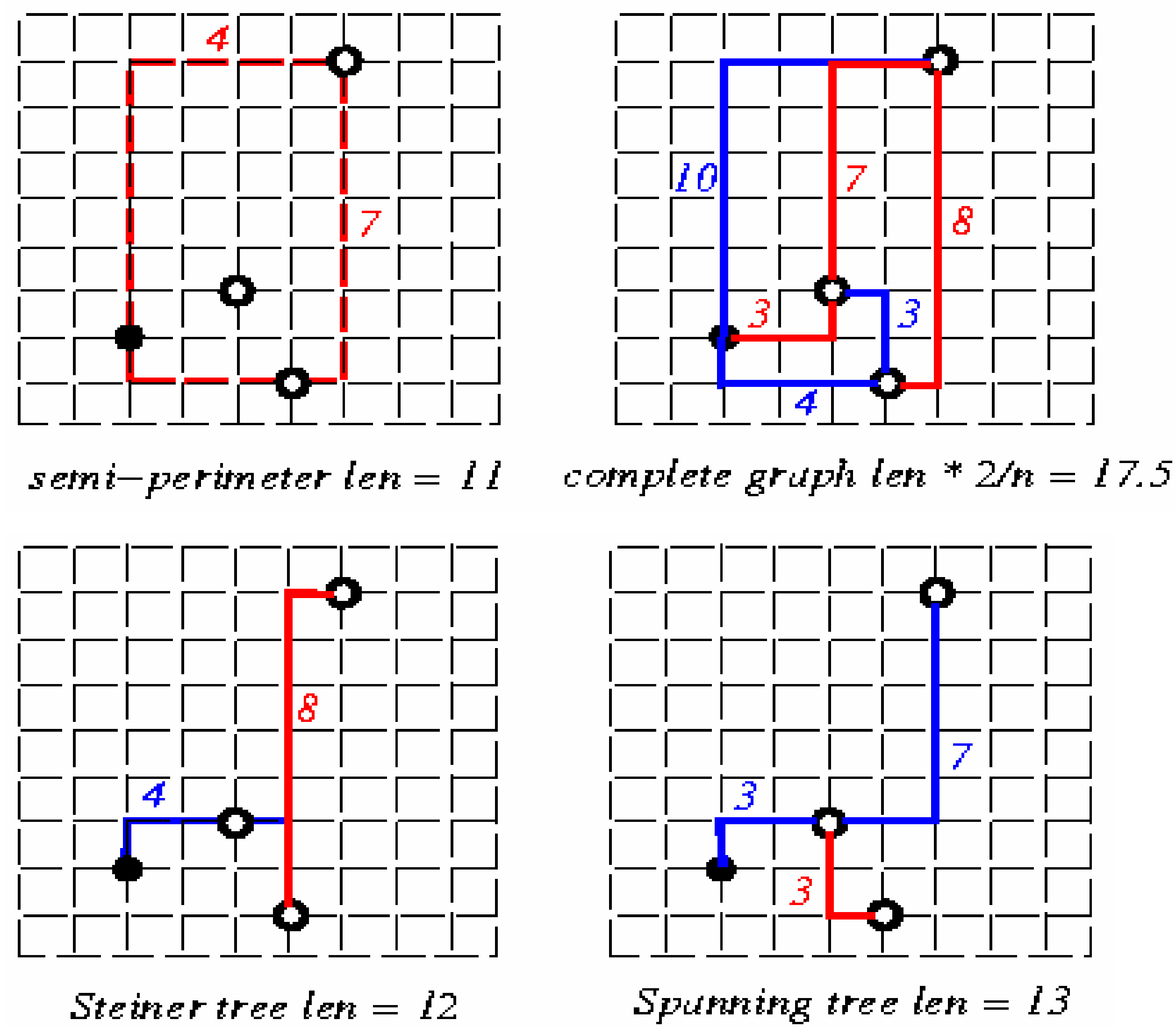
# Estimation of Wirelength

- **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!
- **Squared Euclidean distance:** Squares of all pairwise terminal distances in a net using a quadratic cost function

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

- **Steiner-tree approximation:** Computationally expensive.
- **Minimum spanning tree**: Good approximation to Steiner trees.
- **Complete graph:** Since #edges in a complete graph is $\left(\frac{n(n-1)}{2}\right)$ = $\frac{n}{2}\times$ # of tree edges ($n$-1), *wirelength* $\approx$ $\frac{2}{n}\sum_{(i,\, j)\, \in\, net} dist(i, j)$.

# Estimation of Wirelength (cont'd)

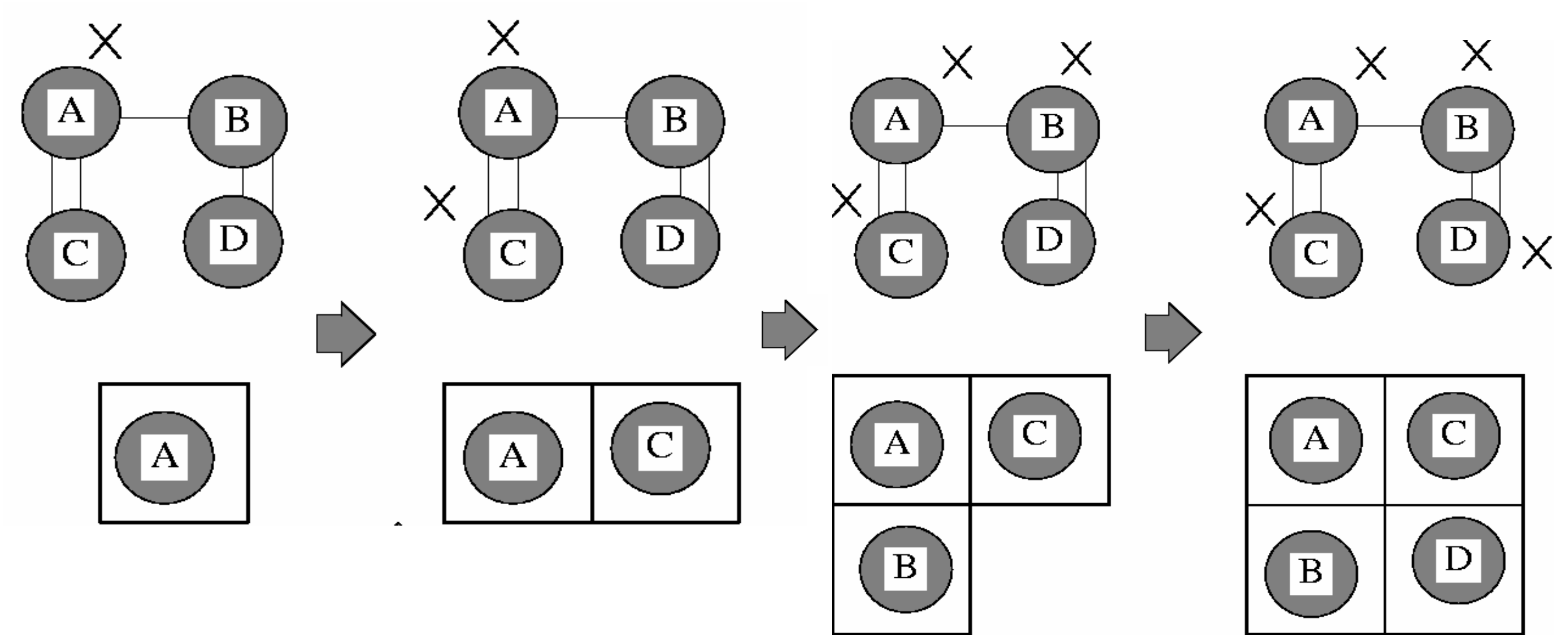


# Placement Algorithms

- 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.

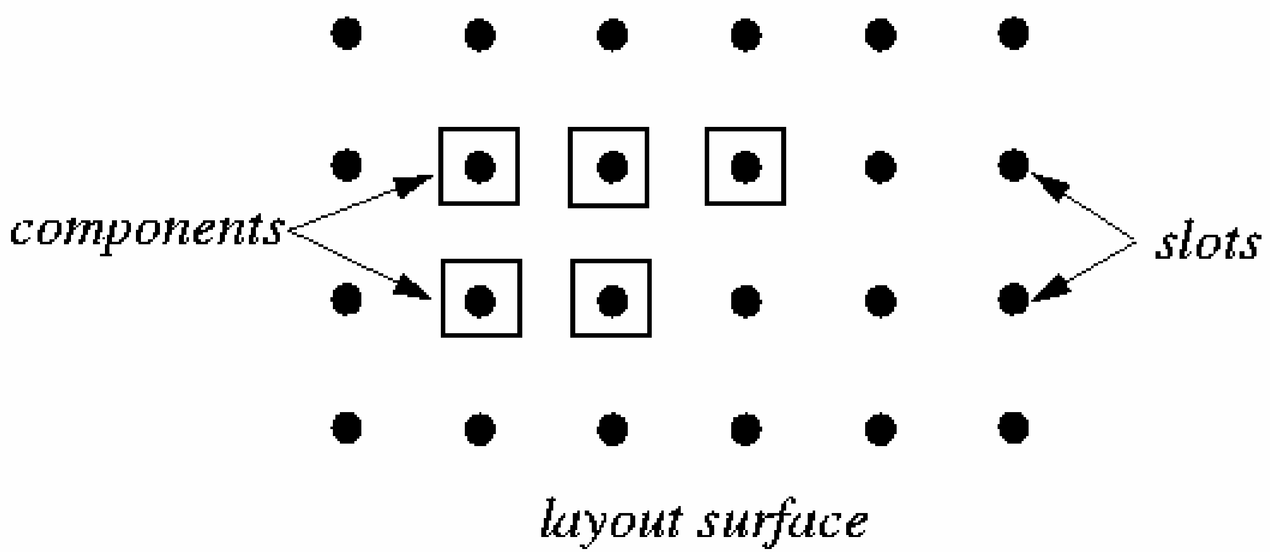
# Bottom-Up Placement: Clustering

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



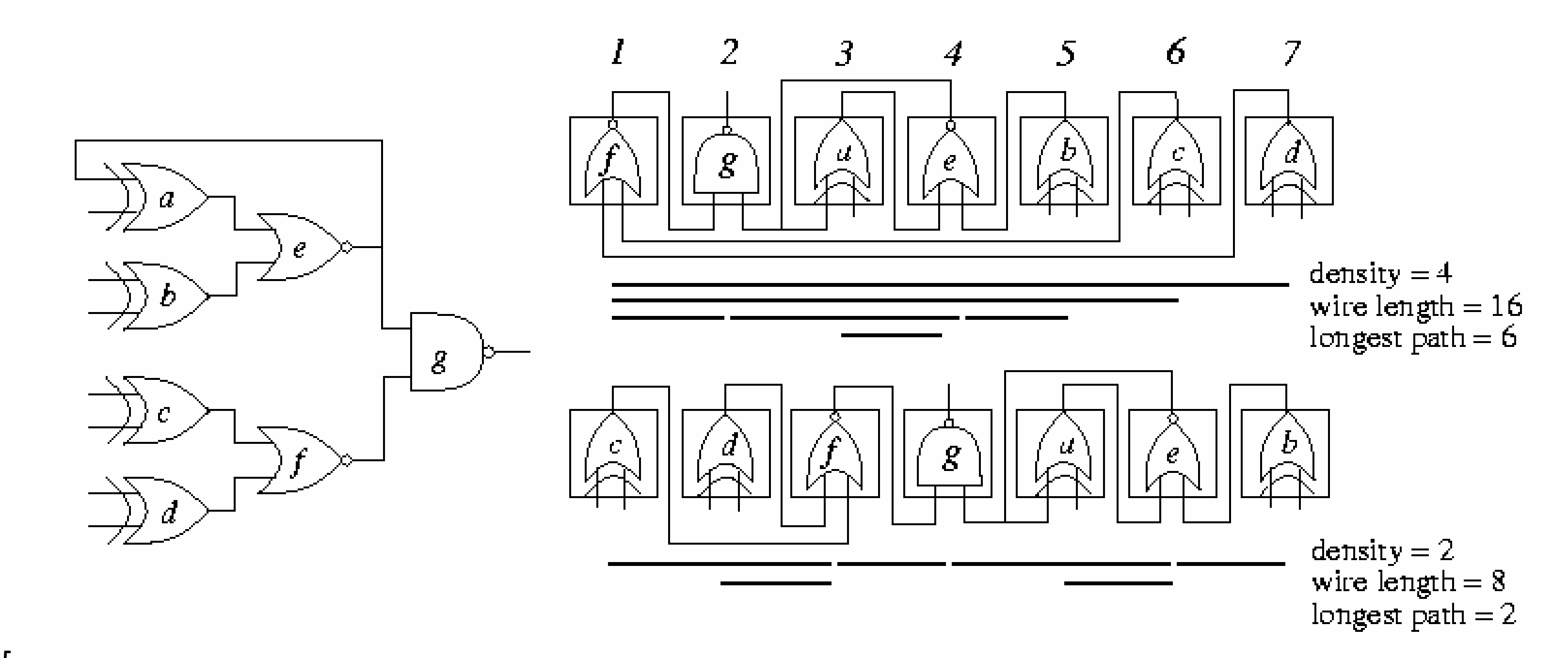
# Placement by Cluster Growth

- 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.



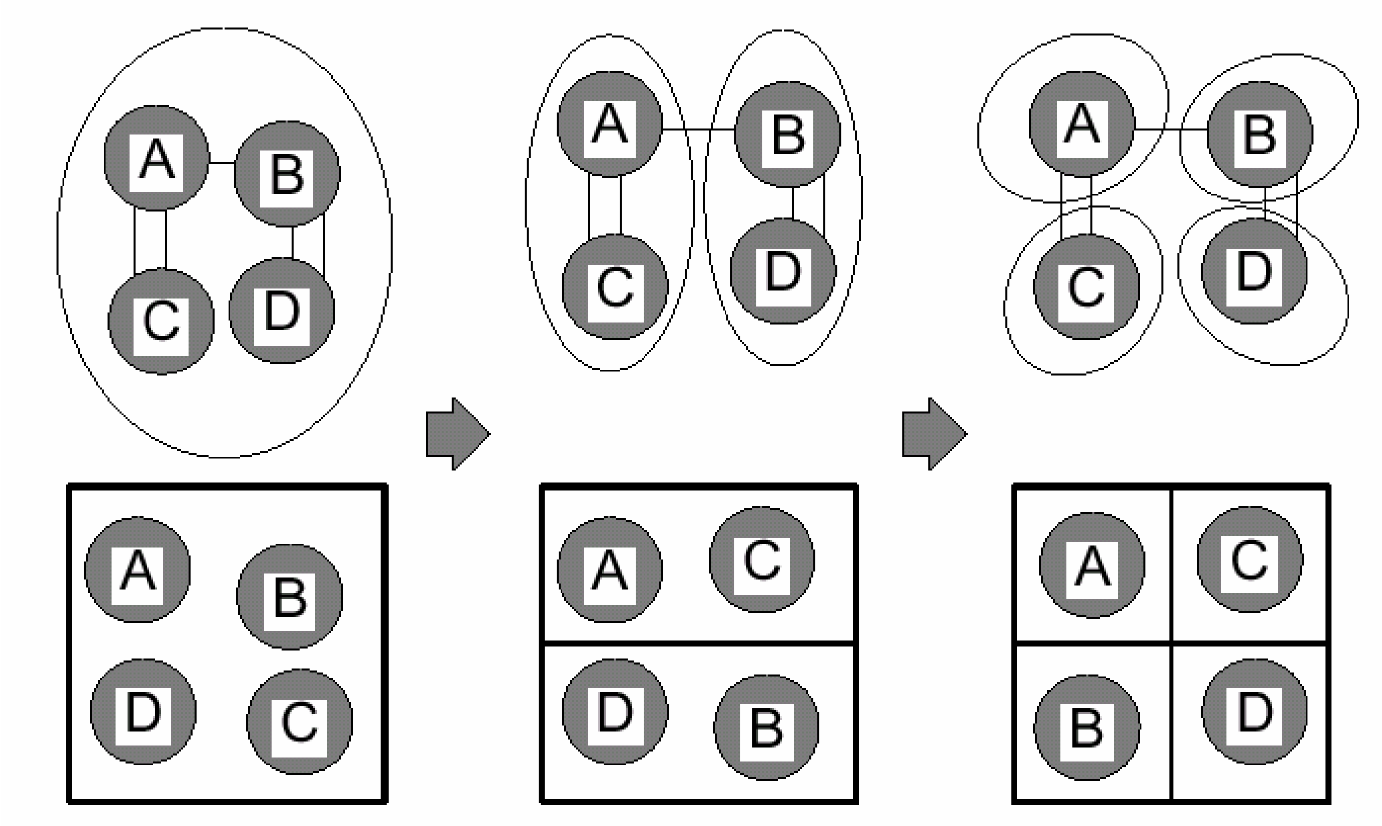
# Cluster Growth Example

- # 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 $\hat{c}_{ae} = 2, \hat{c}_{be} = \hat{c}_{eg} = 1$ ⇒ Place *a* next to *e* (say, slot 3). Continue until all cells are placed.
- Further improve the placement by swapping the gates.



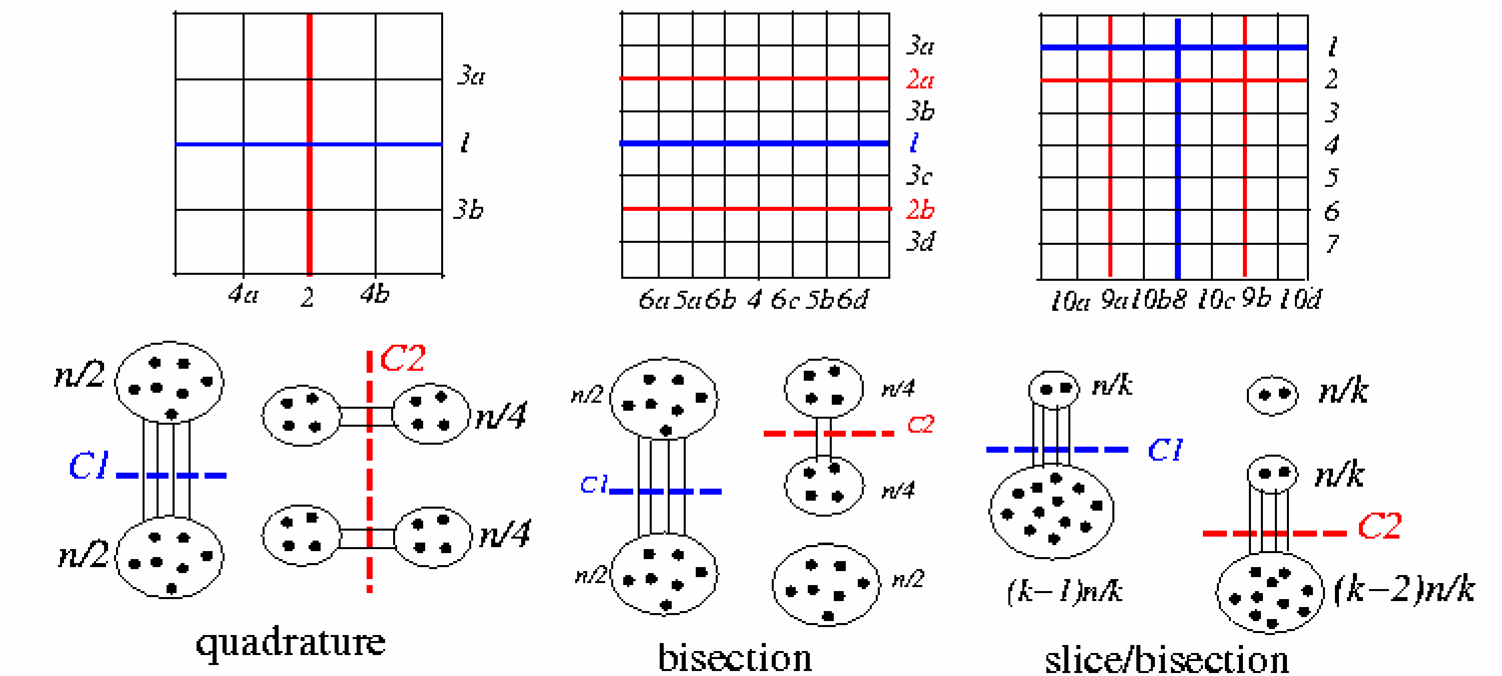
# Top-down Placement: Min Cut

- 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-77.
- **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 */

/* $n_0$: # of cells in a slot */

/* $C$: the connectivity matrix */

**begin**

**if** $(n \leq n_0)$ **then** PlaceCells($N$, $n$, $C$)

**else**

  $(N_1, N_2) \leftarrow$ CutSurface($N$);

  $(n_1, C_1), (n_2, C_2) \leftarrow$ Partition($n$, $C$);

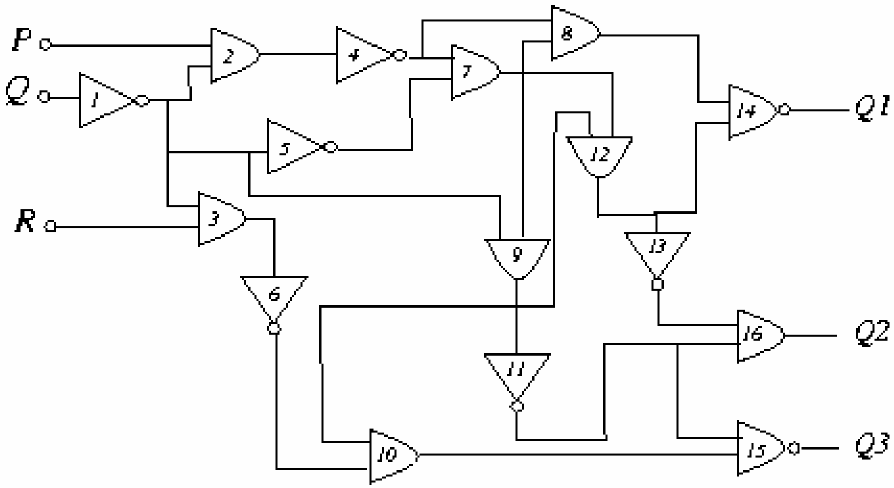
 **Call** Min_Cut_Placement($N_1$, $n_1$, $C_1$);

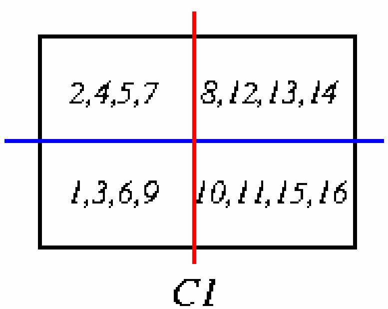
 **Call** Min_Cut_Placement($N_2$, $n_2$, $C_2$);

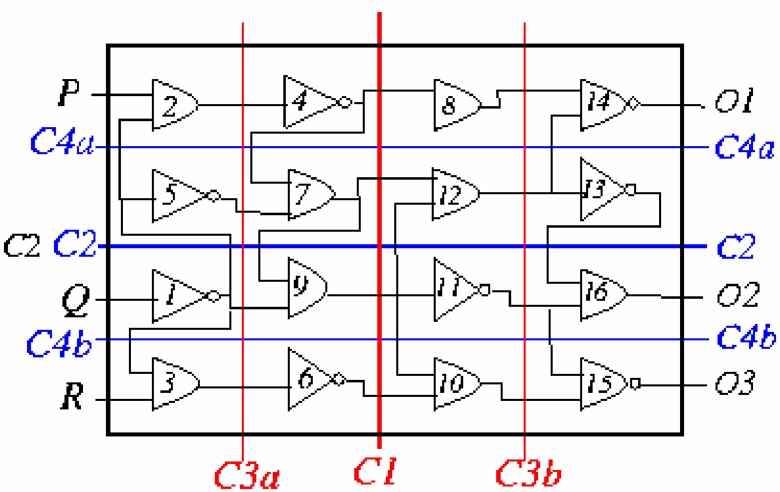
**end**

# Quadrature Placement Example

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

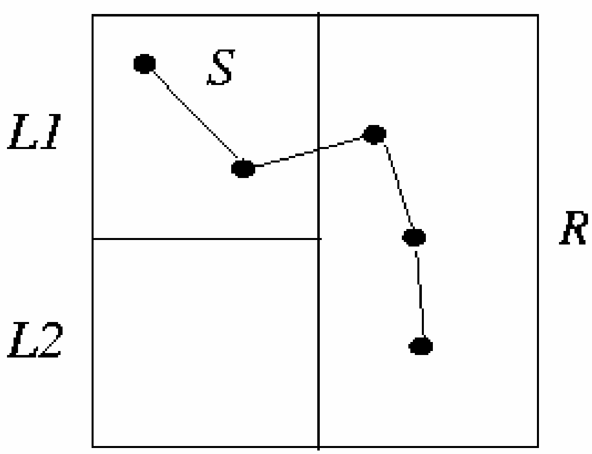


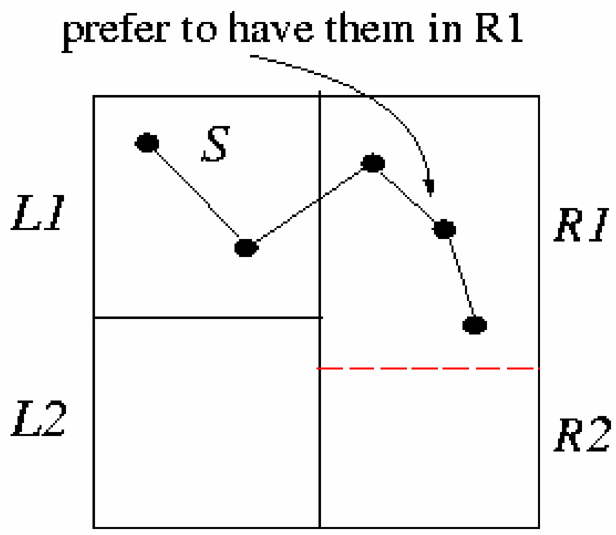



# Min-Cut Placement with Terminal Propagation

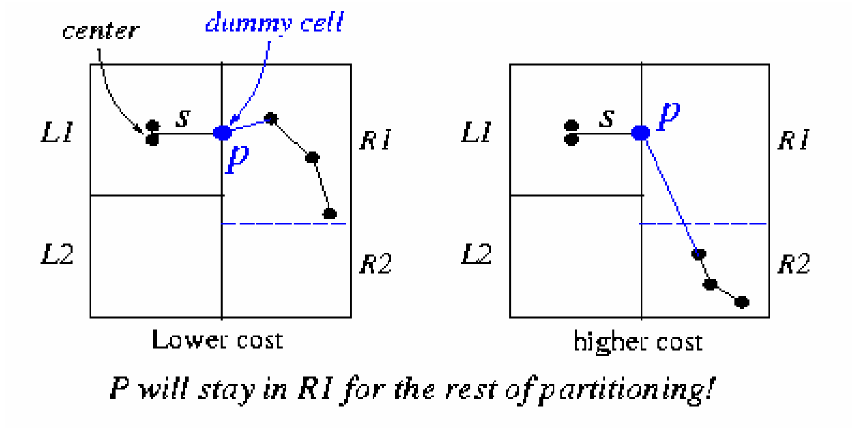
- Dunlop & Kernighan, "A procedure for placement of standard-cell VLSI circuits," *IEEE TCAD*, Jan. 1985.
- Drawback of the original min-cut placement: Does not consider the positions of terminal pins that enter a region.
  - What happens if we swap {1, 3, 6, 9} and {2, 4, 5, 7} in the previous example?



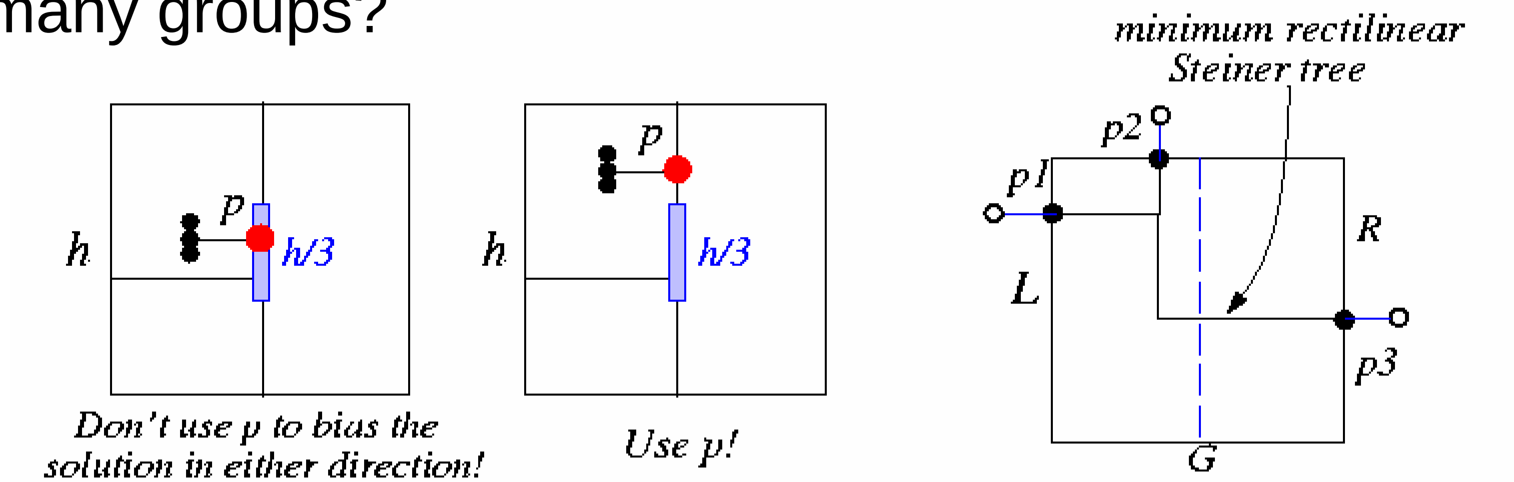



# Terminal Propagation

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

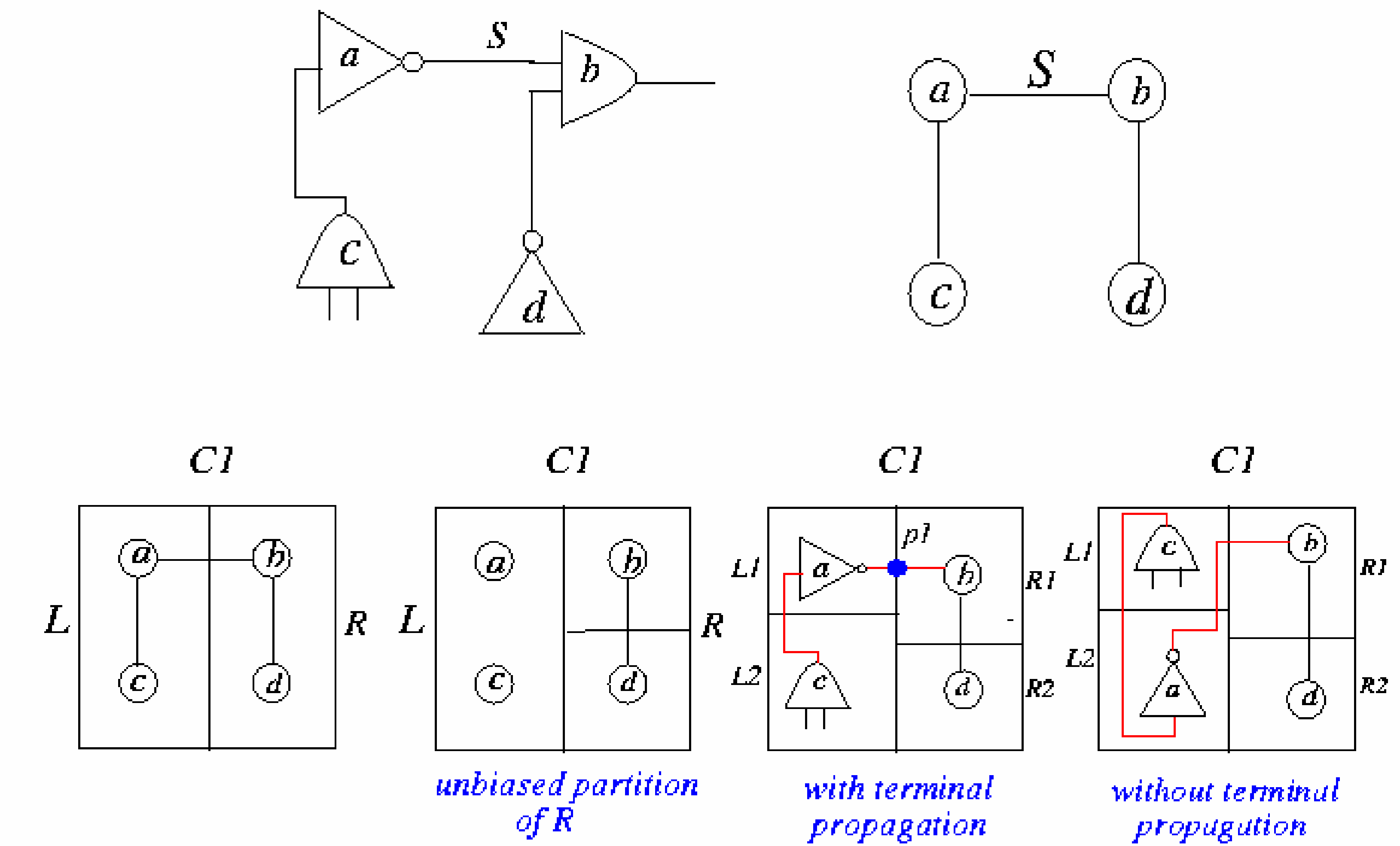



- When not to use $p$ to bias partitioning? Net $s$ has cells in many groups?



# Terminal Propagation Example

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

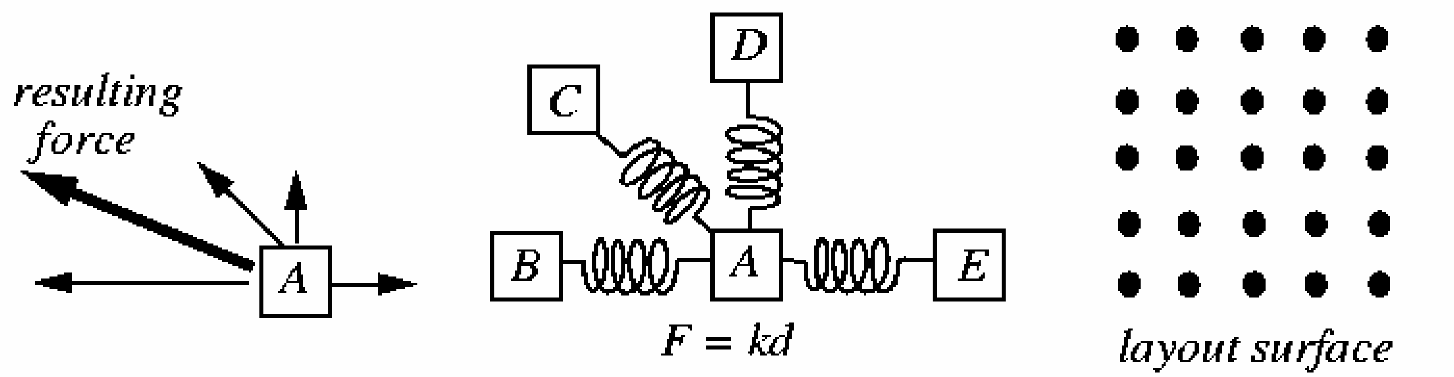


# General Procedure for Iterative Improvement

```
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.



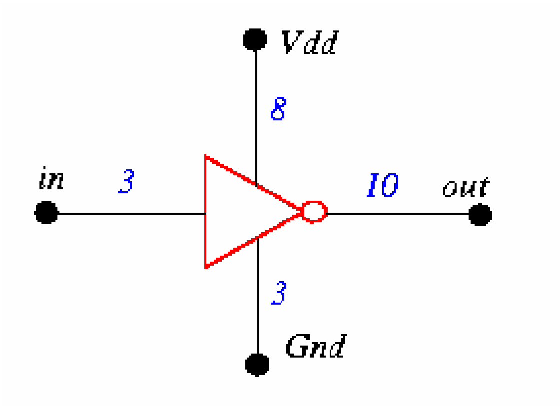
# Finding the Zero-Force Target Location

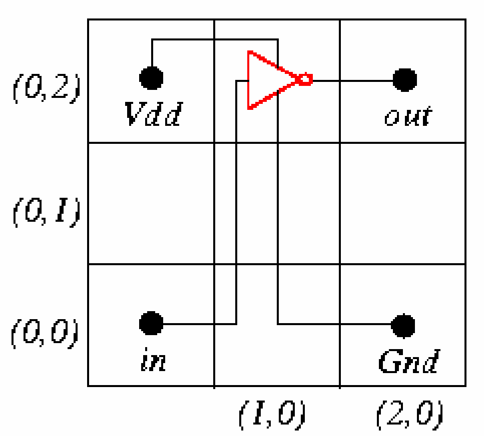
- Cell $i$ connects to several cells $j$'s at distances $d_{ij}$'s by wires of weights $w_{ij}$'s. Total force: $F_i = \sum_j w_{ij} d_{ij}$
- The zero-force target location ( $\hat{x_i}$, $\hat{y_i}$ ) can be determined by equating the $x$- and $y$-components of the forces to zero:

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

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

- In the example, $\hat{x_i} = \frac{8 \times 0 + 10 \times 2 + 3 \times 0 + 3 \times 2}{8 + 10 + 3 + 3} = 1.083$ and $\hat{y_i} = 1.50$.



# 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*.

**Algorithm: Force-Directed_Placement**

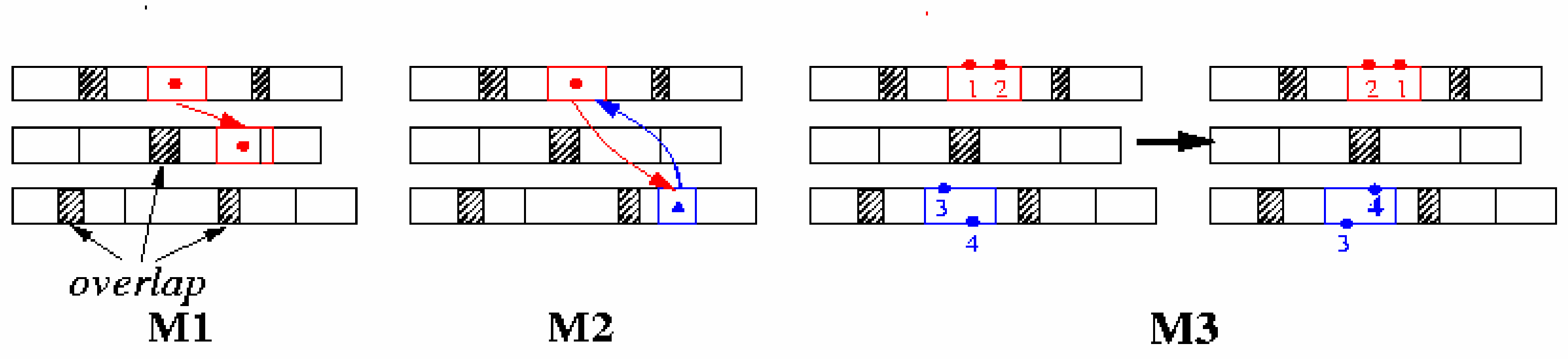
```
begin
Compute the connectivity for each cell;
Sort the cells in decreasing order of their connectivities into list L;
while (IterationCount < IterationLimit) do
    Seed ← next module from L;
    Declare the position of the seed vacant;
        while (EndRipple = FALSE) do
            Compute target location of the seed;
            case the target location
            VACANT:
                Move seed to the target location and lock;
                EndRipple ← TRUE; AbortCount ← 0;
            SAME AS PRESENT LOCATION:
                EndRipple ← TRUE; AbortCount ← 0;
            LOCKED:
                Move selected cell to the nearest vacant location;
                EndRipple ← TRUE; AbortCount ← AbortCount + 1;
                if (AbortCount > AbortLimit) then
                    Unlock all cell locations;
                    IterationCount ← IterationCount + 1;
            OCCUPIED AND NOT LOCKED:
                Select cell as the target location for next move;
                Move seed cell to target location and lock the target location;
                EndRipple ← FALSE; AbortCount ← 0;
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.

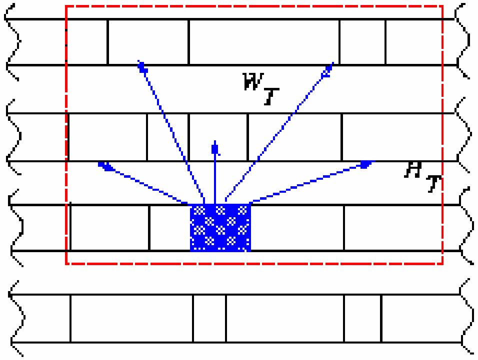
# Solution Space & Neighborhood Structure

- **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

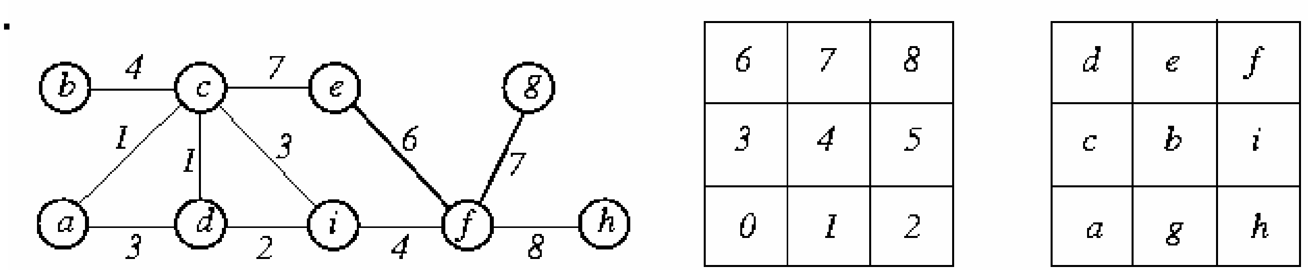
- Cost function: $C = C_1 + C_2 + C_3$.
- $C_1$: total estimated wirelength.
  - $C_1 = \sum_{i \in Nets}(\alpha_i w_i + \beta_i h_i)$
  - $\alpha_i$, $\beta_i$ are horizontal and vertical weights, respectively. ($\alpha_i$=1, $\beta_i$ =1 $\Rightarrow$ half perimeter of the bounding box of Net *i*.)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

# Annealing Schedule

- $T_k = r_k T_{k-1}$, $k = 1, 2, 3, \ldots$
- $r_k$ 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: $T < 0.1$.

# Placement by the Genetic Algorithm

- Cohoon & Paris, "Genetic placement," ICCAD-86.
- **Genetic algorithm:** A search technique that emulates the biological evolution process to find the optimum.
- Generic approaches:
  - Start with an initial set of random configurations (**population**); each individual is a string of symbol (symbol string $\leftrightarrow$ **chromosome**: a solution to the optimization problem, symbol $\leftrightarrow$ **gene**).
  - During each iteration (**generation**), the individuals are evaluated using a **fitness** measurement.
  - Two fitter individuals (**parents**) at a time are selected  to generate new solutions (**offsprings**).
  - Genetic operators: **crossover, mutation, inversion**
- In the example, string = [*aghcbidef*]; fitness value = $1/\sum_{(i,j) \in E} w_{ij}\, d_{ij} = 1/85$.



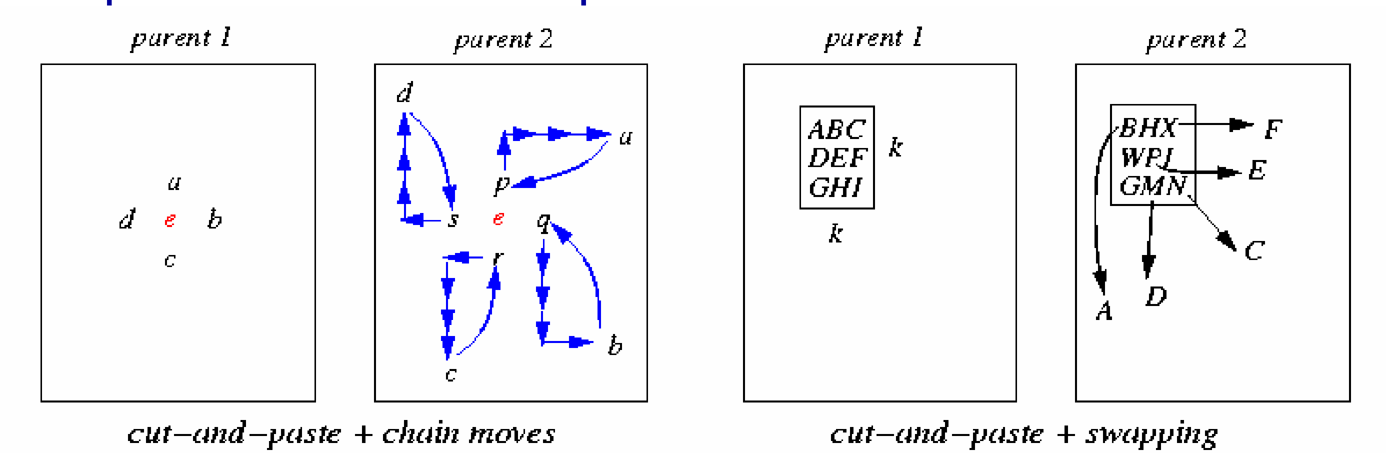

*string: aghcbidef*

# Genetic Operator: Crossover

- Main genetic operator: Operate on two individuals and generates an offspring.
  - $[bidef|aghc](\frac{1}{86}) + [bdefi|gcha](\frac{1}{110}) \rightarrow [bidefgcha](\frac{1}{63})$.
  - Need to avoid repeated symbols in the solution string!
- **Partially mapped crossover** for avoiding repeated symbols:
  - $[bidef|gcha](\frac{1}{86}) + [aghcb|idef](\frac{1}{85}) \rightarrow [bgcha|idef]$.
  - Copy *idef* to the offspring; scan [*bidef*|*gcha*] from the left, and then copy all unrepeated genes.

# Two More Crossover Operations

- Cut-and-paste + Chain moves:
  - Copy a randomly selected cell *e* and its four neighbors from parent 1 to parent 2.
  - The cells that earlier occupied the neighboring locations in parent 2 are shifted outwards.
- Cut-and-paste + Swapping
  - Copy *k* × *k* square modules from parent 1 to parent 2 (*k*: random # from a normal distribution with mean 3 and variance 1).
  - Swap cells not in both square modules.



# Genetic Operators: Mutation & Inversion

- **Mutation:** prevents loss of diversity by introducing new solutions.
  – Incremental random changes in the offspring generated by the crossover.
  – A commonly used mutation: pairwise interchange.
- **Inversion:** [*bid*|*efgch*|*a*] $\rightarrow$ [*bid*|*hcgfe*|*a*].
- Apply mutation and inversion with probability $P_\mu$ and $P_i$ respectively.

**Algorithm: Genetic_Placement**($N_p$, $N_g$, $N_o$, $P_i$, $P\mu$ )

/* $N_p$: population size; */

/* $N_g$: # of generation; */

/* $N_o$: \# of offspring; */

/* $Pi$ : inversion probability; */

/* $P\mu$ : mutation probability; */

**begin**<br>
ConstructPopulation($N_g$); /* randomly generate the initial population */<br>
**for** $j \leftarrow 1$ **to** $N_p$<br>
    Evaluate Fitness(*population*($N_p$));<br>
**for** $i \leftarrow 1$ **to** $N_g$<br>
    **for** $j \leftarrow 1$ **to** $N_o$<br>
        $(x, y) \leftarrow$ ChooseParents; /* choose parents with probability $\propto$ fitness value */<br>
        *offspring*($j$) $\leftarrow$ GenerateOffspring($x$, $y$); /* perform crossover to generate offspring */<br>
        **for** $h \leftarrow 1$ **to** $N_p$<br>
            With probability $P\mu$, apply Mutation(*population*($h$));<br>
        **for** $h \leftarrow 1$ **to** $N_p$<br>
            With probability $P_i$, apply Inversion(*population*($h$));<br>
        Evaluate Fitness(*offspring*($j$));<br>
    *population* $\leftarrow$ Select(*population*, *offspring*, $N_p$);<br>
**return** the highest scoring configuration in *population*;<br>
**end**

# Genetic Placement Experiment: GINIE

- Termination condition: no improvement in the best solution for 10,000 generations.
- Population size: 50. (Each generation: 50 unchanged throughout the process.)
- Each generation creates 12 offsprings.
- Comparisons with simulated annealing:
  - Similar quality of solutions and running time.
  - Needs more memory.