VE527

Computer-Aided Design of Integrated Circuits

Maze Routing Implementation Issues

Outline

- Maze Routing Implementation Issues
  - Expansion Step
  - Data Structures
  - Depth-First Search
  - Global Routing

Implementation Concerns

- Algorithms
  - Question: if we can only process one cell at a time…
  - …then, which cell is next to "label" in the search process

- Data structure (representation)
  - How do we store the routing grid?
  - What do we need in each cell?
  - How do we represent the state of the path search process?

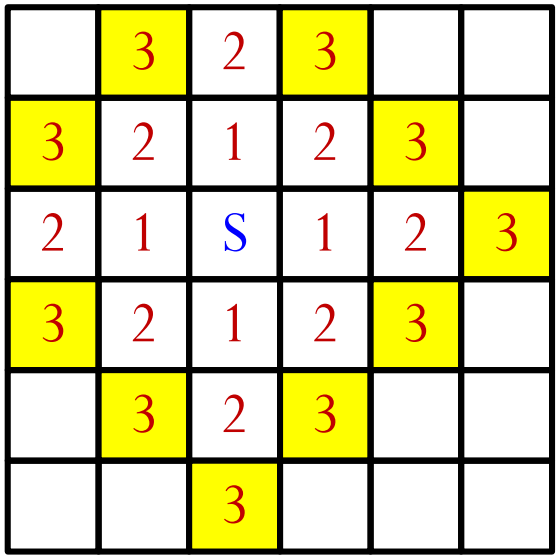
First Issue: Expansion Step

- One big goal
  - Efficient storage: Big layout needs a big grid; put as little in each cell as possible.

- Big Idea: **Do not** store path costs in grid cells
  - If we have big path cost, then we need many bits per cell.
  - Notice: Only cells **most recently labeled** during search are used to **expand** the search.
  - Important terminology: these cells comprise the **search wavefront**.
    - It is the **wavefront** that is really most important….

Important Idea: Search Wavefront

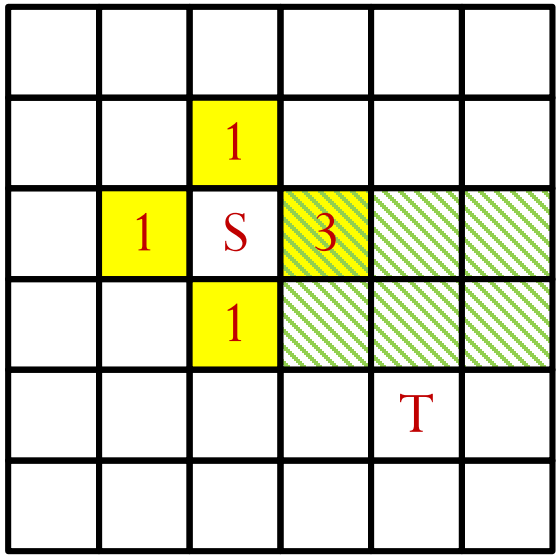
- What is wavefront?
  - Set of cells already **reached** in the expansion process…
  - …that **have neighbors** that we have **not** yet reached.
  - Wavefront is the **frontier** of the active search for new paths.
- Implication
  - **Don't** store the pathcost numbers **in the grid**.
  - Just store the wavefront cells themselves in a **separate data structure**.



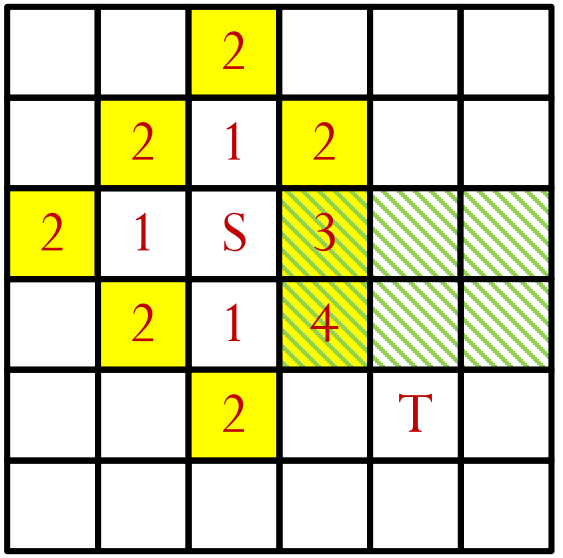
How to Expand Wavefront?

Answer: Expand in the **pathcost** order

After expanding neighbors of the cost "**0**" grid, i.e., S



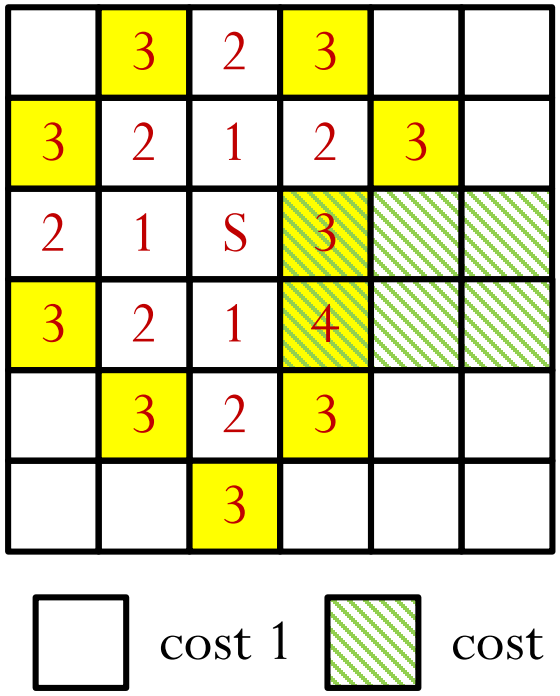
After expanding neighbors of the cost "**1**" grid



cost 1 cost 3

More Complex Wavefront

After expanding neighbors of the cost "**2**" grid



- **Expanded** in **pathcost order**, cheapest cells before more expensive cells.
  - Algorithm: **Cheapest-cell-first search** (a variants of Dijkstra's algorithm)

Cheapest-Cell-First Search

- Find a **cheapest** cell C from the **wavefront**.
- Find the neighbors {N1, N2, … Nk} of cell C that you have **not reached** yet.
- Compute the cost of expanding through C to reach the cells in {N1, N2, … Nk}.
- Add new cells {N1, N2, … Nk} to the wavefront data structure, including their newly computed pathcosts, {pathcost(N1), pathcost(N2), …, pathcost(Nk)}.
  - Also, add to each of these new cells a pointer to cell C as the **predecessor**.
  - Mark the routing grid to remember that we have **reached** these cells {N1, N2, … Nk}.
- Remove cell C from the wavefront.
- Repeat with the **next cheapest cell on wavefront**…

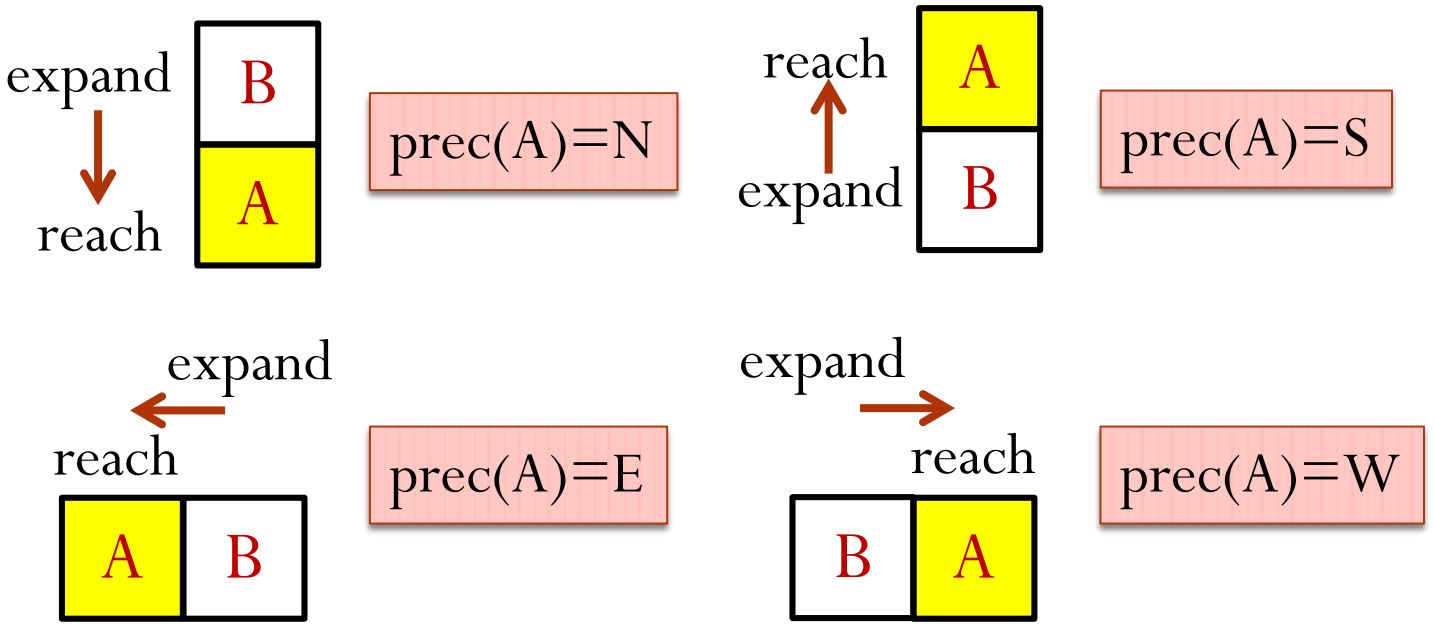
Cell Predecessor

- **Predecessor** of cell A on the wavefront is a "tag" that tells **direction** from which cell A was reached, i.e., it's how we found A.

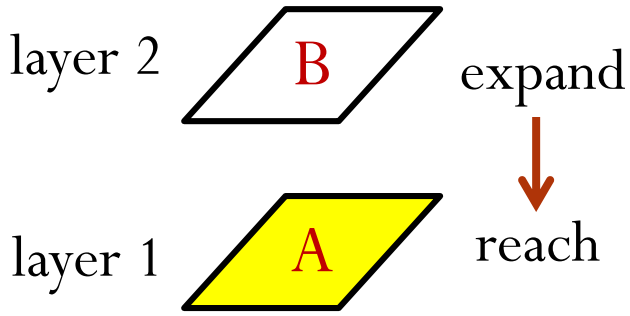
- Why do we have to remember this?
  - Because we **do not mark pathcosts** in the grid any longer.
  - Then, to backtrace the path, we cannot simply follow the numbers in decreasing order.
  - We need a real "trail" that points, cell to cell, from target back to source.

Marking the Predecessor

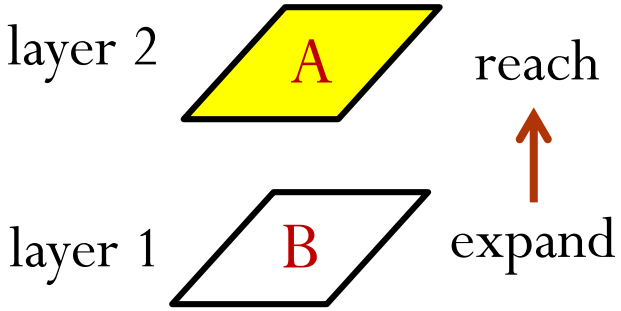
- Assume 2 routing layers. 6 pred(A) tags:
  N, S, E, W, Up, Down
  - We only need 3 bits to store 6 tags.
  - **Note**: pred(A) is the direction from which cell A was reached



Marking the Predecessor (cont.)

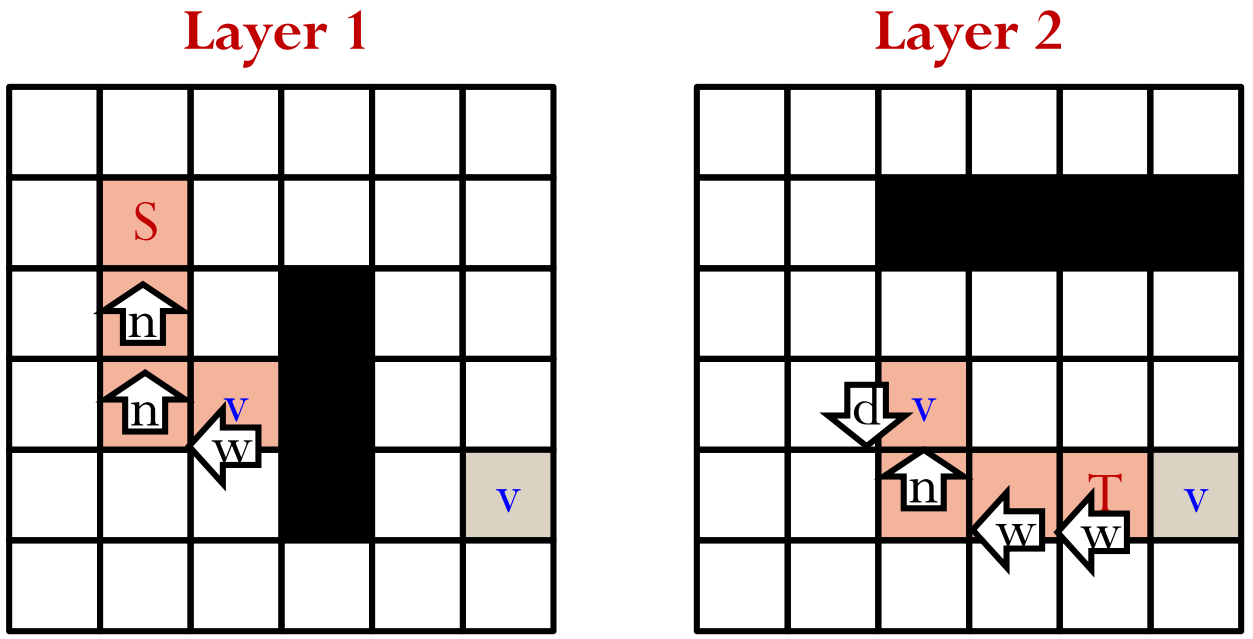


prec(A)=Up



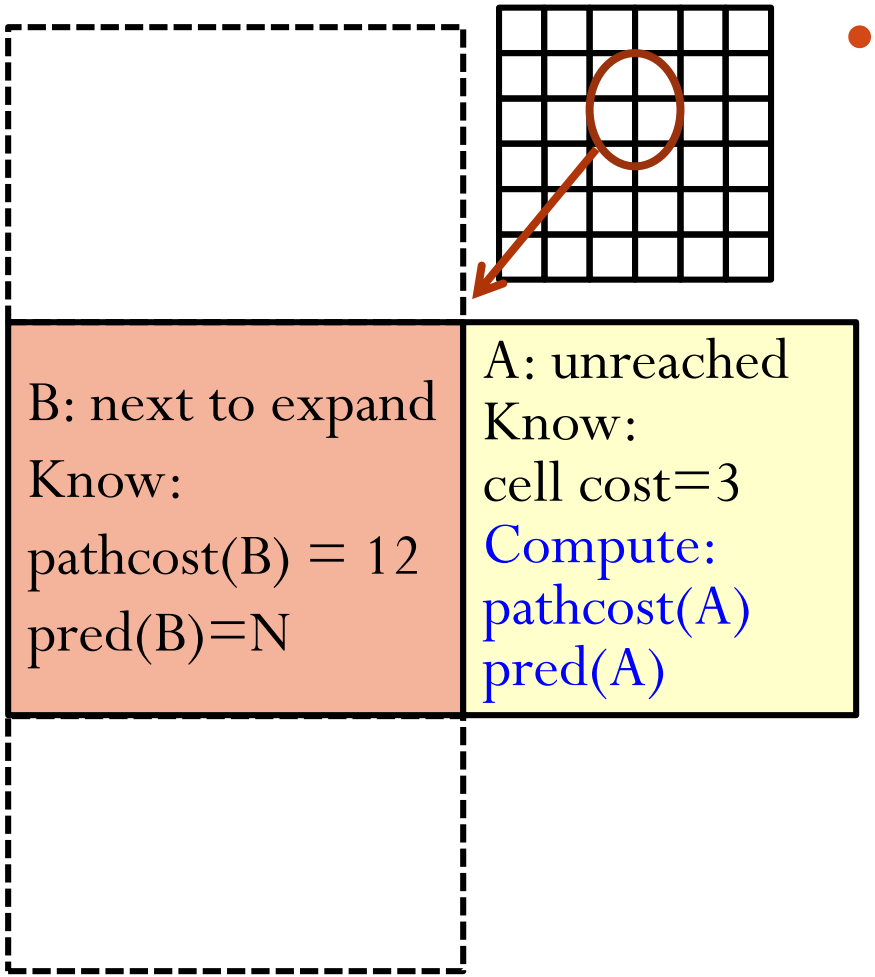
prec(A)=Down

We Store Predecessors on Grid



- Result: backtrace is easy! Follow pred(C) tags in the grid.

Example: Expanding & Reaching



- Expand cell B, reach cell A
  - B is with **cheapest** cost on wavefront.
  - A is B's **unreached** neighbor.
  - Compute cost to reach A from B: 12+3=15
  - Add this new "cell object" to the wavefront:

<layer=L; cell location=(x_A, y_A); pathcost=15>

  - Mark cell A as Reached(A)=true
    - In future, we won't try to put it on the wavefront **again.**
  - In cell A in the gird, set pred(A) =W

Basic Maze Routing Algorithm

```
wavefront = { source cells };
while (we have not reached target cell) {
    if ( wavefront == empty )  quit; // no path to be found
    C = get lowest cost cell on wavefront structure;
    if ( C == target )  {
        backtrace path in grid; // follow pred( ) pointers to source
        cleanup & return; // we found a path
    }
    foreach ( unreached neighbor N of cell C ) {
        mark cell N in grid as reached.
        mark pred(N) with direction from cell N to C;
        // compute cost to reach
        pathcost(N) = pathcost(C) + cellcost(N) ;
        add cell N to wavefront, indexed by pathcost(N);
    }
    delete cell C from wavefront;
}
```

Observations

- Core algorithm is fairly simple. Works for grid with non uniform costs.

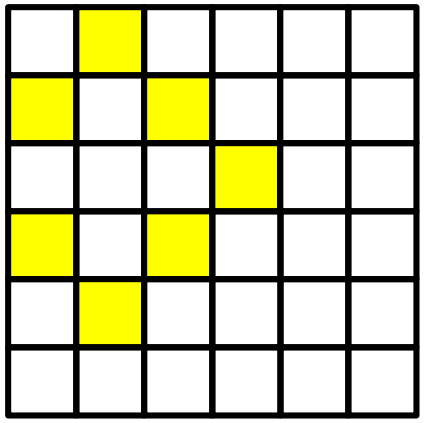
- Extends naturally to other features
  - **Multi-point nets**: just mark all the cells of intermediate paths as source, put them all in the wavefront to route to net point.
  - **Multiple routing layers**: just use parallel grids. When you expand to reach neighbors, check UP, DOWN to other layers. Use pred(Cell) = UP, DOWN to do backtrace.

Outline

- Maze Routing Implementation Issues
  - Expansion Step
  - Data Structures
  - Depth-First Search
  - Global Routing

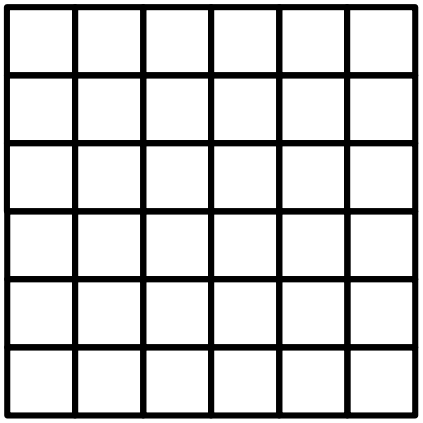
Two Key Data Structures: Overview

- Routing grid
  - Routing surface. Contains blockage. Has costs of each cell.
  - Mark these cells if they have already been reached.
  - Mark predecessor.



- Wavefront
  - Holds active cells **to be expanded**.
  - Cells store pathcost.
  - Indexed on pathcost; always expand cell with cheapest pathcost next.

Routing Grid



- Each grid cell C stores:
  - (x, y) coordinate of cell C
  - Layer of grid cell C
  - Cost(C) (a small number)
    - **Note**: we can use cost = −1 to indicate blockage
  - Pred(C) tag = N,S,E,W,Up,Down
  - Reached(C), 1 bit Boolean true/false

Wavefront

- Each cell C in wavefront stores:
  - (x, y) coordinate of cell C
  - Layer of grid cell C
  - Pathcost(C) = length of the shortest path to cell C

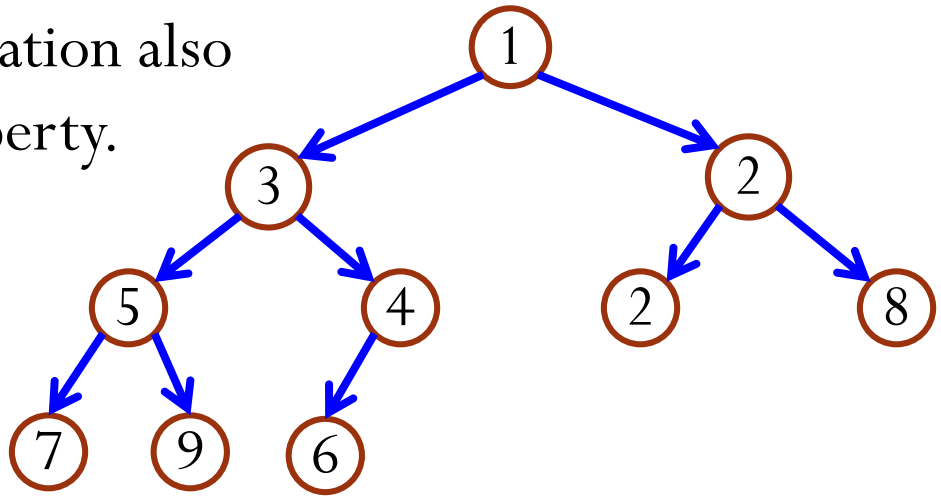
- We need something clever here:
  - Need fast **extracting minimum** operations, to take a cell with the smallest pathcost from wavefront
  - Need fast insertion, to add a cell into the wavefront

Wavefront Structure: Min Heap

- Store cells of wavefront in a **min heap**.
  - Also called a **min priority queue**.
  - Classical data structure designed for fast insertion and extracting minimum operations.
    - These two operations have $O(\log N)$ time complexity for N objects.

Recall: Min Heap

- A min heap is a **complete binary tree** that
  - For **any** node v, the key of v is smaller than or equal to ($\leq$) the keys of any **descendants** of v.
- The item with the smallest cost is the **root**!
- Insert operation "bubbles up" the new item in heap to ensure the min heap property.
- Extracting minimum operation also ensures the min heap property.



Outline

- Maze Routing Implementation Issues
  - Expansion Step
  - Data Structures
  - Depth-First Search
  - Global Routing

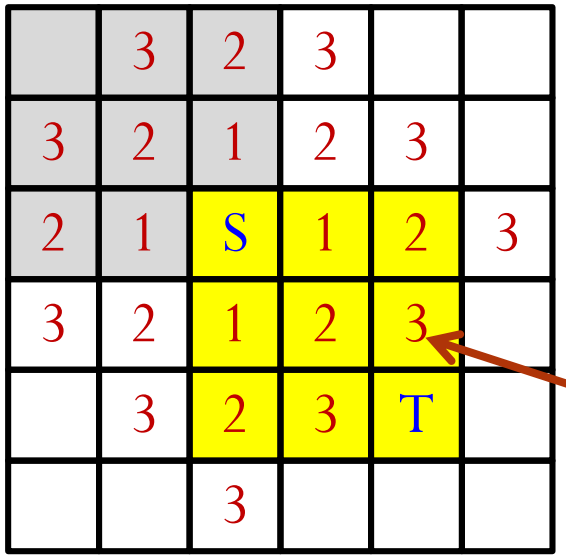
Expansion Process, Revisited

- Problem:
  - Expand lots of cells to find one path to the target.
  - CPU time is proportional to number of cells you expand.
  - No attempt to search in direction of target first.

- Questions:
  - Can we actually bias expansion so we search **toward** the target?
  - Can we do this and still keep **guarantees** of reaching target with minimum cost path?

Motivation for Smart Search

Expanding **away** from the target seems to be a waste of time.

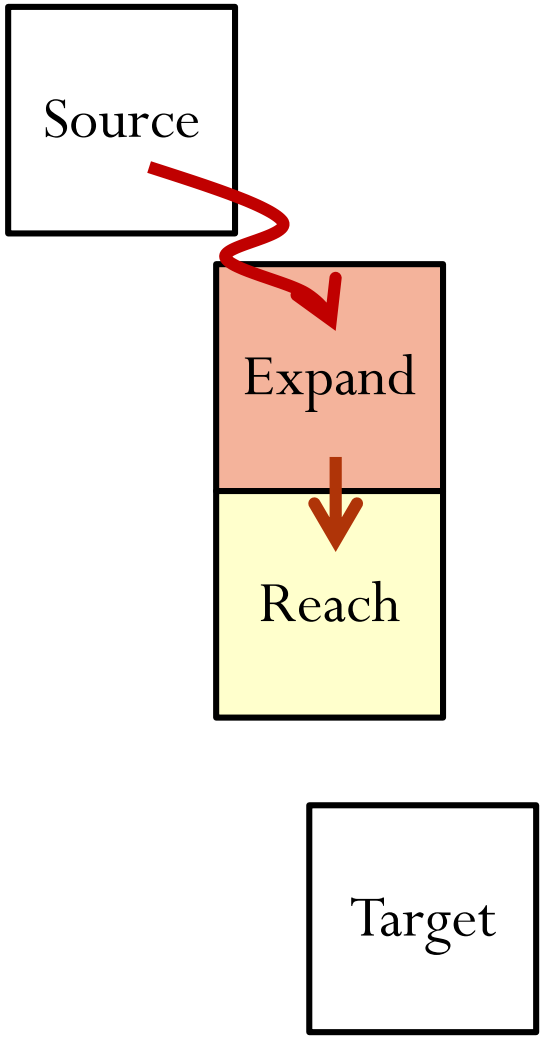


Searching **toward** the target in the shaded region, sometimes called the "source-target box", seems a good idea to try first.

Smarter Search: Maze Search With Predictor

- Add **predictor function** to the cost, to direct the search **toward** the target.
- Plain maze router
  - Add a cell C to wavefront with cost that measures cost of **partial path**, source-to-cell(C).
- Smarter maze router
  - Add cell C to wavefront with cost that **estimates entire source-to-target cost** of path **through the cell**.
  - Trick: estimate this as pathcost(source to cell C) + **predictor(cell C to target)**.
  - We call this method as smart depth first search (DFS) with a predictor in contrast to classical breadth first search (BFS).

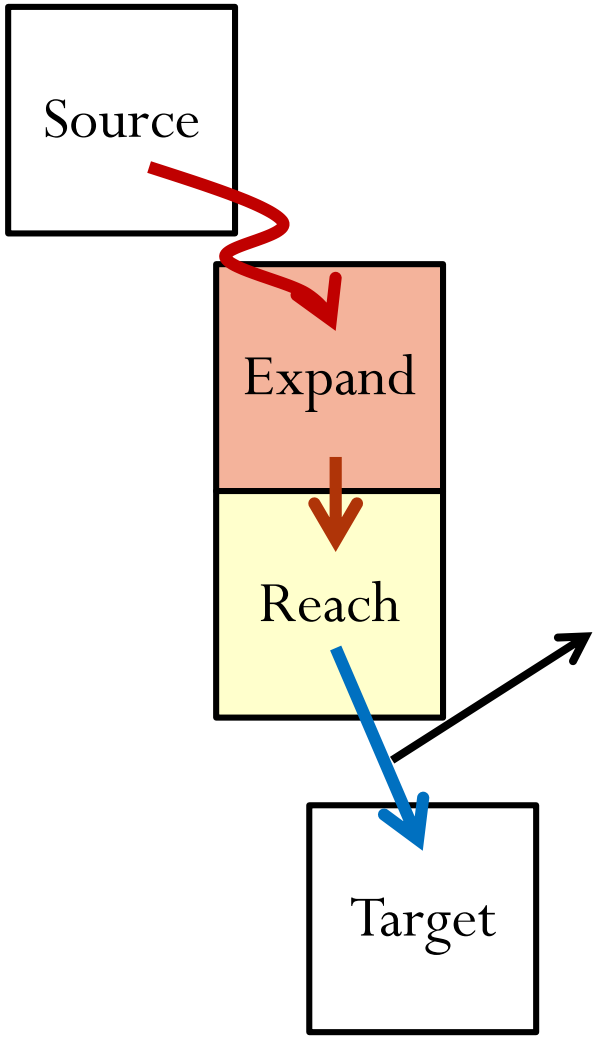
Plain Maze Routing



Reached cell goes on wavefront with this cost:

pathcost(source to cell being expanded) + cost of newly reached cell

Add A Depth-First Predictor



Reached cell goes on wavefront with this cost:

pathcost(source to cell being expanded) + cost of newly reached cell

+ estimate of pathcost(from newly reached cell to target)

A typical estimator is:

(Manhattan distance to target) × (Smallest cell cost)

Technical Results

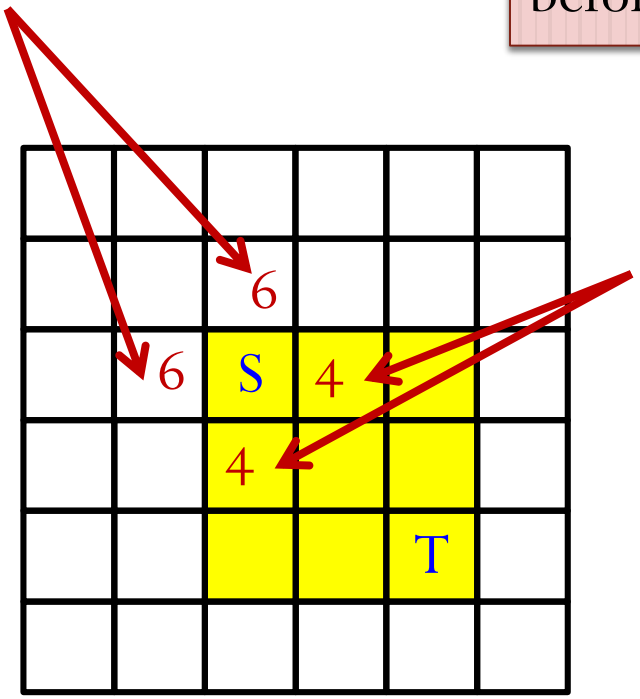
- Depth first predictor
  - It is famous application of a classical idea: **A* search**.
  - Since predictor is a **lower bound** on extra pathcost you add to reach target…
  - …by the property of A* search, you are **guaranteed** to get the minimum cost path.

- What does it do?
  - It **alters the order** in which we expand cells.
  - It prefers to expand cells that are **closer to the target first**.

Depth-First Expansion Example

Pathcost = 1;
Distance to target = 5
Esitmate of cost: 1+5=6

Observation: Search prefers to stay **inside** the bounding box of the source-target rectangle before it expands other cells.

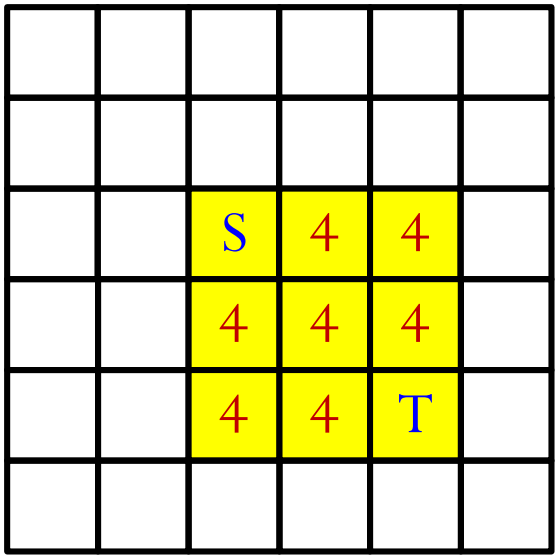


Pathcost = 1;
Distance to target = 3
Esitmate of cost: 1+3=4

Depth-First Expansion Example

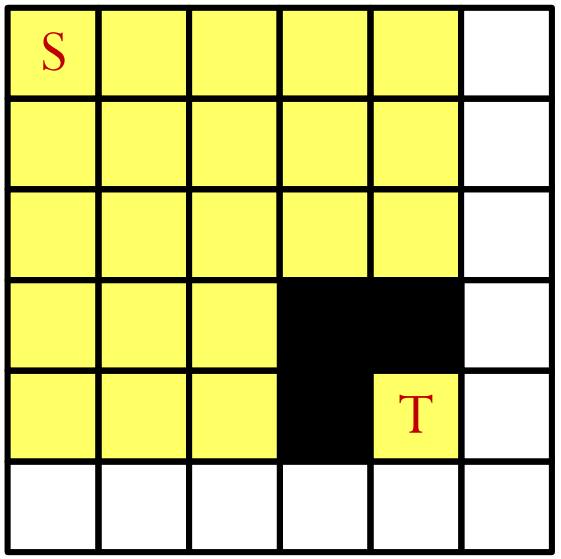
- It turns out for this example all the cells in this box cost the same!
  - **In this box**:

  pathcost(source→cell) + estimated_pathcost(cell→target) = constant
  - So – it expands **toward** the target!

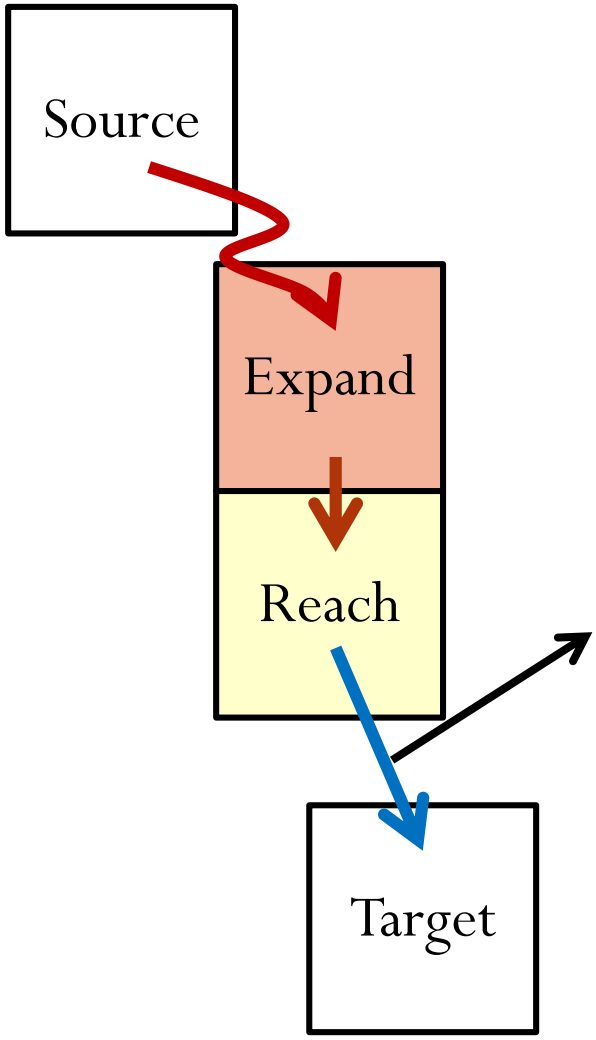


DFS Expansion: Unintended Consequences

- This is "the" example of which "stay in the box" is **inefficient**.
  - The target is blocked inside the source-target box.
  - **Problem**: DFS explores the **whole shaded** rectangle before it tries anyplace else.
  - In this case, it may be faster to search outside the box!
- But, generally, this heuristic is good!



A Variation of the Heuristic



Reached cell goes on wavefront with this cost:

pathcost(source to cell being expanded) + cost of newly reached cell + **K** × estimate of pathcost(from newly reached cell to target)

K is usually (1 + a very small number)

Idea: Insist that expand in a cheaper path toward target. Distorts optimum a bit, but it is faster.

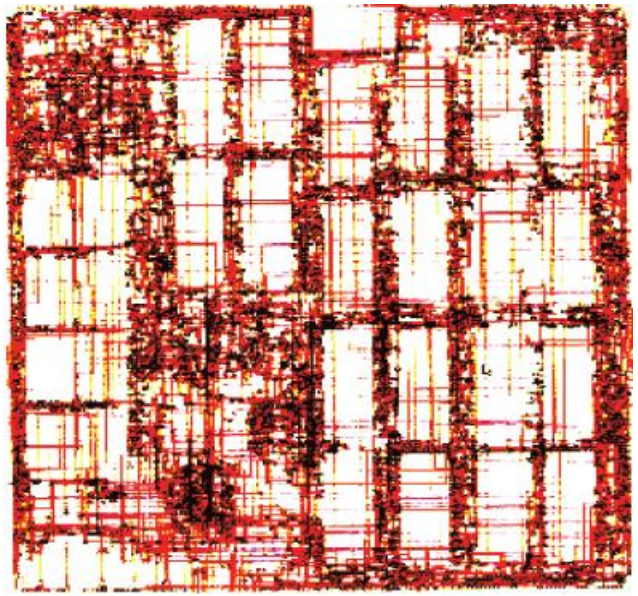
Outline

- Maze Routing Implementation Issues
  - Expansion Step
  - Data Structures
  - Depth-First Search
  - Global Routing

Reality: ASIC Scale & Complexity

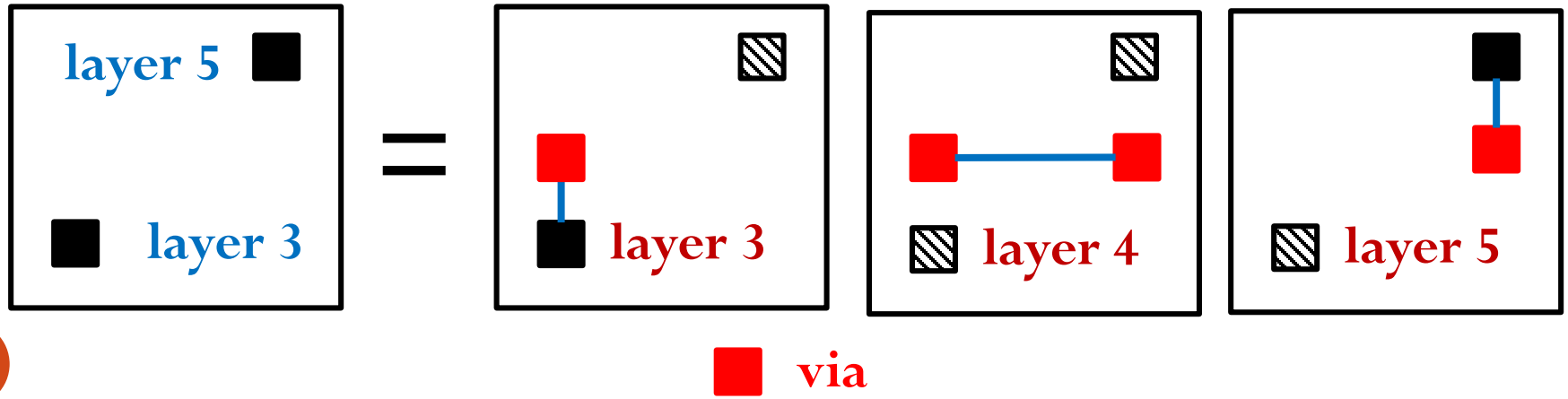
- Big chip, 1cm x 1cm
- 20-50 million nets
- Modern IC technology, ~100nm pitch for wires (grid size)
- So, 100K x 100K routing grid!
- 10 routing layers
- 100 billion grid cells!
- Do we really do it like this?

No!



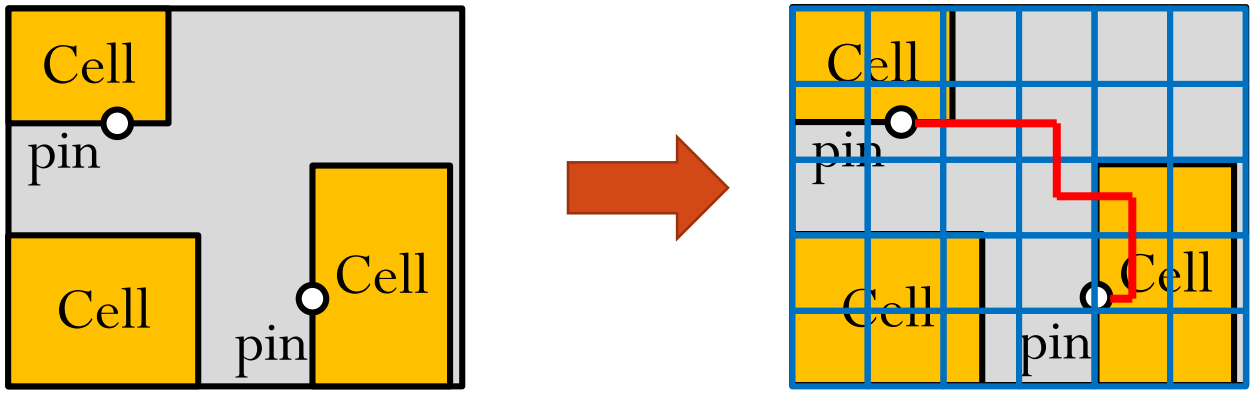
One Way: Preferred Routing Directions

- Every layer of metal wiring has a **preferred direction**.
  - E.g., metal 3, 5, 7, 9 prefer **vertical**; metal 4, 6, 8, 10 prefer **horizontal**.
  - As a result, all the wires on these layers look like straight lines.
  - If you need to bend, use a **via**. Makes it easier to embed lots of wires.
- This is **one way** to reduce complexity.

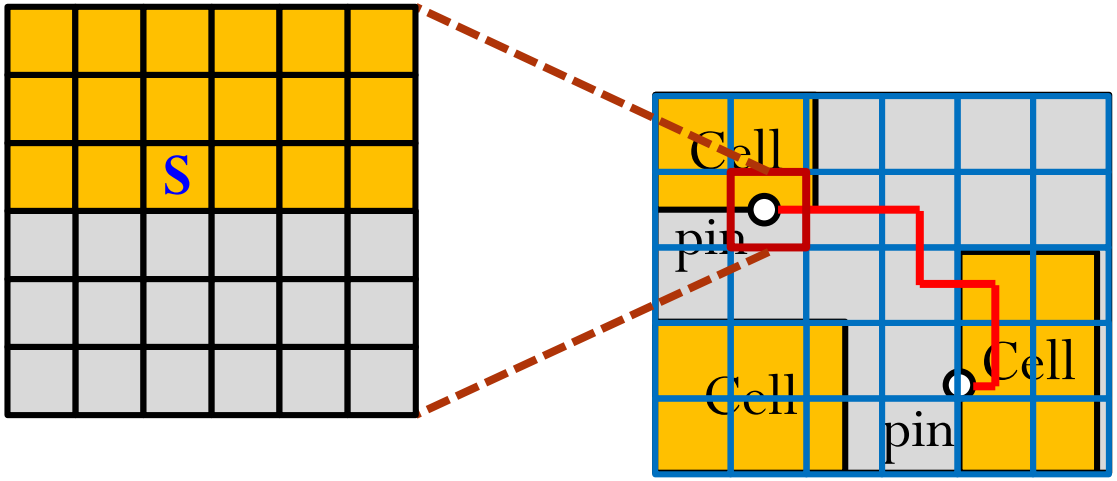


Divide & Conquer: Global Routing

- All the routing we have done so far is called **Detailed Routing**, which plans **exact, final** path of wire.
- How to deal with huge scale of chips? **Global Routing**
  - Global routing also imposes a grid on surface of the chip.
  - But now, this grid is much coarser, e.g., each grid cell is about 200 wire grids × 200 wire grids in size.
- Big Idea: **Maze route** through these big, coarse regions.



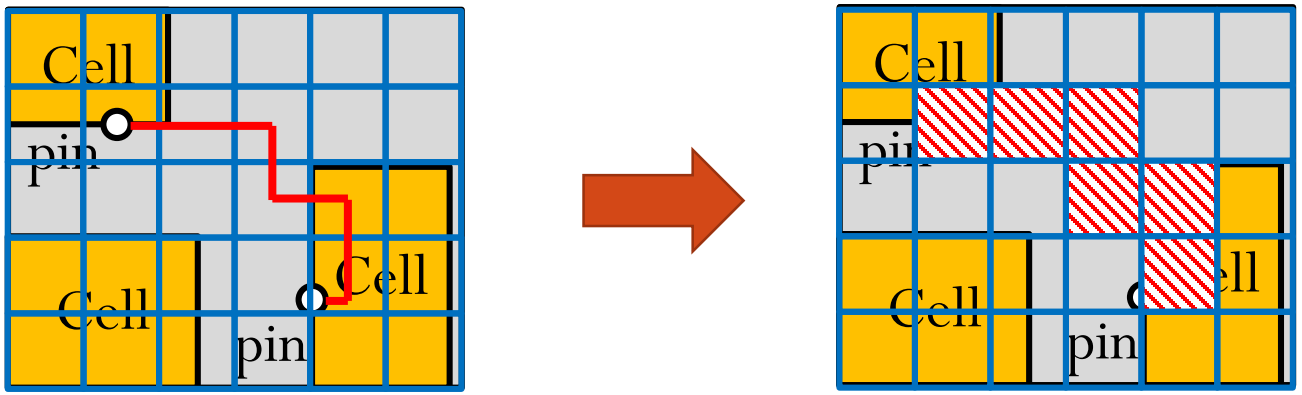
Global vs Detailed Routing: Geometry



- Each cell in **Global Routing** grid is a box (called a **GBOX**) with size typically 100~200 wire grids on a side.
  - This example just has 6 for clarity.

What Global Routing Does?

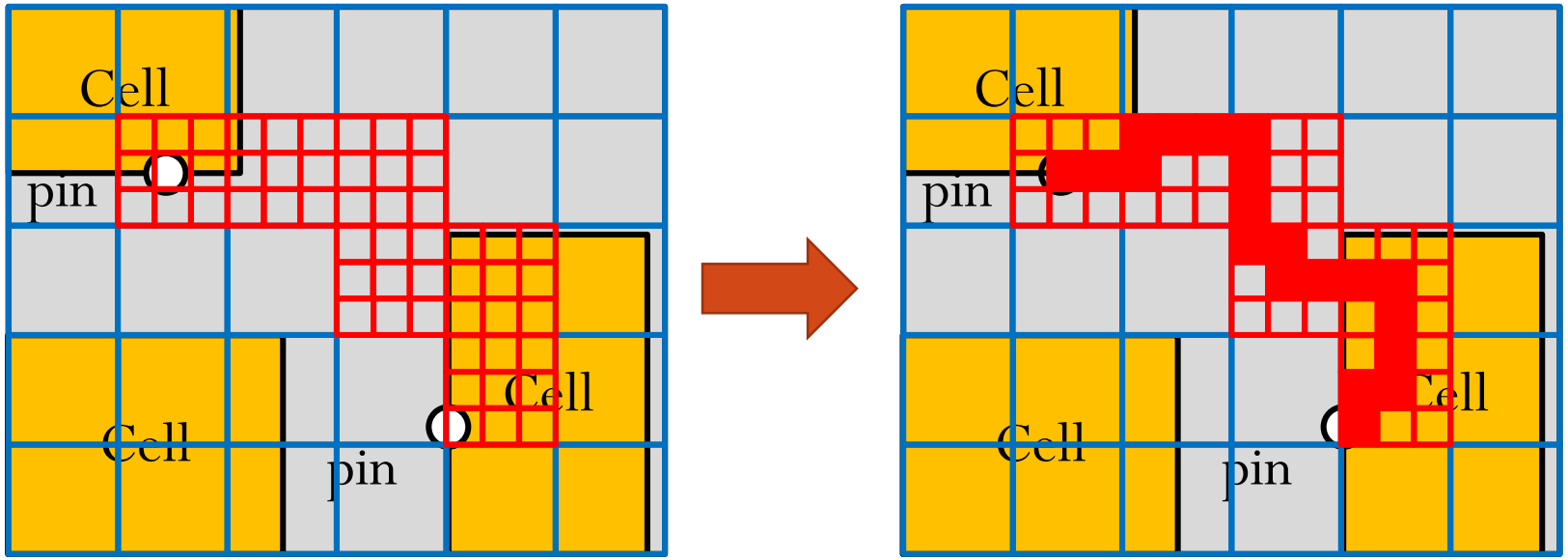
- Global routing tells us we want this net to use this **rough** path; but not on which **exact** path inside this region.
  - Generate **regions of confinement** (i.e., coarse path) for a wire.
- Makes sure overall **wiring congestion** is reasonable.
  - Balance **supply** (how much available space) vs **demand** (how many paths want to go here).



What Detailed Routing Does?

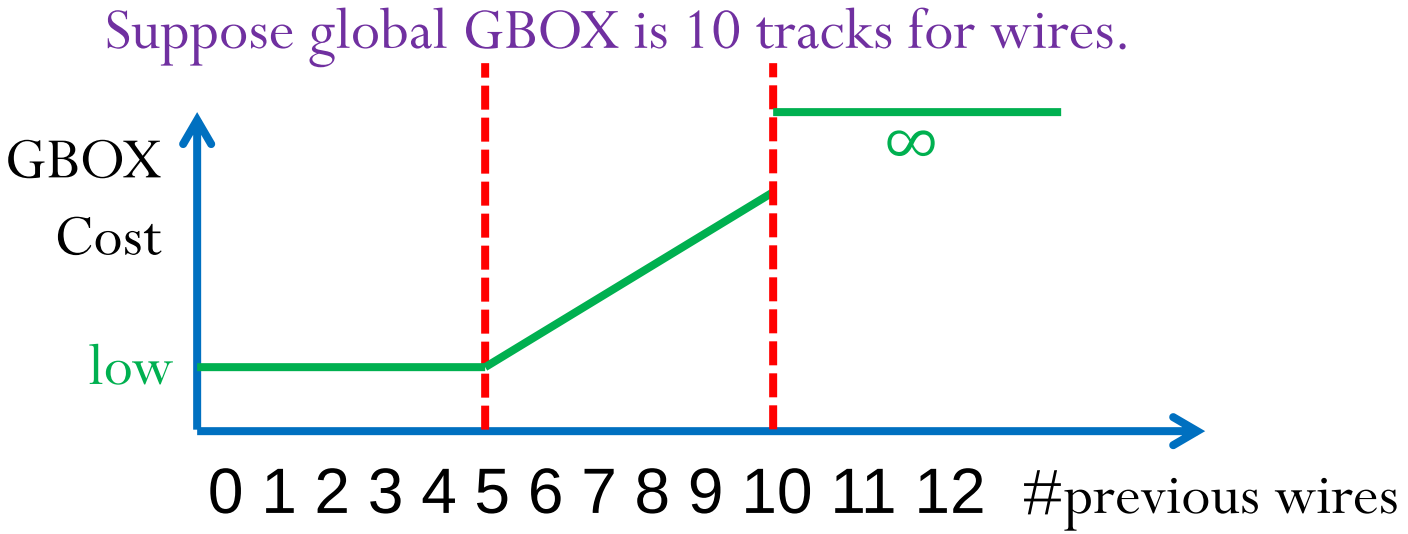
- Detailed routing embeds **exact** paths in these regions.
  - Simplest model: **a finer grid**. Require wires to use tracks on this grid.

This reduces computation!



Global Routing as Maze Routing

- What's different for a global router? **Cell Cost Function**
  - **Detailed router**: grid cells are blocked (cost=infinity) or they have a cost proportional to how much we want to avoid them.
  - **Global router**: grid cells have **dynamic** cost, proportional to how many **previously** routed wires used this cell, how many more can fit **later**.



Maze Routing: Summary

- Has been around a long time.
  - Very **flexible** cost-based search; can be recast to attack many problems.
  - **Dominates** most modern routers…
  - Some other approaches, e.g., sophisticated ”grid-less” representation of “space”.
- Lots of ancillary problems (and solutions)
  - Hierarchy/abstraction (e.g., global routing).
  - Iterative improvement strategies.