DESIGN AUTOMATION

5MD20

the A* algorithm
and area routing

Ralph Otten
otten@ics.ele.tue.nl
February-March 2010
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:

Metal1: horizontal preference
Metal2: Vertical preference
Metal3: 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 savings 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:

\[ \pi r^2 \]

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

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, \#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
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, $\#\text{layers}$ 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.

- but memory is the least of our problems!

sequence 1,1,2,2,1,1,2,2,…
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
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.
"informed" search

- this is a slight modification of the lee 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: \( \text{cost}(v) = g(v) + h(v) \)

- informed search

- lee
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 not underestimated (that is, at least the lower bound)
- Key is that \( h(v) \) can be estimated properly (lower bound for the cost); this is easy for (planar) grid graphs
- The vertices with grey costs are not visited during the expansion (their cost is more than 7)
A* intuition

• incrementally build all routes from the starting point until goal reached
• prefer routes that appear to lead towards the goal (like all informed search algorithms)
• to guide use a heuristic distance estimation from any given point to the goal (e.g. straight-line distance)
• what sets A* apart from best first search is that it also takes the distance already travelled into account
• this makes A* complete and optimal, i.e., A* will always find the shortest route if any exists
• not guaranteed to outperform simpler search algorithms (e.g. a maze-like environment)
the algorithm

• A* maintains a set of partial solutions, i.e. paths through the graph starting at the start node, stored in a priority queue
• priority assigned to a path with the lowest \( f(p) = g(p) + h(p) \) (here, \( g(p) \) is the cost of the path so far, i.e. the weight of the edges followed so far. \( h(p) \) is an estimate of the minimal cost to reach the target from the endpoint of \( p \))

• for example, if "cost" is taken to mean distance travelled, the detour-free distance between two points on a map is a heuristic estimate of the distance to be travelled: the lower \( f(p) \), the higher the priority (so a min-heap could be used to implement the queue).

on the grid for area routing with equal weights:
• \( h(p) \) is the manhattan distance between endpoint and target
• \( f(p) \) is the label assigned to the endpoint gridcell
• if \( h(p)=0 \) the algorithm simplifies to the lee algorithm
the algorithm

```
function A*(start, goal)
  var closed := the empty set
  var q := make_queue(path(start))
  while q is not empty
    var p := remove_first(q)
    var x := the last node of p
    if x in closed
      continue
    if x = goal
      return p
    add x to closed
    foreach y in successors(p)
      enqueue(q, y)
  return failure
```

Here, successors(p) returns the set of paths created by extending p with one neighbor node. It is assumed that the queue maintains an ordering by $f$-value automatically. In the closed set (closed), all the last node of p (nodes with paths found) are recorded, so as to avoid repetition and cycles (making this a graph search). The queue is sometimes analogously called the open set. The closed set can be omitted (yielding a tree search algorithm) if either a solution is guaranteed to exist, or if the successors function is adapted to reject cycles.
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) // found destination!
            break;

        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
            }
        }
    }
}
```
properties

• A* is *complete* in the sense that it will always find a solution if there is one  
  (like breadth first search)

• if the heuristic function $h$ is *admissible*,  
  meaning that it never *overestimates* the actual minimal cost  
  then A* is *optimal*, that is, it finds the shortest path

• A* is also *optimally efficient* for any heuristic $h$,  
  meaning that no algorithm employing the same heuristic  
  will expand fewer nodes than A*,  
  (except when there are several partial solutions  
   where $h$ exactly predicts the cost of the optimal path)