(*Lec 13*) Placement & Partitioning: Part II

- What you know
  - That there are 3 big placement styles: iterative, recursive, direct
  - Placement via iterative improvement using simulated annealing
- What you don't know
  - The other 2: recursive and direct placement
  - The fact that they have many points of great similarity
  - Real algorithms for doing recursive or direct placement
  - Recursive: 2 most famous heuristics: K&L, F&M
- What's later still (part III)
  - Direct: classical quadratic formulation + Tsay-style legalization

Copyright Notice

Where Are We?

- Physical design--placement via recursive, direct methods

	M	T	W	Th	F	
Aug	27	28	29	30	31	1
Sep	3	4	5	6	7	2
	10	11	12	13	14	3
	17	18	19	20	21	4
	24	25	26	27	28	5
Oct	1	2	3	4	5	6
	8	9	10	11	12	7
Midsem	15	16	17	18	19	8
break	22	23	24	25	26	9
	29	30	31	1	2	10
Nov	5	6	7	8	9	11
	12	13	14	15	16	12
Thnxgive	19	20	21	22	23	13
	26	27	28	29	30	14
Dec	3	4	5	6	7	15
	10	11	12	13	14	16

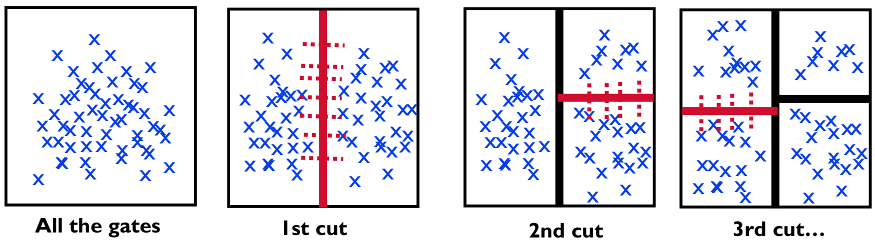
Introduction
Advanced Boolean algebra
JAVA Review
Formal verification
2-Level logic synthesis
Multi-level logic synthesis
Technology mapping
Placement
Routing
Static timing analysis
Electrical timing analysis
Geometric data structs & apps

Deadlines

- Project2
  - November 8 (next THU)
  - Email us your website URL
- Homework 4 (still TBD on the .pdf on website)
  - Propose: Nov 15 (Thu in class)
- Paper #2
  - Propose Nov 13 (Tue in class)
  - Decided for this one to still be written – we'll do PPT for Paper3
- What's left in 760
  - HW5 (layout, timing), HW6 (geometric data structs, short)
  - Project 3 (a REAL floorplanner – pay attn to last prob on HW4!)
  - Paper 3 – something about layout

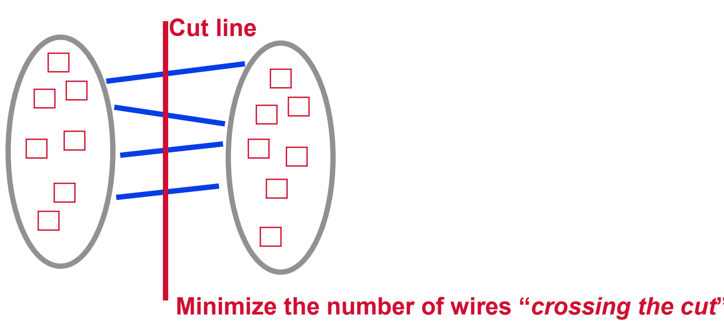
Recursive Placement: Min-Cut

- Basic idea
  - The "*simplest*" placement decision you can make is: cut the chip into 2 pieces, partition gates over the 2 sides, minimize wiring in between
  - Can continue doing this recursively with results of each partition step



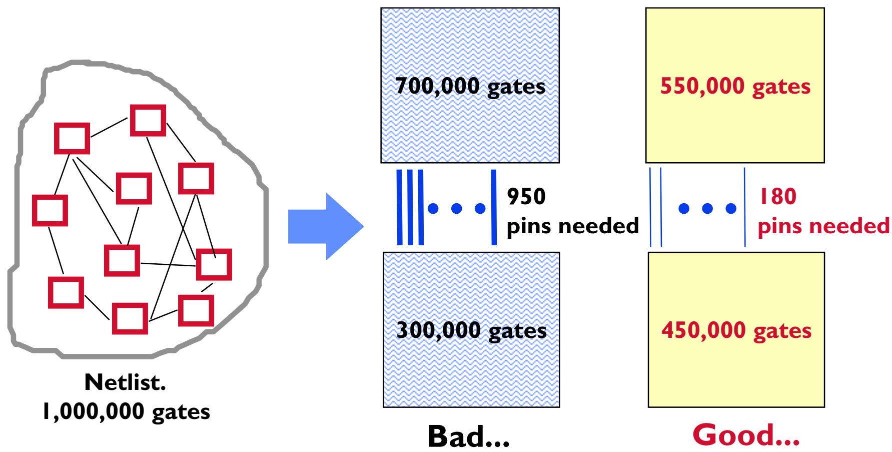
Abstract Problem To Solve: *Partitioning*

- *Partition* the gates between 2 regions so that
  - Capacity (# of gates allowed) on each side is not exceeded
  - Cost (i.e., the number) of wires across the cut is minimized
- Classical problem is *bipartitioning*

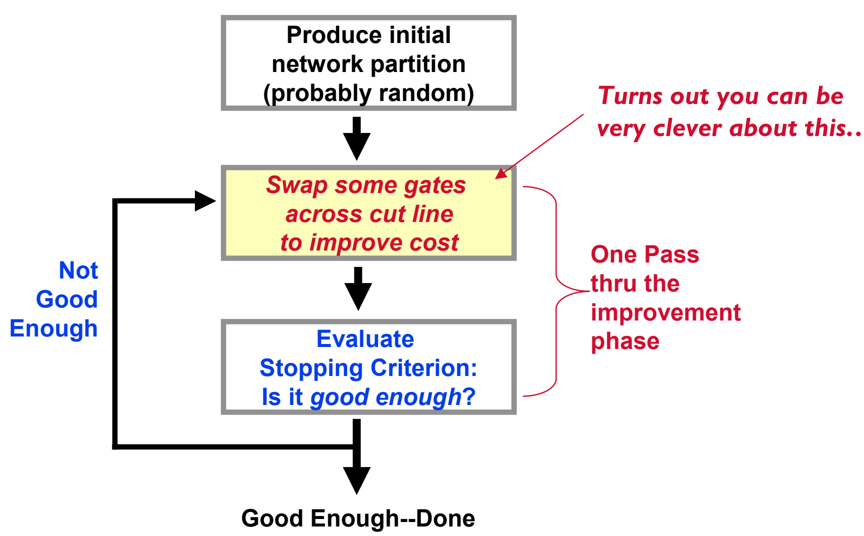


Note: Its Very Easy to Do this Poorly

- Example capacity constraint: <= 600K gates per side
  - Capacity constraint: <= 500K gates/side
  - IO constraint: <=200 "pins" == connections on each side



Solution Style: Iterative Improvement

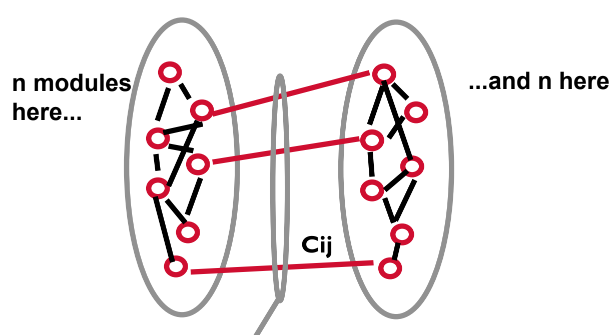


Bipartitioning Algorithms

- **Kernighan-Lin**
  - **Most famous partitioning heuristic, core of all others**
  - **Right idea -- but too slow**
  - **Aside: Yes, *that* Kernighan, of C-UNIX fame; this was his PhD thesis...**
- **Fiduccia & Mattheyses**
  - **Start with Kernighan-Lin (K-L), but made it fast**
  - **Clever data structure, and some very minor tweaks make the algorithm linear O(n) for n gates for one improvement pass**
  - **Core of most serious partitioners today**
  - **Aside: *fi-DOO-chi-uh* and *muh-TEE-sis*, 2 guys at GE Labs in early 1980s**

K&L Partitioning Model

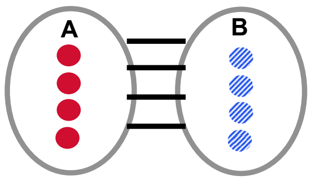
- **Minimize total *sum of costs* of nets across cut**
  - **Each net has a cost cij, and we assume 2n gates, to be partitioned equally into n gates on each side of the cut**



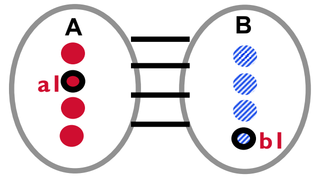
$$T = \sum_{\text{Cut nets}} c_{ij} = \text{minimum}$$

K&L Improvement Procedure

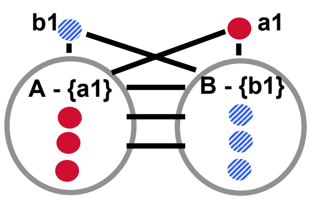
0. Start with any (random) partition



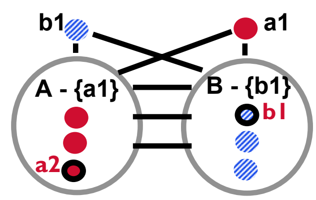
1. Identify a1• in A, b1• in B so that swapping a1, b1 gives *maximum gain*



2. Swap a1 and b1 and *lock* them in place--they cannot swap again



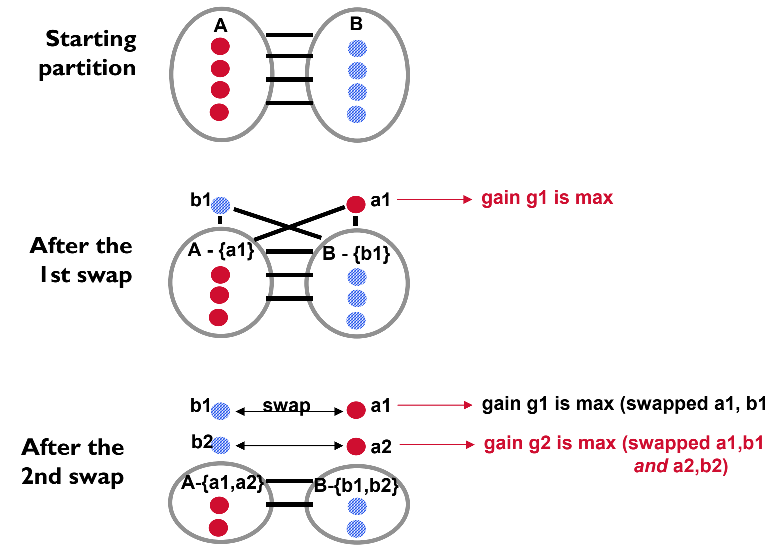
3. Continue: Identify a2,• b2• so that swapping a2, b2 gives *max gain…*



K&L: Critical Ideas

- **Gain**
  - Gain is the change in cost that results from swapping one gate in the A-side with one gate in the B-side.
  - Compute it as $\underbrace{\Sigma_{cut}\ cij}_{\text{After swap}} - \underbrace{\Sigma_{cut}\ cij}_{\text{Before swap}}$
- **Be *greedy*, but *persevere***
  - Always do the very best next swap (biggest Gain number) you can
  - Do this swap even if it’s negative, ie, even if it makes the partition worse
  - So, the swap to pick is the strictly biggest gain:
    - Biggest positive gain (makes cut better)
    - Smallest (closest to 0) negative gain (makes cut the least worse)
- **Do all n swaps**
  - …ie, swap until nothing is left to swap.

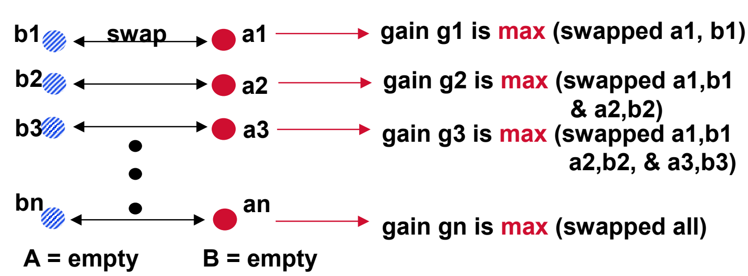
K&L Example



Continue…

K&L Example

- Keep going until *all* n pairs swapped

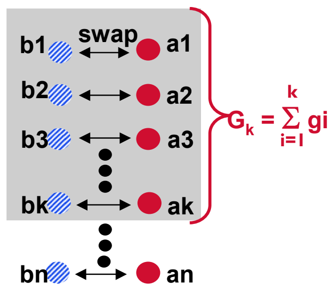


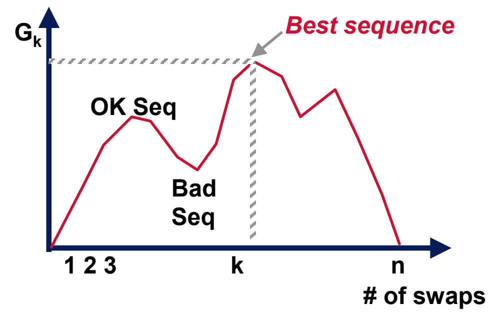
- New problem: How *many* swaps should we keep?
  - Makes no sense to do them all--this just *reverses* A and B!
  - What the **best** set of all these swaps the maximizes overall gain?
  - This is the next really clever part of K&L…

K&L: Picking Swap Sequence

Important result

- Gain from doing swaps 1,2, ... k in sequence is $G_k = \sum_{i=1}^{k} gi$
- Gk is *NOT* monotonic function of k--it has hills & valleys





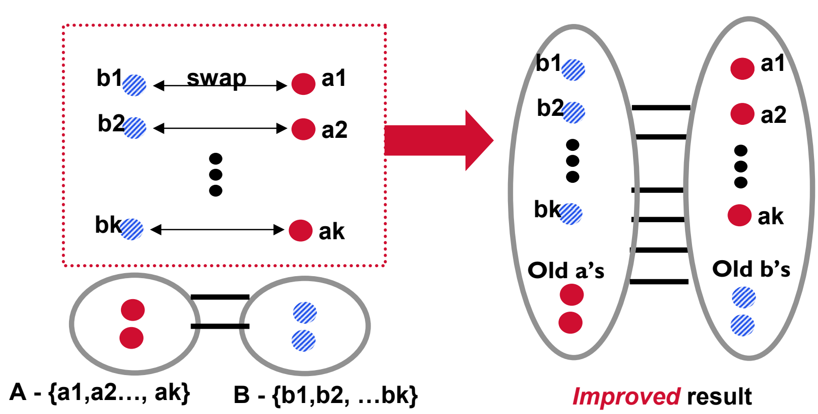
Interpretation:

- Good to do some *locally bad* swaps, to get to *globally better* answer
- This is a particular form of "hill-climbing"

K&L: Doing the Swaps

Interpretation:

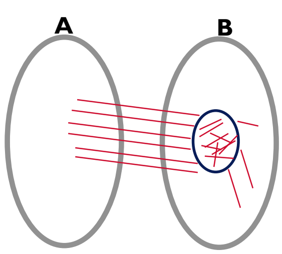
- We will commit to do these k swaps, since they **MAXIMIZE** gain



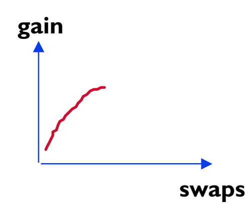
K&L Procedure

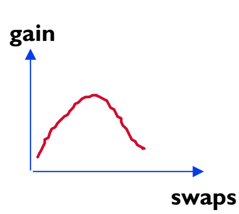
- So, what do we actually do?
  - Evaluate **all** n swaps, maximizing the gain on ***each*** swap
  - (Remember: gain may be **negative** on some swaps--this is OK)
  - After all n swaps done, find k that gives **best** $G_k = \sum_{i=1}^{k} gi$
  - **Commit** the swaps a1,b1 & a2,b2 & a3,b3 … & ak,bk
  - This completes one "improvement pass" of K&L
- K&L overall
  - Just keep doing improvement passes until it's not getting better

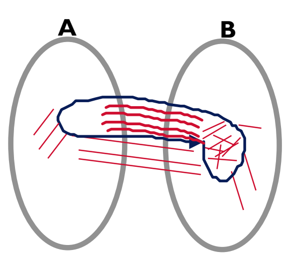
K&L: Why It Works



Dense, interconnected cluster of gates on the B side. *Really* wants to be on the A side…

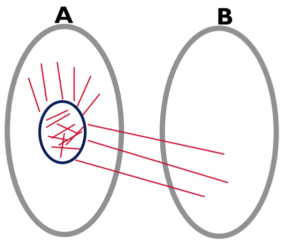




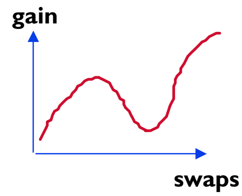


But as you start swapping individual gates over to B, intermediate results look *really* bad -- you suddenly are splitting that dense cluster *across* partitions…

K&L: Why It Works



But, eventually the whole cluster ends up on the A side, and overall this is the better solution for partition



- Hill climbing is *critical*
  - By looking at all possible max gain swaps...
  - ...and then picking the best sequence of max gain swaps...
  - ..while tolerating individual swaps that are bad (negative)
  - ...you can get a much better overall solution

K&L: Quality & Complexity

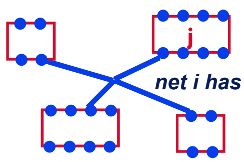
- How well does K&L work?
  - Fabulous. Amazingly good (with respect to what came before it)
  - The "be greedy but persevere" idea is a very famous idea
- How fast does K&L run
  - Not.
  - Why? How do we select the "swap with best gain"?
  - It's possible to tag every gate with some partial numerical info that can be used to calculate a single gain for a single swap, quickly
  - Still need to find the best: intrinsically requires sorting the gates by these numerical tags
  - Sort of n points is O(n logn), but we do this n times, as:

    $$\underbrace{(n \log n)}_{\text{1st pair}} + \underbrace{(n-1 \log n-1)}_{\text{2nd pair}} + \underbrace{(n-2 \log n-2)}_{\text{3rd pair}} + \dots + \underbrace{(2 \log 2)}_{\text{next to last}} + \underbrace{(1 \log 1)}_{\text{last pair}}$$

  - This sum is $O(n^2 \log n)$. Ouch, for big n.

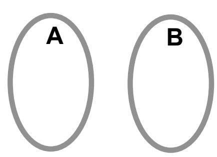
Improving K&L: Fiduccia & Mattheyses

- **F&M: Keeps good parts of K&L**
  - **Iterative improvement, with multiple passes, as before**
  - **Hill-climbing, selecting best swap sequence**
- **F&M: New ideas**
  - **Better data structure for finding what to swap for max gain**
  - **Clever accounting for what needs to get updated after cell moves**
  - **Gates not just points anymore: they have *size***



F&M Basics

- **New constraint: Balance criterion**
  - **Each cell i now has size s(i), so just having same # of cells in each partition is not enough, we need better defn of "balance"**

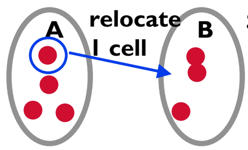


size of A = $\Sigma_{i \varepsilon A}$ size(i) = |A|

size of B = $\Sigma_{i \varepsilon B}$ size(i) = |B|

- **Balance criterion: want |A| / (|A| + |B|) ≈ r for some 0 <= r <= 1**
- **Ideally want r around 1/2, but doesn't have to be exactly 1/2**
- **Typically, allow r to be in a range, for example: r in [0.3, 0.7]**
- **User gets to pick this range of r**

F&M Procedure: Skeleton



gain = reduction in # nets cut by partition

Choose a cell to relocate across partition
that gives max gain,
that doesn't mess up balance too much

Swap it, but then *lock* this cell:
it doesn't move again

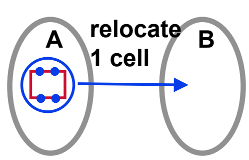
Update gains of other affected cells

If more free cells that won't mess up balance
go choose another cell to move
else quit -- this is 1 complete pass

- **Hard stuff:** ***Choose best cell? Update gains?***

F&M: Finding the Move of Max Gain

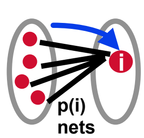
- Observation



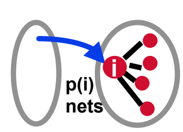
Cell i has p(i) pins, it can attach to at most p(i) distinct nets

If you relocate cell i, the *most* you can affect the gain is p(i)

-p(i) <= gain <= +p(i)



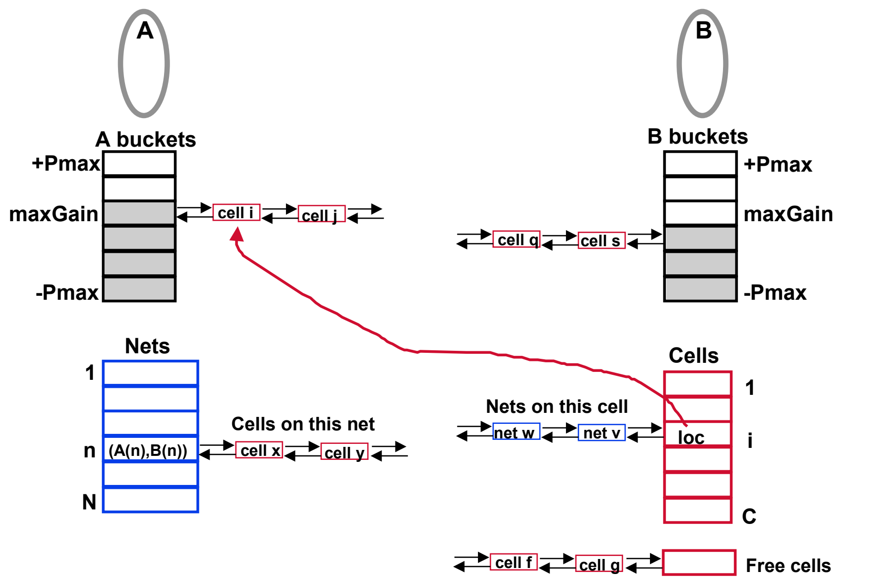
Worst-case
gain = -p(i)



Best-case
gain = +p(i)

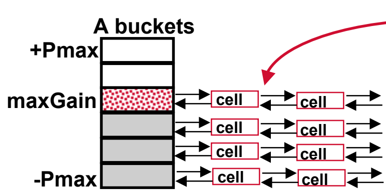
- We can generalize this
  - Overall, the gain from any move is bounded by $|\max_i \{p(i)\}|$
  - Suggests a data structure to exploit this...

F&M: Bucket Sort Data Structures



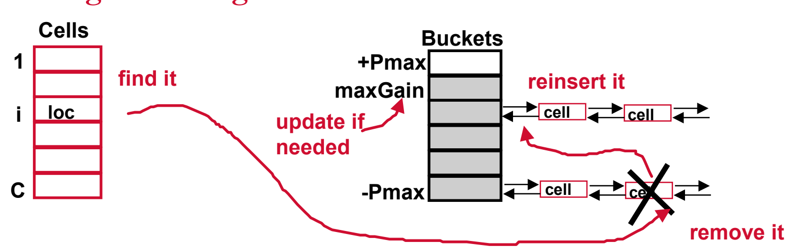
F&M: Finding Move for Max Gain

- To choose cell to move for max gain



Keep maxGain pointing at cells which will give max gain; can just grab a cell here

- To update if gains change



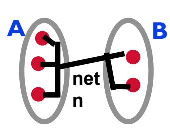
F&M Bucket Data Structure

- How much time to find max gain move?
  - *Constant* time, O(1), just look at maxGain slot in buckets
  - Also, to maintain maxGain in constant time, assuming you have to know where all the cells are; again just some pointer hacking
  - To move n cells, F&M is O(n), K&L was O(n^2 log n)
  - *Big improvement!*

- What's left?
  - How to *update* all gains on all affected cells when we move one cell ?
  - K&L assumed this was O(n) for a single swap (you might to update all the cells), so was O(n^2) to do all n swaps
  - How do we improve on this?

F&M Improvements: Net Criticality

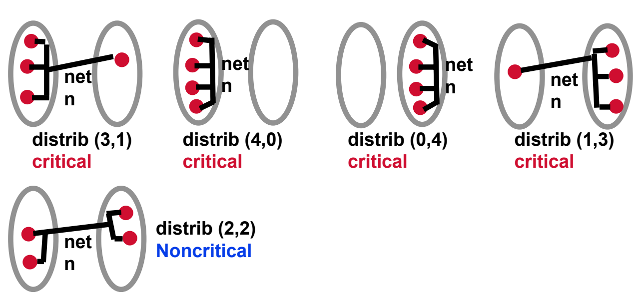
- Defn: *Distribution of a net n*



Distribution of n = (A(n), B(n))
= (# cells of n in A, # of cells of n in B)
Example at left, distrib = (3,2)

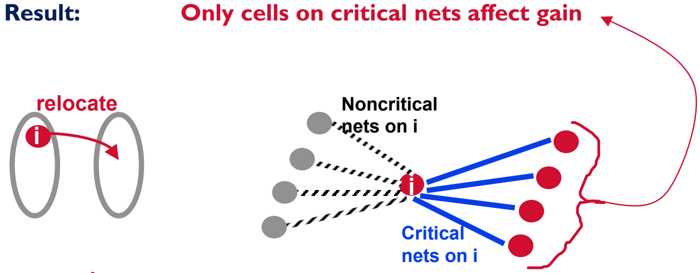
- Defn: *Critical net*
  - Net n with (A(n), B(n)) critical if A(n) or B(n) = 0 or = 1

Examples
4 pt net



F&M Improvements: Net Criticality

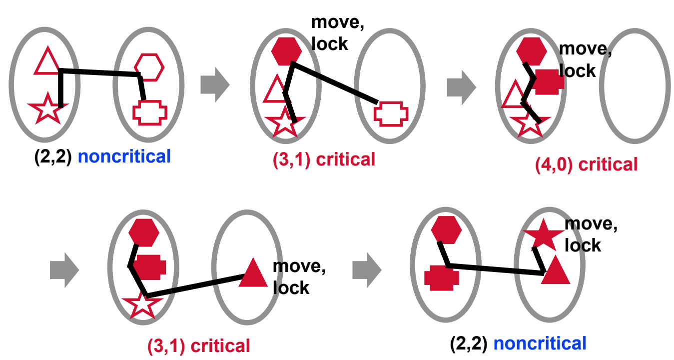
- Critical nets -- *who cares?*
  - Observation 1: if you move 1 cell on critical net, *you change the gain!*
  - Observation 2: if net *not* critical, no matter what cell moves, *no change*
  - Result: Only cells on critical nets affect gain



- Minor point
  - Need to look at nets that were critical *before* you moved cell(i)...
  - ...and nets that go critical *after* you move cell(i)
  - Just a little more bookkeeping

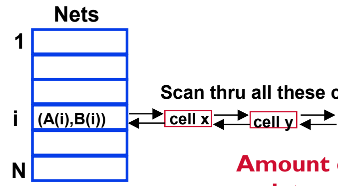
F&M: Using Criticality

- An important side effect from *locking* cells after move
  - *Nets can't stay critical forever!*
  - Eventually, cells on a net *freeze* their positions, so net can't contribute to further gain changes
  - Example



F&M: Net Updates

- **Defn: *Net Update***
  - Net update is the process of scanning *all* the cells on net i to see if they are critical (ie, are they connected to any other *critical* nets?)



Amount of "work" to do one of these update scans is just proportional to how many cells are attached to this net

- **Key result**
  - You never have to do more than a *constant* number of updates on any net in one improvement pass of F&M. Constant = 4, it turns out
  - Little nets (2,3 cells) after few moves: all their cells *freeze*
  - Big nets (>=4 cells) after few moves: they freeze to be *noncritical*

F&M: Complexity (Informal)

- **How much work to do 1 improvement pass?**
  - Recall, 1 pass means start with all cells free, do max-gain moves until no cells are free--they are all locked in one side of partition
  - We know that updating maxGain is O(1)
  - What about updating all cell gains on affected cells after a move?

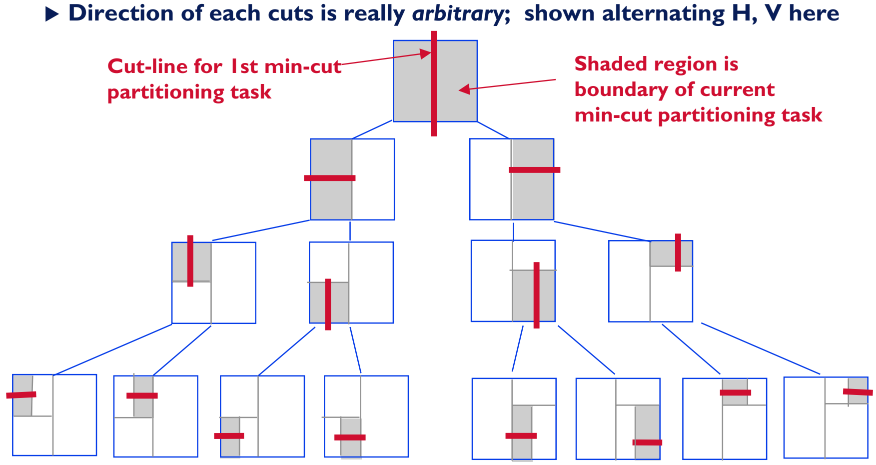
- **Look at nets...**
  - Total work = $\sum_{\text{all nets } i}$ (work per net i)
    - = $\sum_{\text{all nets } i}$ (work / update of net i X # of updates on i)
    - <= $\sum_{\text{all nets } i}$ (work / update of net i X constant)
    - <= constant X $\sum_{\text{all nets } i}$ (work / update of net i)
    - <= constant X $\sum_{\text{all nets } i}$ (#cells on net i)
    - <= constant X (total number of cell pins in entire netlist)
    - <= constant' X (#cells in the netlist)
    - = linear in problem size, O(n) (woohoo!)

F&M: Summary

- Complexity of 1 pass is *linear* in problem size
  - A very important and practical result
- Impact
  - Everybody liked results K&L was capable of getting
  - F&M preserves the overall improvement strategy, but makes it extremely fast, linear in problem size
  - Can attack very large netlists with F&M, ~1M cells
  - This is now one of the ***default, standard*** ways to do big partitioning probs
- How about today…?
  - Additional set of ideas improves this further, uses some hierarchical clustering, and some other clever stuff.
  - Best tool around today is called " hMetis", from U Minnesota. You can get it from their web page; like CUDD, widely used, easy to use.

Back to Recursive Placement via Min-Cut

- Think of repeated partitioning process as a "hierarchy" of cuts
  - Direction of each cuts is really *arbitrary*; shown alternating H, V here



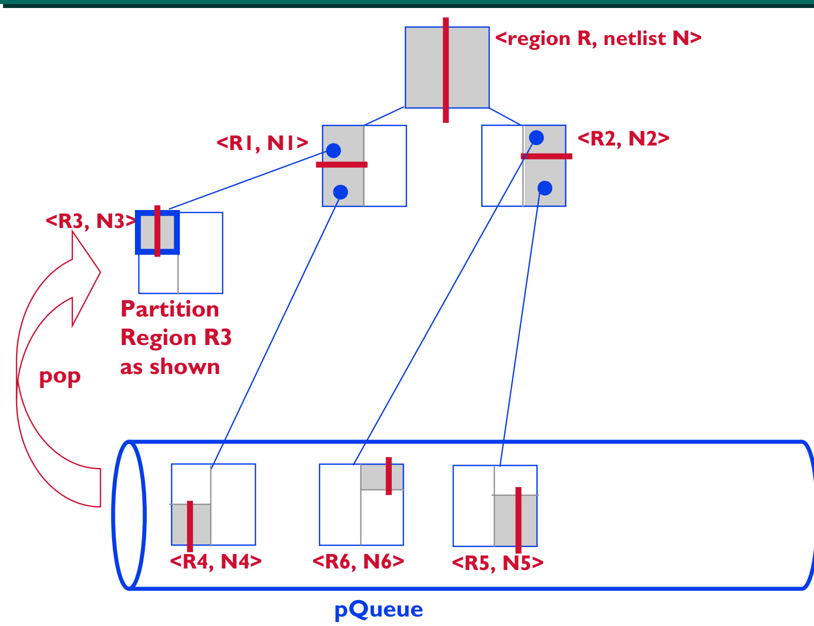
Can keep cutting until just 1 gate/region, or can quit with <=K gates/region

Back to Recursive Partitioning

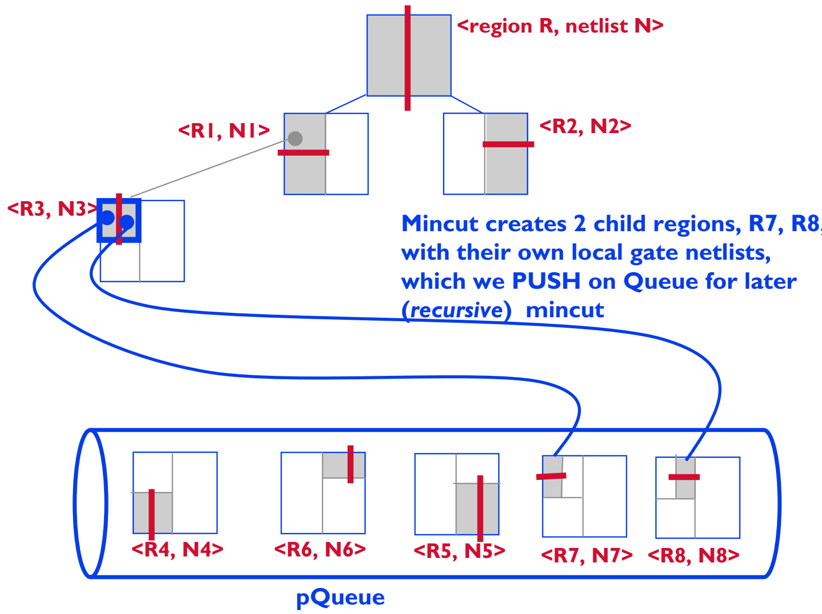
- How do people actually apply partitioning to placement?
  - Start: initial netlist of gates + wires;
    Let pQueue be a queue of regions on chip (from cut hierarchy) that
    we still need to partition;
    pQueue = <whole chip, entire netlist>;

```
while( pQueue not empty ) {
   <R, N> = pop region and netlist for gates in this region from pQueue;
   choose a cut direction for region: horizontal or vertical;
   decide the balance criterion you want to hit (ie, look where you
     think you want to put the cutline on your placement grid);
   execute partition algorithm on <R,N> to yield 2 new partitions,
     <A,Na>, <B,Nb>
   if(region A of <A, Na> is big enough to keep cutting)
      push <A, Na> onto pQueue;
   if(region B of <B, Nb> is big enough to keep cutting)
      push <B, Nb> onto pQueue;
}
```


Recursive Cutting Example



Recursive Cutting Example

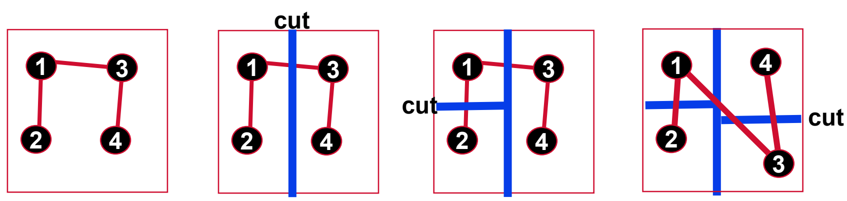


Congestion vs Wirelength

- Mincut optimizes congestion
  - It tries directly to minimize number (or sum of costs on) wires across all cuts as we recursively go down the cut "hierarchy"
  - This "tends" to minimize wirelength…
- …but, it doesn't really guarantee to minimize it
  - There are some particular problem cases that mincut can create
  - Luckily, there are also some decent fixes
- We'll look at just one: Terminal Propagation
  - By Dunlop & Kernighan

Partition Optimizations

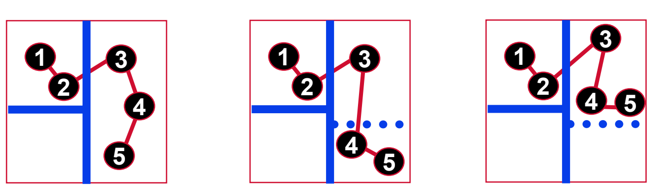
- **Terminal propagation**
  - **Now a standard technique, introduced by Dunlop and Kernighan of Bell Labs, for use with Kernighan-Lin partitioning improvement**
  - **Idea: enforce some “coupling” between objects that are connected, but which have ended up in *different* regions after a hierarchy of slicing cuts**



Oops! Cut was good, but global wiring was ignored

Terminal Propagation

- **Basic idea**
  - **Represent, at least crudely, what’s going on in the *other* partitions while you do each cut**
  - **Need somehow to create/represent a view of how global wires are evolving across cuts**

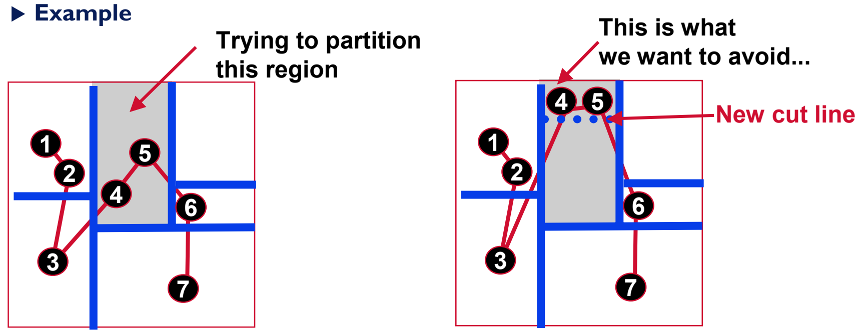


Bad partition, it lengthens this already long global wire

Better partition, it keeps modules on global wire more local

Terminal Propagation

- Mechanics
  - You try to build a crude "geometric wire model" for the wires that are going across the other cuts, and use this to bias your current partitioning task to minimize spreading connected objects across the current cut
  - Example

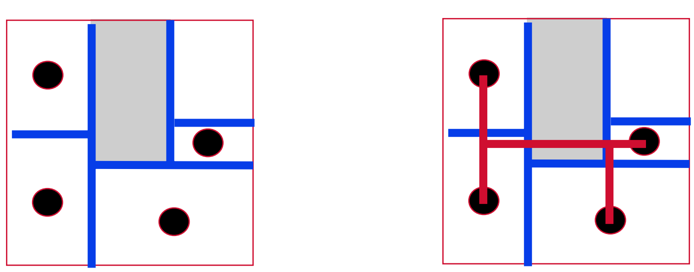


Terminal Propagation: Mechanics

- Dunlop & Kernighan used a simple *Steiner* wire model

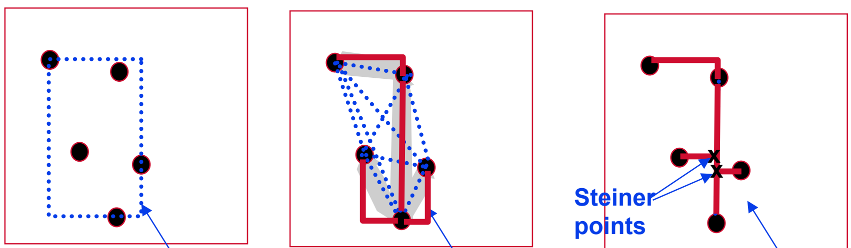
Ignore cells actually *in* the region you are partitioning, and lump all other cells in other regions at the geometric *center* of their respective region

Quickly build a decent *Steiner tree* wiring that connects all these center pts



Aside: Steiner Wire Models

- **Recall: many wire estimation models for a given set of points**
  - Idea is to estimate the length or shape of the wire, without having to actually call an expensive detailed router to route it



Half-perimeter of the the smallest bounding box is most common scalar estimator

Minimum Manhattan spanning tree is very practical. Can do this in $O(n^2)$ for n points

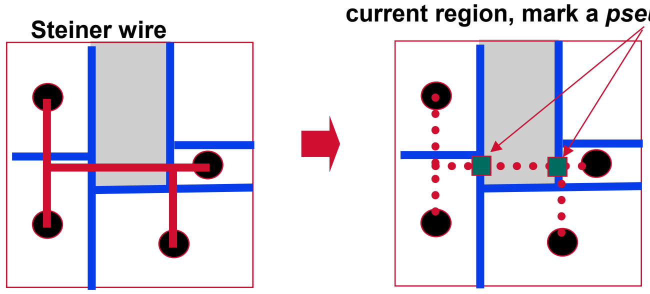
Minimum Steiner tree is the best you can get, but very expensive to approximate

Back to Terminal Propagation

- **Dunlop & Kernighan used a *really simple* Steiner heuristic**
  - Idea is to get a rough idea of "where" the global portion of wires connected to objects inside a "live" partition wants to go

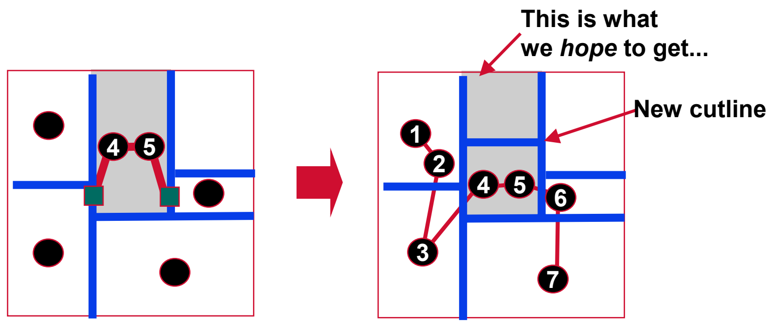
Now, where the wire crosses the boundary of the current region, mark a *pseudo pin*

Steiner wire



Terminal Propagation

- Use these pseudopins to "bias" the current partition
  - ie, model them as fixed, unswappable modules on each side of the cut, with *high-weight connections* to them
  - Use this to alter swapping cost function to favor swaps that keep modules *close* to their pseudopins



Recursive Min-cut Placement: Summary

- Relies on good bipartitioning algorithms for netlists
  - Cut the netlist into 2 parts, minimizing (number or cost of) wires crossing the cut between the 2 parts
  - K&L was the first good quality, practical partitioning algorithm
  - F&M made it fast for large-scale IC designs

- ASIC placement
  - Min-cut optimizes congestion *directly* (wires crossing cuts)
  - Min-cut only *indirectly* optimizes netlength
  - Tricks for introducing some notions of "net length" into min-cut placer
  - Practical for > 1M gate netlists, very fast, some recent innovations based on smart clustering (hMETIS) improve it over F&M
  - Usage on the rise in recent high-end placers for huge ASICs