analysis and synthesis of electronic circuits
traditional task decomposition

- placement
  - partitioning
    - dividing the netlist into smaller parts
      controlling the "sizes" of the parts and
      the number of interconnections between them
  - floorplanning
    - determining the relative positions of the parts
      and their target shapes
  - detailed placement
    - manipulating the position and orientation
      of geometrically fixed modules in the netlist

- routing
  - global routing
    - determining wire congestion
      on the basis of relative positions
  - detailed routing
    - determining the exact geometry
      of the interconnections in the netlist
generating wires: routing

in general, routing is a tough problem that cannot be solved optimally

• given:
  - a set of pin locations
  - a set of nets (prescription for connecting the pins)
  - a routing area, possibly with obstacles and with finite capacity

• constraint:
  - generate a wire pattern that fully connects the pins of each net
  - drc correct

• objectives:
  - minimize wire length
  - good quality wire pattern
versions of the routing problem

• global routing
  - pins and obstacles everywhere
  - overflow allowed

• maze routing
  - pins and obstacles everywhere
  - no overflow

• switchbox routing
  - pins on 4 sides of a box, no obstacles
  - no overflow

• channel routing
  - pins on bottom and top side only, no obstacles
  - channel height can change, completion guarantee.
maze runners

- the entire routing space is represented as a grid
- parts of the grid are blocked
  - components that use the layer
  - previously introduced wires
  - areas preserved for special purposes
- the size of the grid corresponds with the wiring pitch
- the goal is to find a sequence of adjacent gridcells from a source cell to a target cell
- well known algorithm ("lee algorithm") uses
  - wave propagation
    (a wave is all cells that can be reached in i steps)
  - wave propagation stops when the target is in a wave
  - retracing for finding the shortest path
maze runner example
maze runner example
maze runner example
multi-layer lee routing

- extension to multi-layer routing is straightforward:
  - use a '3-dimensional graph'

- furthermore, we always use 'a direction of preference' to maximize routability, typically:

  ![Diagram showing multi-layer routing]

  - Metal3: horizontal preference
  - Metal2: Vertical preference
  - Metal1: horizontal preference
tuning the edge costs

- setting the cost requires a careful trade-off
- typically it is per layer:
  - in the direction of preference: 1
  - perpendicular: 5
  - via to neighboring layer: 10

- the cost level of a layer can be varied

- also, certain patterns could be discouraged, or disallowed:
  - to improve routability
  - stacked vias. this can be modeled in the vertex expansion
problems of maze runners

- large memory requirements
  - start with two waves:
    one from the source and one from the target
  - start in a corner
  - use other wave encoding
  - limit the area (framing)
  - use line router first

- large cpu-time requirements
  - space saving often are also time savings
  - use a depth-first technique towards the target
  - keep track of detours and prefer lower number of detours

- shortest length is not the only objective
  - works well for finding trees
  - other objectives can be included

- no completion guaranteed
  - per net yes
  - but globally not
reducing the wave size

• pick the starting point closest to the edge:

• expand two fronts instead of 1:

\[ \pi r^2 \quad \text{and} \quad 2 \times \pi \times \frac{1}{4} r^2 = \frac{1}{2} \pi r^2 \]

we visit half the number of vertices on average
issues with lee maze routing

- no *completion guarantee*
  - it is sequential nature: routing a net blocks other nets.
  - approach:
    * cost heuristics, ordering, direction of preference
    * rip-up-and-reroute
- slow *$O(n^2)$* behaviour
  - especially for large, empty areas
  - approach:
    * many heuristics
- *grid-level* modeling accuracy of layout is restrictive
  - approach: use finer grid (at the expense of run-time)
reducing memory requirement

- there are many grid points:
  - \( m \times n \times \#\text{layers} \), on a large chip that’s easily 50 Million
  - \( m, n = 10000 \) for a 1M gate design, \( \#\text{layers} \) is 6
  - so that’s 600 million grids
- the straightforward graph datastructure requires about 60 bytes per grid point/layer.
- using the regular properties of the grid graph, this can be reduced to 1 byte per gridpoint.
maze runner example
maze runner example
reducing memory requirement

- there are many grid points:
  - \( m \times n \times \#\text{layers} \), on a large chip that’s easily 50 Million
  - \( m, n = 10000 \) for a 1M gate design, \#layers is 6
  - so that’s 600 million grids

- the straightforward graph datastructure requires about 60 bytes per grid point/layer.

- using the regular properties of the grid graph, this can be reduced to 1 byte per gridpoint.

- but memory is the least of our problems!

sequence 1,1,2,2,1,1,2,2...
reducing run time

CPU-time is the major issue of the Lee router.

where is it spent?
  sub-optimality of the implementation (avoidable)
  sorting the wave front by costs (unavoidable)
  visiting vertices and edges (somewhat avoidable)

how can we reduce the number of vertices evaluated during the vertex expansion?

any ideas on how to speed this thing up?
trading in optimality for speed

• reduce the search space by putting a window around the bounding box of the pins: if the search fails, increase the size of window.

  set window to enclose the 2 pins

• use 'line search' for large, empty areas:
line search example

source

sink
net ordering

- the routing order of the net has a huge influence
- typically, route small nets before large nets
- in the example, we should route net ‘b’ before net ‘a’

route net a before net b

- no ‘silver bullet’ exists.
- what works fine for one situation, is bad for another.
rip-up-and-reroute

• my opinion: this is **EVIL**, but a necessary **EVIL**
• it is ugly, no systematic approach exists. heuristics help a lot

• two approaches in dealing with completion:
  - do not route a net if no path exist: creates ‘**opens**’
    • the grid graph is sparser
  - still route the net: creates ‘**short circuits**’
    • instead of removing occupied edges, assign very high cost.

• then: rip-up some nets, and route them again.
• but... which nets are to be ripped out?
  - with strategy 1: pick a routed net in the neighborhood
  - with strategy 2: rip-up the nets that short
• this takes significant amounts of run-time.
moving towards the target: A*

- this is a slight modification of Lee’s algorithm.
- the cost of a vertex is now also dependent on the estimated cost that is required to reach the destination.
  - result: the ink-blob does not grow as fast in the ‘wrong’ direction.

- So: \( \text{cost}(v) = g(v) + h(v) \)

- actual cost of path from \( s \) to \( v \)
- estimated cost for the remainder of the path from \( v \) to \( d \)
the advantage of $A^*$

- lee

$A^*$
A*: vertex expansion

```c
void AStarVertexExpansion(GRAPH * g, VERTEX * s, VERTEX * d) {
    VERTEXSET wfront; // wavefront of vertices
    VERTEX * v;
    wfront.add(s); s->setCost(0); // adds source to front
    g->resetParentPointers();

    while((v = wfront.removeLowestCost() ) != 0) { // gets lowest cost out
        if(v == d)
            break; // found destination!

        VERTEX::EDGEITER(v); EDGE * e;
        while(e = edgeIter.next() ) { // all edges of shortest
            VERTEX * neighbor = e->otherVertex(v);
            if(neighbor->isAlreadyInsideFront())
                continue; // avoids stepping back
            int vertexCost = v->cost() + e->cost(); // Cost of s to n
            vertexCost += estimatedCost(neighbor, d); // Est. of n to d
            if(vertexCost < neighbor->cost() ) { // Important!
                neighbor->setCost(vertexCost);
                neighbor->setParent(v);
                wfront.add(neighbor); // Add neighbor to front
            }
        }
    }
}
```
the A* algorithm in action

- \( \text{cost}(v) = g(v) + h(v) \)
- if \( h(v) = 0 \) (or constant), then A* is the same as the lee algorithm

- the algorithm produces the minimum-cost path if the remaining cost \( h(v) \) is at least the lower bound

- key is that \( h(v) \) (lower bound for the cost) can be estimated properly. This is not true for general graphs, but easier for (planar) grid graphs

- the vertices with grey costs are not visited during the expansion (their cost is more than 7)
channel routing

- given:
  - an area model (rectangle)
  - net list with pins (fixed on longitudinal sides)
  - design rules
- constraints:
  - connect all nets
  - design rule correct
- optimization criteria:
  - minimize channel height
  - minimize via’s
  - minimize wire length
channel router terminology

http://foghorn.cadlab.lafayette.edu/cadapplets/ChannelRouter.html
channel routing example
classical channel routing

- classical channel routing problem uses two layers:
  - one containing the pins and all lattitudinal parts
  - one containing all longitudinal parts
  - each net has only one longitudinal part
- horizontal constraints:
  - wires in the same layer and overlapping intervals need different tracks
- vertical constraints:
  - wires that have pins at the same longitudinal height must change layer before they overlap
- if the vertical constraints form cycles then the routing cannot be completed in the classical model
- if there are no cycles in the vertical constraints then a solution to the classical channel routing problem exists, but finding the minimum number of tracks is NP-hard
concept: reduce search space:

- implement each net by a single horizontal wire (trunk)
- connect pins to wire by vertical branches
- 'tetris'-style compaction: share the rows
- the trunk spans the entire net:

```
2 4 1 3 6 6 7
   7
   6
   5
   4
   3
   2
   1
```

```
1 3 5 2 7 5 4
```
horizontal constraint graph

an edge in the complement of the horizontal constraint graph indicates that two segments might end up in the same track

each node represents an interval (of a net). assume that the ‘color’ of the node is the track number.

track assignment = graph coloring of the horizontal constraint graph:

constraint: no pair of 2 adjacent nodes have the same color
objective: minimize the number of colors.

for general graphs coloring is NP-hard
horizontal constraint graph

each node represents an interval (of a net).
assume that the 'color' of the node is the track number.
track assignment = graph coloring of the horizontal constraint graph:
  constraint: no pair of 2 adjacent nodes have the same color
  objective: minimize the number of colors.

for general graphs coloring is NP-complete
the “density” of a channel

- a net extends from its leftmost terminal to its rightmost one
- local density at an arbitrary column $C$
  - $\text{Id}(C) = \# \text{ nets split by column } C$
- channel density
  - $d = \max \text{Id}(c)$ over all $C$
- relationship to horizontal constraint graph?
  - local density $\Leftrightarrow$ clique
  - $d \Leftrightarrow$ size of maximum clique
- lower bound:
  - $\# \text{ tracks} \geq d$
coloring the graph

5 colors (= tracks)

4 colors (= tracks)
optimally coloring: 'left edge' algorithm

Hashimoto & Stevens 1971

```java
set.read(nets) // store all nets

for(color = 1; set.isNotEmpty(); color++) {
    int x = 0; // left coordinate
    while(interval * i = set.getFirstLeftOf(x)) {
        i->setTrack(color); // assign to track
        x = i->rightCoordinate();
    }
}
```
vertical constraints

channel needs to satisfy both horizontal and vertical constraints!
cycles in the vertical constraint graph

dogleg

this is called a ‘dogleg’
using doglegs

- doglegs may reduce the longest path in VCG

question:
which nets to cut such that channel height is still minimized?

this was proven NP-hard (Szymanski 1985).
even worse, nobody ever came up with decent heuristic solution...

bottom line: this model turned out to be very poor!
alternative approach: ‘greedy’ routing

= take ad-hoc (greedy) decisions,
... and make the best of it.

while(there are unrouted nets) {
    pick a ‘promising’ net
    route it as as best as you can
}

instead of:

set.read(nets) // store all nets

for(color = 1; set.isNotEmpty(); color++) {
    int x = 0; // left coordinate
    while(interval * i = set.getFirstLeftOf(x)) {
        i->setTrack(color); // assign to track
        x = i->rightCoordinate();
    }
}
this issue with greedy routing

• the concept is to take (small) greedy steps:
  - each time implement the pattern in full detail.
  - no back-tracking! once a wire is laid down, it stays there!

ALARM!
Putting down net 1 like this blocks the connection from 5!
new bottom track needed!!!
the greedy routability invariant

• at each step:
  - lay down a partial wire pattern in full detail
    (tracks are not changed anymore)

• such that the following invariant is maintained:
  - 1: the existing layout pattern is drc correct
  - 2: still unrouted pins (nets) remain routable.

• this should guarantee routing completion!

• this works for channel routing
• it does not work for area routing!
column-greedy routing

- idea: sweep a column from left to right through the channel
  - the part left of the column is fully routed, the right part isn’t

- maintaining the routability invariant by widening the channel

- published by Rivest and Ficuccia, DAC 1982
column-greedy routing

for (each column) {
    // Part 1: take care of the invariant constraint
    for (each pin at bottom/top of this column) {
        if (no connectable trunk exists) {
            if (no empty track)
                widen channel, create new track
            create trunk at first empty track
        }
        connect pin to trunk
    }
    // Part 2: heuristic optimization
    collapse trunks of same net
    move trunk up/down in the direction of upcoming pins
    other heuristics
}

Rivest and Fiduccia 1982
net greedy routing

- idea: route nets one at a time [Groeneveld, ICCAD 1987]

- routability invariant maintained by allocating a 'territory' above each unrouted pin (this is a 'keep-out' column)

- the channel is partitioned into 2 halves, each of flexible height
net-greedy gridless routing

using contours (one per layer)

gridless
variable width
variable spacing
completion guarantee

- net greedy: all instances of the classical routing are routable
- ... all except one unlikely situation:
completion guarantee (2)

• adding just one empty column makes the entire channel routable
completion guarantee (3)