**Placement**

Readings: Chapter 4

Pick relative location for each gate

Seek to improve routeability, limit delay, reduce overall area

- NAND
- AOI
- DFF
- INV
- NOR
- DFF

**Placement Restrictions**

Rectangular chip

I/Os at periphery

1. I/O
   - NAND
   - AOI
   - I/O

2. I/O
   - DFF
   - INV
   - NOR
   - I/O

3. I/O
   - DFF
   - NOR
   - I/O

4. I/O
   - I/O
   - I/O
   - I/O
Placement Goals

Reduce Area
  Reduce height
    Fewer signals per routing channel
  Reduce width
    Fewer feedthroughs
    Balanced row width

Reduce Delay
  Reduce wirelength

Placement Cost Function - Wirelength

Most systems use Manhattan routing (North, South, East, West, no diagonals)

Wirelength estimate = 1/2*(perimeter of bounding box) = “Semi-perimeter”
Placement Cost Function - Cutsizes

Routing area estimate = row width*($\sum$ channel heights) + feedthroughs
  Problem: don’t know where a signal will be routed

Maximum vertical cutsizes estimate channel height
Maximum horizontal cutsizes estimate feedthroughs

Placement via Recursive Bipartitioning

Partition circuit in two dimensions recursively until each partition has one node
  Propagate terminals

Chip

Problems:
  Critical Paths
  Balancing row widths with variable sized cells
**Force-Directed Placement**

Signals replaced by springs
- spring constant(net) = 1/(net fanout)
- seek to minimize the forces on each node
- increase spring constant for more critical nodes

**Forces**

Spring forces
- $F_x = \text{spring constant} \times (\text{other}_x - \text{my}_x)$
- $F_y = \text{spring constant} \times (\text{other}_y - \text{my}_y)$
- Net force = $\sum \text{spring forces}$
**Force-Directed Algorithm**

While improvements are still being made
- Pick node (randomly, or node with greatest force imbalance)
- Move node to optimum location
- Relocate any node at this location
  - (swap pair, or recursively find optimums, don’t move any node twice)
- If move doesn’t result in overall reduction in force imbalance, undo full move

---

**Placement Local Minima**

Greedy placement algorithms (force-directed, recursive bipartitioning) can easily get stuck in local minima

---

Need a method that is less susceptible to local minima
**Annealing**

Annealing: Cooling hot metals to form good crystal structures
- Start at high temperatures - atoms move randomly about
- Cool at specific cooling schedule - leave enough time for atoms to attract into crystal lattice

**Simulated Annealing**

Move nodes randomly
- Initially "high temperature" - allow bad moves to happen
- Lower temperature, accepting less and less bad moves
- Slowly "cool" placement to allow good structure to form
SA Acceptance Criteria & Cooling Schedule

Compute delta = cost(old_placement) - cost(new_placement)
if (delta>=0) accept
else if (random < e^{delta/Temp}) accept, else reject /* 0<=random<=1 */

Initially temperature is very high (most bad moves accepted)
Temp slowly goes to 0, with multiple moves attempted at each temperature
Final runs with temp=0 (always reject bad moves) greedily "quench" the system

SA Cost Function

Simulated Annealing requires a cost function that captures quality of placement
Smaller cost means better placement
Multiple concerns captured in one metric

\[ \text{cost(placement)} = e \left( \sum_{\text{semi}} \text{perimeter}(i) \right) + c_i \bullet \text{critical \_ path \_ delay} \]
SA Move Function

Define the allowed transformations of the circuit

Circuits after transform should be somewhat similar (good neighborhood)
Transforms should be able to eventually reach all legal configurations

Standard moves:

Swap
Move to row (swap with empty destination)
I/Os only allowed at sides of chip. No moves swap I/Os with cells.

Optimization - windowing
Only allow moves in a small neighborhood (finds more good moves)

Simulated Annealing Algorithm

Create initial placement
old_cost = cost(placement);
for (temp = max_temp; temp >= min_temp; temp = next_temp) {
    for (iteration = 0; iteration < max_iteration; iteration++) {
        make_move();
        new_cost = cost(placement);
        if (old_cost < new_cost)
            if (random >= exp((old_cost - new_cost)/temperature)
                undo_move();
    }
}
**Practical Issues with Simulated Annealing**

Cost functions must be carefully developed, “fractal” & smooth

Example: if (critical_path > required) cost += 1000
vs. cost += max((critical_path - required), 0)

Cost functions must be FAST - called millions of times
Best: easy to calculate deltas, such as just updating length of changed nets
if updating nets connected to swapped nodes, beware of shared nets

---

**Practical Issues with Simulated Annealing**

Balancing multiple objectives complex, requires LOTS of testing
Area efficient vs. performance efficient implementations

Cooling schedule also takes LOTS of testing. Some suggestions (VPR-ish):
1.) Start at a temperature 20*measured standard deviation of random swaps
2.) Make say 10xNumNodes moves per temperature
3.) Decrease temp based on acceptance rate $\alpha$ of moves at temperature
   
   $\alpha > .96$ - $.5x$
   $ .96 < \alpha < .80$ - $.9x$
   $ .80 < \alpha < .15$ - $.95x$
   $ .15 < \alpha$ - $.8x$

4.) Stop when it no longer accepts any bad moves over a couple iterations.
Simulated Annealing Example

You are teaching a group project class for $N \geq 14$ students. You ask them to tell you people they would love to work with, and people they would hate. Have an annealer form teams of 4-5 students each.

Initial group formation:

Move function:

Cost Function:

---

Genetic Algorithm

“Breed” configurations, with survival of the fittest

Start with sample population

Evaluate with a cost function, eliminating less fit configurations

Generate new configurations from surviving configurations, and repeat
**Genetic String**

Express a placement as a string of characters, numbers, etc.

Node Ids in string, with position indicating location

\{ B F D G E H A I C \}

---

**Genetic Algorithm Operators**

**Crossover**

Take two strings, split at random point, and join left side of A with right of B

Parent 1: Parent 2:

\{BFDGE | HAIC\} X \{GDBFE | CHIA\} → \{ \}

Note: each node should appear in the string EXACTLY once

For any repeats in right side, replace with a missing cell

Parent 1: Parent 2:

\{BFDGE | HAIC\} X \{GCAFE | DHIB\} → \{BFDGE \}
Genetic Algorithm Operators (cont.)

Mutation
Randomly swap two cells (similar to simulated annealing move function)

Child:
{ E D I G H F B C A } → { }

Inversion
Randomly select subsequence of string and reverse it

Child:
{ G | A F E D B H | C I } → { }

Genetic Algorithm Algorithm

Create initial population of size generation_size;
for (generations = 0; generations < requested_generations; generations++) {
  for (I = 0; I < generation_size; I++) {
    choose parents randomly, weighted by population member’s fitness;
    offspring[I] = crossover(parent1, parent2);
  }
  randomly apply mutations to offspring, with probability prob_mutation;
  randomly apply inversions to offspring, with probability prob_inversion;
  select new population from current population & offspring, based on fitness;
}
return most fit population member;