# (*Lec 15*) ASIC Layout: Routing by Maze Search

- What you know
  - Elementary ASIC gate placement by annealing
  - Given the netlist: where do we put gates to get min. estimated wire length
- What you don't know
  - How to actually wire the gates together: called *routing*
  - Flavors of routing: global versus detailed, area versus region
  - Our technical focus: area routing by maze routing



# Copyright Notice

© Rob A. Rutenbar 2001
All rights reserved.
You may not make copies of this material in any form without my express permission.

## Where Are We?

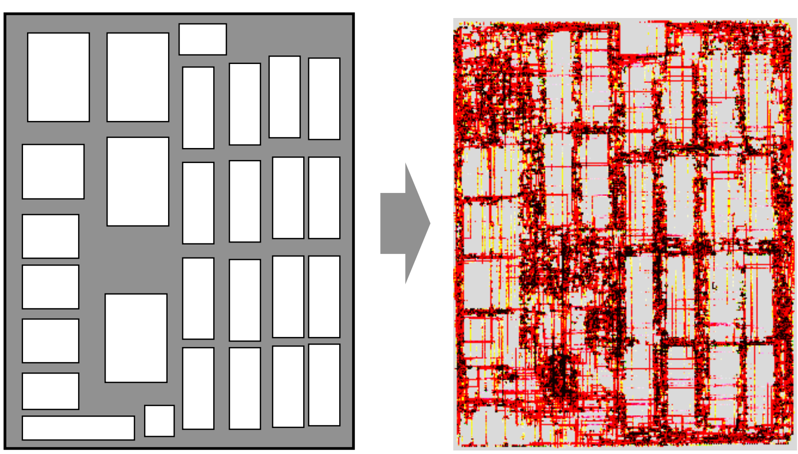
- Physical design--how to *wire* the placed gates...?

| | 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 |
| | 15 | 16 | 17 | 18 | 19 | 8 |
| | 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



## Routing: The Problem



Dozens of circuits,
1000s of blocks,
1,000,000s of gates

Many meters of wire.

## 3 Basic Routing Problems

- Size complexity
  - Big chips have an enormous number (100,000s, 1,000,000s) of wires
  - Not every wire gets to take an "easy" path to connect its pins; there may be too much "congestion", make path-finding hard
  - Essential to connect them all--can't afford to tweak many wires manually
- Shape complexity
  - It used to be that the representation of the layout was a simple "grid"
  - You knew where pin could / couldn't be, where wire could / couldn't go
  - In modern fab processes, it's not like this anymore.
  - All wire geometry, wire material layers can have complex geometric rules they must obey to be "design rule legal" in the layout
- Timing complexity
  - It's not enough to make sure you connect all the wires
  - You also must ensure that the delays thru the wires are not too big



## Basic Solutions

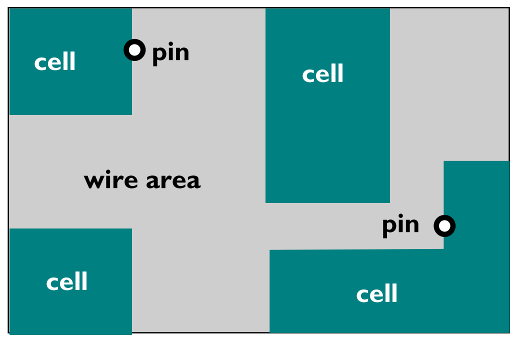
- Size complexity
  - Divide & conquer: don't just solve "one big routing problem"
  - Solve of sequence of routing problems that "refine" routing
  - Start with "global" model of routing, end with "detailed" routing
- Shape complexity
  - Coarse routing steps: are often "gridded", ie, you assume wires fall on some nice grid of legal locations. This is a simplification, but OK here.
  - Detailed routing steps: either require some underlying grid for all the pins, or use "gridless" path search techniques to find paths
- Timing complexity
  - First, make sure placement is good enough that you can hit timing
  - Account for timing (using different abstractions of "time") at each level of routing, from coarse to fine
  - Iterative improvement: identify problems, go back and try to fix 'em

## A (Very) Short Historical Tour: Routing

- In the beginning of chip routing…
  - Used ideas borrowed from PC-board routing
  - Only had 2 routing layers (one Horizontal, one Vertical, typically)
  - So, had to route "around" the placed objects, not "over" them

So, start with overall floorplan of placed chip.

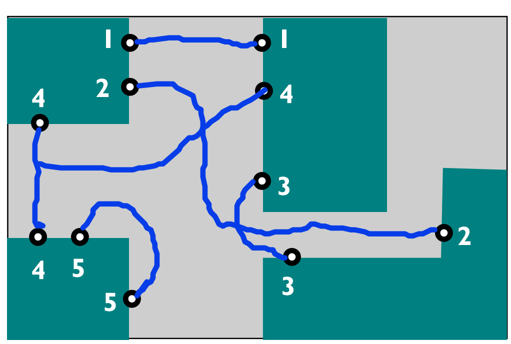
Want to connect pins thru the regions left empty for routing






## Routing: Global Routing

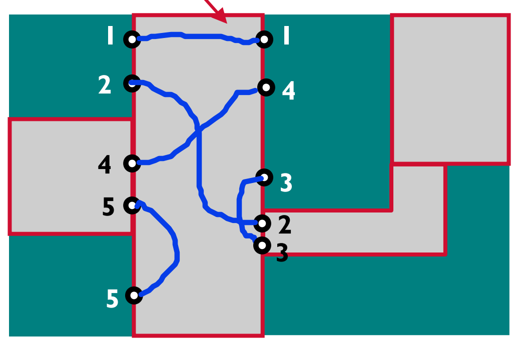
- Usually start with *global* or *coarse* routing
  - Chop up chip into big regions
  - Decide *thru which regions* the wires will go, but not exactly where each rectangle of each individual wire will go
  - Idea is to plan global paths for the wires, so we know early we can fit them all in each region when we finally embed detailed rectangles



## Routing: Global Routing

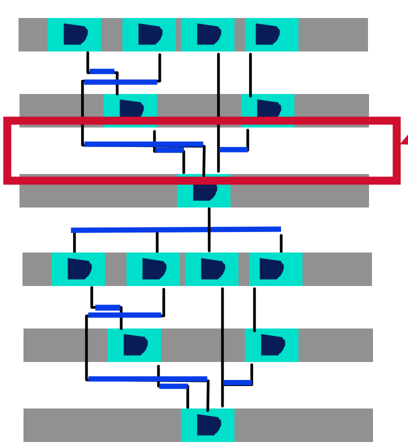
- **Result of global routing**
  - In each region of the chip, we know exactly which wires go thru that region, and we know roughly where the pin IOs are to enter and exit
  - Typical decomposition for ASICs is into *rectangular* regions, as below
  - In this example, signals only enter on the 2 opposite sides

One routing channel, with signals that use the channel known from global routing



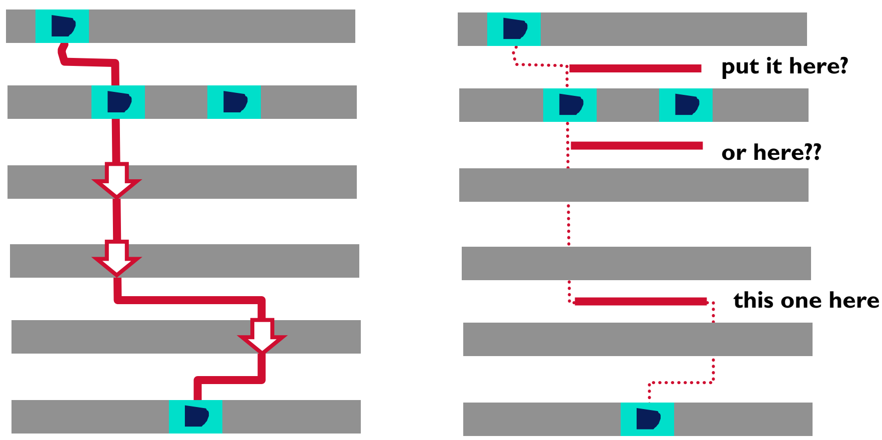



## For Row-Based Placements



- **Alternates logic & wiring**
  - Regions for wiring called channels, pins on top & bottom
  - Used when you have only 2 or 3 layers of metal wiring
  - Global routing determines where row-spanning signals cross the rows, and where the horizontal extent of signals are placed

## Global Routing for Row-Based ASICs




Determines where each row gets crossed by row-spanning signal

Determines which channels horizontal parts of signal will use, when there is choice

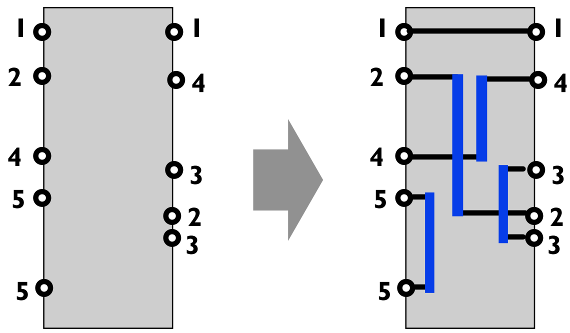


## Aside: Placement + Global Routing

- Smart row-based placers do some global routing
  - Helps decide if placement is good, by looking at where global routing wants to use space
  - Routing can make rows wider if you need to add space to let signals cross the rows (depends on metal layer, use of pins in cells, etc)
  - Routing can make layout taller if you need lots of tracks for wiring in each channel. If you make smarter decisions about where to put horizontal parts of the wiring in global routing, can get smaller layouts.
- How?
  - Can do some decent global routing inside an annealing-based placer
  - Start global routing near the end, when you have OK evolving placemnt
  - Can look at row crossings, predicted congestion in channels, etc
  - Try to evolve placement and global routing *at same time.*

## Routing: Detailed Routing

- *Detailed* routing follows global routing
  - Detailed here means "actually put down the exact final rectangles that make each individual wire"
  - In this case, you would use a *channel router*, which wires up a channel-shaped rectangle with pins on the 2 opposite sides

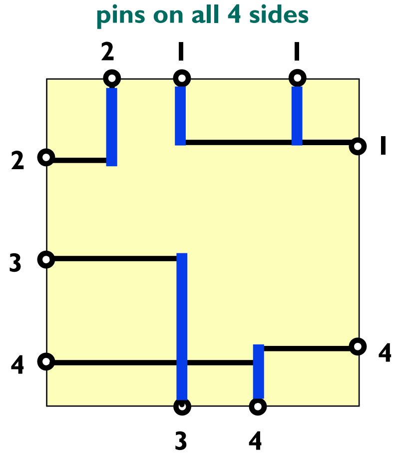




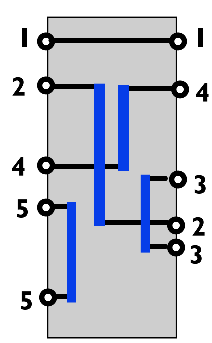

## Routing: Detailed Routing

- Different detailed routers exist for different region shapes:

*Switchbox router* for when rectangle has pins on all 4 sides

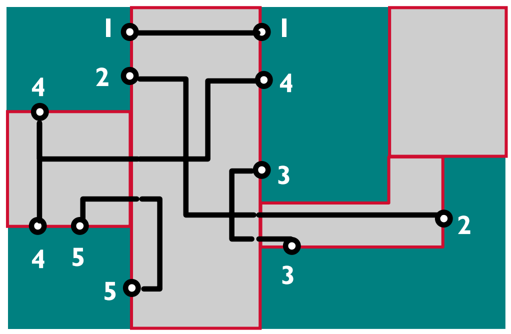



*Channel* router



## Routing: Global + Detailed

- Repeat for each region until the whole chip is routed.
- Does it always work...?
  - Nope
  - Often get some unrouted nets which require some rework by hand.





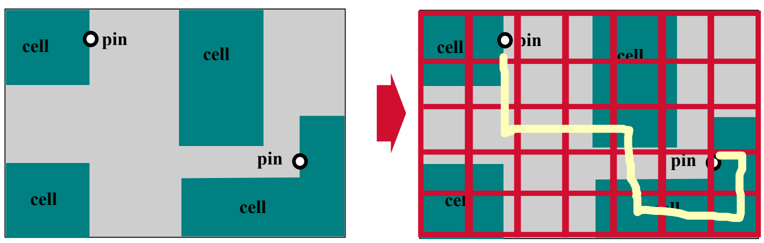

## Historical Tour

- Channel-ed layout styles
  - Dominated when we had 2-3 layers of available wiring
  - Disappeared when we got to 4, 5, 6, 7, 8… layers of wiring
- What's different now?
  - Route over the top of most placed objects, not "around" them
  - Get very *different* geometric models of global and detailed routing
- Interesting historical aside:
  - Earliest routers, for boards, viewed task as "one big routing problem"
  - They routed over the *entire* board area, routed *one* net at a time.
  - These are now called *area routers*
  - Area routers gave way to *region routers* (eg, channel routers) when we did chips with limited metal wiring layers
  - Now, we have lots of metal layers…we are back using *area routers* again

## Technology Marches On…

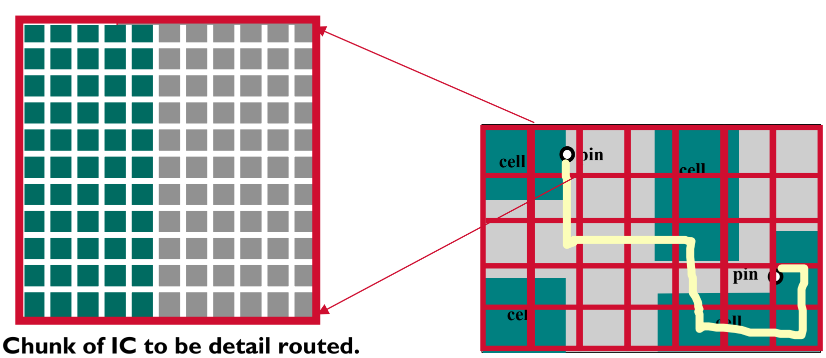
- Now have lots of layers of wiring.
  - Don't have to only put wires *between* the blocks of the chip
  - Now you can put wires *over* blocks of the chip
  - Area routers are designed to be good at dealing with obstacles.

Chop up chip, cells and all, into regions for global routing





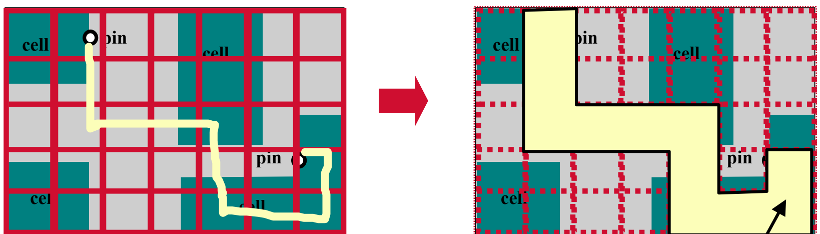

## Global Routing Today



Chunk of IC to be detail routed. Pins appear on boundary or anywhere inside the region. Typically modeled as a grid of legal wire locations called ***tracks***; typically 10-20 tracks in each dimension of one cell

# Routing Refinement Today

- **Global routing**
  - Track **supply** (how many available tracks) vs **demand** (how many paths want to go thru this cell in global grid)
  - Routing generates *regions of confinement* (ie, coarse path) for a wire

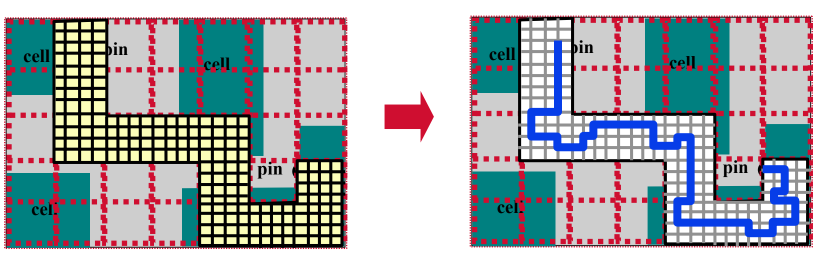



Global routing tell us we want this net to use *this* rough path; But we don't know the *exact* path in this region



# Routing Refinement Today

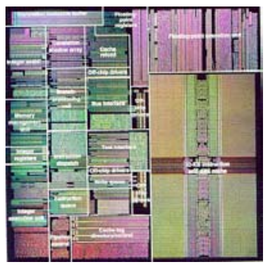
- **Detailed routing embeds exact paths in these regions**
  - Often insist on a **grid**: require wires and pin to use tracks **on** this grid
  - Tolerate off-grid pins: most geometry is on grid, fix-up exceptions




Global router tells us to search for detailed paths on these tracks in this region

Detailed router tells us exact final path in this region; may allow it to go off grid if needed

## Typical Problem



~ 10mm x 10mm

- Big ASIC chip today
  - ~8 layers of wiring
  - In 0.12um technology, 20 x 20 track unit cell is 10um x 10um box
  - 1cm x 1cm chip is 2000 x 2000 global routing grid
- Often, use yet *more* wiring hierarchy
  - Functional blocks get their guts wired up first, leaving wires between blocks unrouted
  - Then, we do "chip level assembly" routing, which just routes cross-chip global signals among blocks
  - This can *still* be ~ 100,000 nets
- Still a big, very hard problem

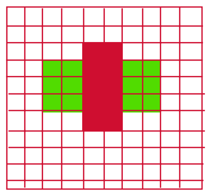


## 2 Remaining Problems

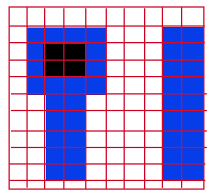
- Shape complexity
  - So far, pictures we have drawn have shown paths on nice, simple grids
  - It's not quite like that, if you look closely at modern technologies
- Timing complexity
  - ..and, we have not said anything about timing yet

## Shape Complexity: Modern Design Rules

- Lambda rules
  - Big idea in 1980: one fundamental distance unit -- lambda $\lambda$
  - All design rules are multiples of this unit
  - Allows process independent scaling of physical rules



Minimum CMOS FET, channel is 2x3



Minimum metal1 wires with minimum contact

  - In a $\lambda$ -scaled process, 1 $\lambda$ is the smallest physical unit of distance, for a width, height, separation, overhang, etc
  - And, $\lambda$ is "big", every size, distance is just a few $\lambda$s

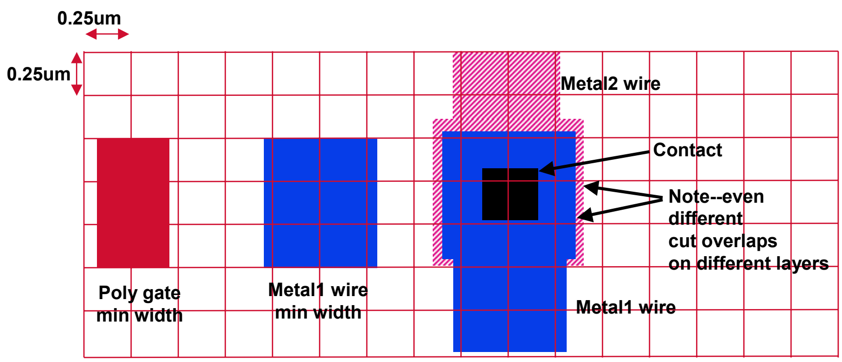


## Deep Submicron Design Rules

- Unfortunately, $\lambda$ rules don't work anymore
  - They were an OK approximation of industrial reality in 1980s
  - In the 1990s, things got difficult.
  - In later 1990s, these are not *remotely* close to reality
- Jargon: "*very deep submicron*" or more recently "*nanometer-scale*" design rules
  - Minimum size "thing you can draw" in a modern process is called the "feature size" of the process
  - Typically this is ~ length of the polysilicon gate on the FET
  - Submicron processes: this feature size is < 1 um
  - Deep submicron processes: this feature size is < 0.5 um
  - Very deep submicron processes: this feature size is << 0.5 um
  - Nanometer-scale processes: this feature size is < 100nm = 0.1um

# Deep Submicron Design Rules

- Consider a basic 0.25um process
  - If this was λ rules, every distance would be a multiple of 0.25um
  - *But it's not*
  - Everything is *uniquely* sized for fab yield and performance and density






# Deep Submicron Design Rules

- Manufacturing grid
  - Every edge of every rectangle must be on some fundamental grid, limited by the accuracy of the lithography--the optical printing of masks
  - In λ rules, λ is this grid, λ is big relative to feature size: min feature ~ 2 λ
  - In real processes, the mfg grid is very small, 1/10 or 1/20 or 1/50th of the feature size. Today, feature sizes can be 1, 5, 10 nanometers

- *Big* problem for routing
  - You cannot build and maintain a routing grid this fine
  - 5mmX5mm area at 10nm mfg grid = $500{,}000^2$ grid cells, 250 billion cells.
  - Purely gridded routing strategies stop working here...

## Timing Complexity

- Remains a huge, somewhat open problem
- *Not* uniquely a router problem
  - If placement is bad, you cannot get the wires short enough to meet timing requirements
    - *So you could improve the placement*
  - But, maybe the the placement is OK--the routing is not too good, so congestion forces nets to take long detours and be too long
    - *So you could improve maybe the global routing*
  - But maybe its really electrical crosstalk from neighbor wires screwing up the signals on your wire
    - *So you could improve detailed routing to move these wires away*
  - But maybe the layout is really as good as you can do--your logic sucks, its too big and its too deep and it uses slow gates
    - *So you could improve your logic synthesis--and start layout over?*
- Lots of messy, circular dependencies here; <u>active</u> problem



## So, What Will *We* Look At...?

- Classical maze-style area-routers
- Big characteristics
  - Area router, not a region router
    - So can handle obstacles well...
    - ..but they are sensitive to *order* in which wires routed.
  - Can use it for global and for detailed routing
    - …but we will focus mostly on detailed routing
  - Can handle messy gridless routing constraints
    - …but we don't have time to explain how (take 18-763)
    - We will do simple gridded style
  - Can handle some timing constraints
    - ...but we don't have time to do this. (take 18-763)
- Interestingly, this is a very old idea, yet hugely important still

## Routing: Maze Routers

- **Our Topics:**
  - History
  - Basic Mechanics
    - Two-point nets in one layer - unit cost
    - Multipoint nets
    - Multiple layers
    - Weighted cost
  - Design Variants
    - Vanilla scheme
    - Depth-first search (Rubin's scheme)



## Maze Routing: History

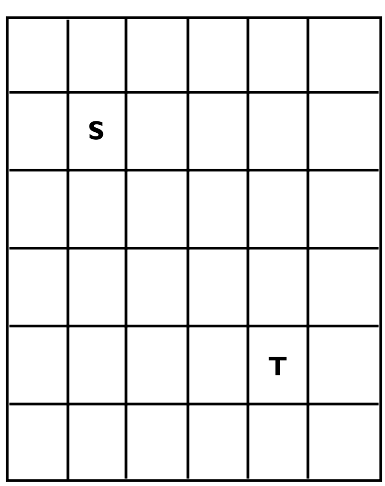
- **1961**
  - Lee, C. Y., "An algorithm for path connections and its applications", *IRE Trans. on Electronic Computers*, pp. 346-365, Sept. 1961.
  - Chester Lee of Bell Labs invents the algorithm; gets famous for "Lee routers"
- **1974**
  - Rubin, F., "The Lee path connection algorithm", *IEEE Trans. on Computers*, vol. c-23, no. 9, pp. 907-914, Sept. 1974.
  - Frank Rubin comes up with a way to make it go much faster.
- **1983**
  - Hightower, D., "The Lee router revisited", *ICCAD*, pp. 136-139, 1993.
  - Dave Hightower, who originally got famous for coming up with an alternative to the Lee-router (a Hightower line-probe router) that was faster and used a lot less memory, undergoes a spiritual conversion and implements a killer maze router. Trick is: now machines have enough (real and virtual) memory to do big maze routing tasks.

## Maze Routing: Strategy

- Strategy
  - One net at a time - completely wire one net.
  - Optimize path - find the best wiring path.
- Problems
  - Early nets wired may block path of later nets.
  - Optimal choice for one net may block later nets.
- Solutions
  - Careful net ordering.
  - Careful optimization to include impact on later wiring.
  - *No Guarantees.*
  - (How do people *really* do it today? Let router remove blocking nets or shove them aside; called ripup/reroute or shove-aside routing.)

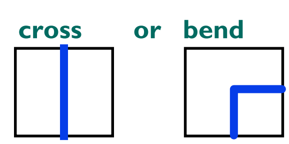


## Maze Router: Basic Idea



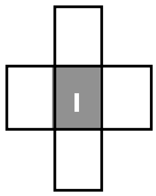

- Given:
  - Grid - each square represents where one wire can cross.
  - A source and target.
- Problem:
  - Find shortest path connecting source and target.

wires can:

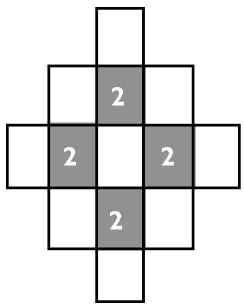


## Maze Routing: Expansion


Start at the source.




Find all new cells that are reachable at distance 1, ie, all paths that are just 1 unit in total length - mark all with distance.

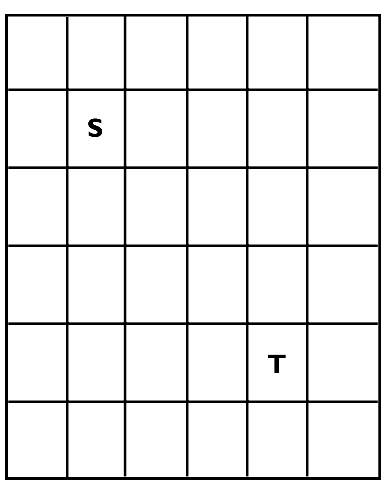



Using the distance 1 cells, find all new cells which are reachable at distance 2.

Repeat until the target is found.

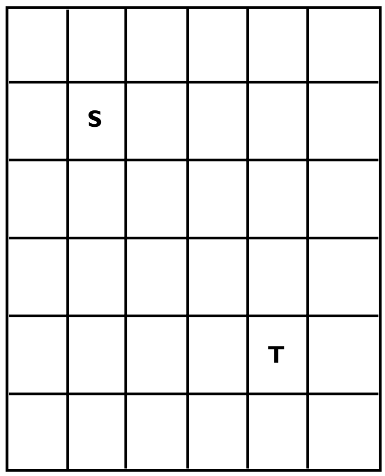


## Maze Router: Expansion




- Strategy
  - Expand one cell at a time until all of the shortest paths from S to T are found.
  - Expansion creates a wavefront of paths that search broadly out from source cell until target is hit

## Maze Router: Backtrace

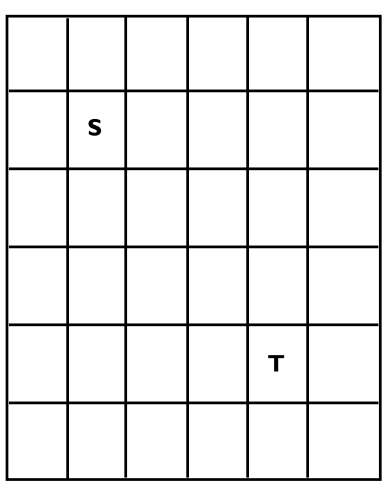



### Now what? Backtrace

- Select a shortest-path (any shortest-path) back to the source and mark its cells so they can't be used again.
- Since there are many paths back, optimization information can be used to select the best one.
- Here, just follow the path costs in the cells in descending order...



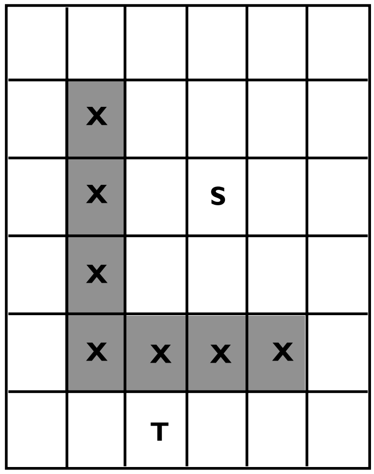
## Maze Router: Clean-Up




### Now what? Clean-up

- Clean up the grid for the next net, leaving the **S** to **T** route as an obstacle.
- Now, ready to route the next net with the obstacles from the previously routed net in place in the grid.

## Maze Router: Blockage




- Blockages
  - All future nets must route around this blockage.



## Classical Maze Router

- Three main steps:
- Expansion
  - Breadth-first-search to find all paths from source to target.
- Backtrace
  - Walk the shortest path back to the source and mark path cells as used.
- Clean-Up
  - Erase all distance marks from other grid cells before the next net is routed.

## Maze Router: Concerns

- Storage
  - Do we need a really big grid to represent a big routing problem?
  - What info required in each cell of this grid?
- Complexity
  - Do we really have to search the whole grid each time we add a wire?
- Technology
  - Just 1 wiring layer? How do we do 2 layers? 3? 4? 6??
  - Complex wire widths or spacings?

- 2 issues here
  - Applications of basic algorithm
  - Implementation issues for the basic algorithm

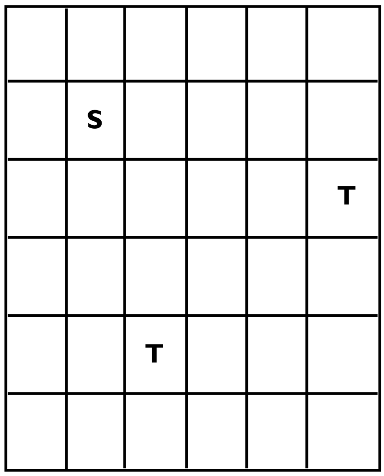


## Applications: Multipoint Nets

- Multipoint Nets
  - One source -> Many targets.
  - You get this with any net that represents fanout

- Strategy
  - Use maze route algorithm to find path from source to nearest target.
  - Relabel all cells on the path as sources and rerun maze router using all sources simultaneously.
  - Repeat for each segment.

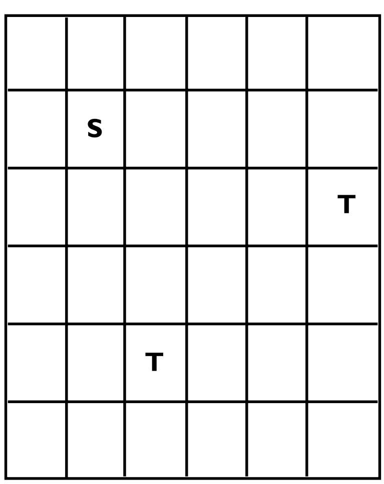
## Multipoint Nets




- **Given:**
  - A source and many targets.
- **Problem:**
  - Find shortest path connecting source and targets.

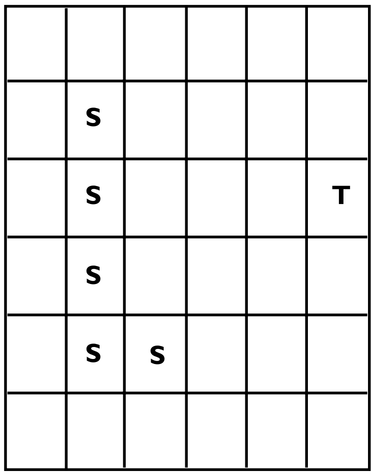


## Multipoint Nets




- **First…**
  - Run maze route to find the closest target.
  - Start at source, go till you find ANY target.

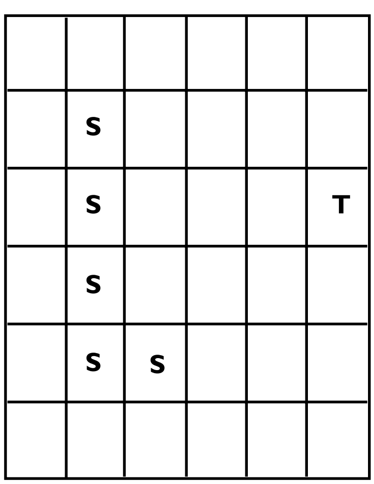
# Multipoint Nets




- Second...
  - Backtrace and relabel the whole route as sources for the next pass.
  - We will expand this entire set of source cells to find the net segment of the net
  - Idea is we will look for paths of length 1 away from this whole set of sources, then length 2, 3, etc.
  - Go till you hit another target

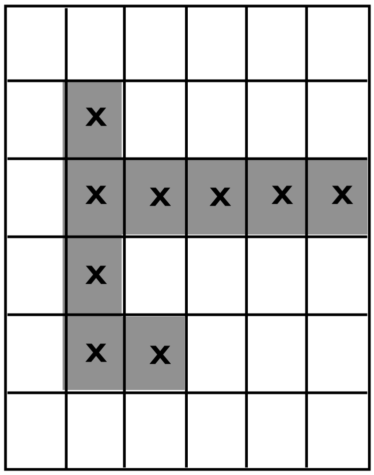


# Multipoint Nets




- Trick
  - Expand simultaneously from all of these sources to find the shortest path from the existing route to the next target.

## Multipoint Nets




- Finally
  - Do usual cleanup
  - Mark all of the segment cells as used and clean-up the grid.
  - Now, have embedded a multipoint net, and rendered it as an obstacle for future nets

© R. Rutenbar 2001

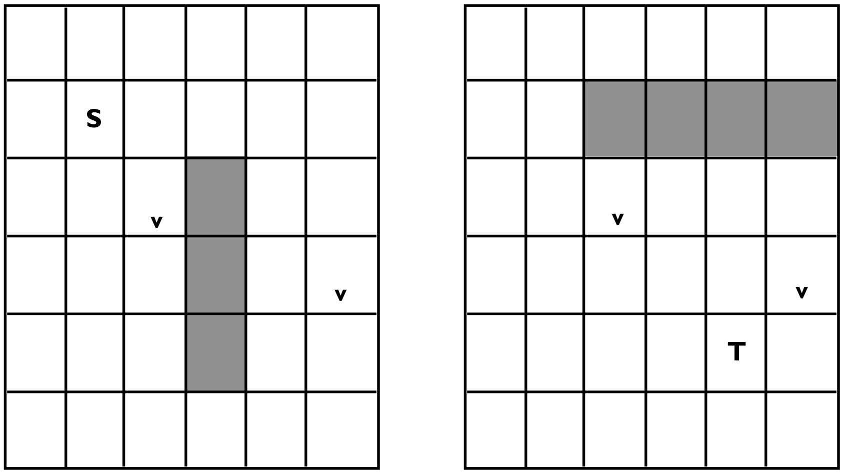

## Application: Mutilple Layer Routing

- How -- mechanically do we handle multiple layers?
  - Parallel grids, vertically stacked, one for each layer.
  - Use vias to access other layers.
  - Label cell as to whether a via is permitted at this location.
- New expand
  - Out on each layer,
  - Up/down at each via.

© R. Rutenbar 2001

# Multiple Layers

- 2 parallel grids, vertically stacked
  - **Expansion can now go UP/DOWN; vias can go where "v" mark is**

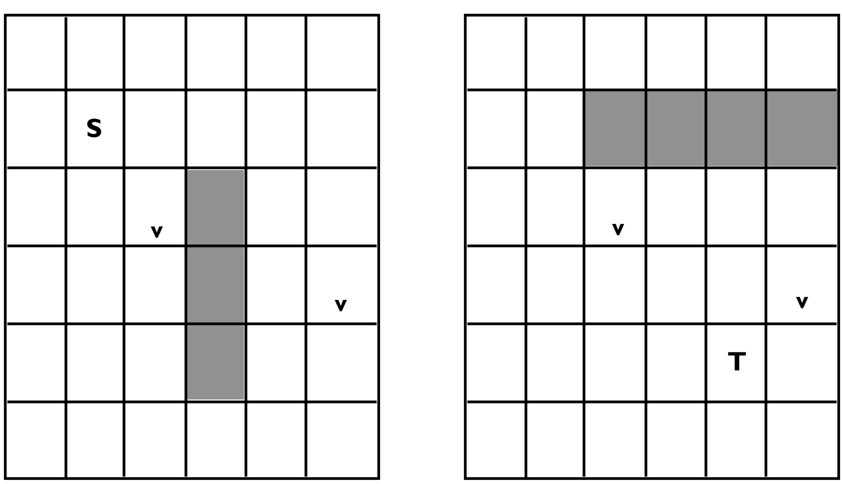



**Layer 1** **Layer 2**

© R. Rutenbar 2001


# Multiple Layers

- Backtrace & cleanup as usual

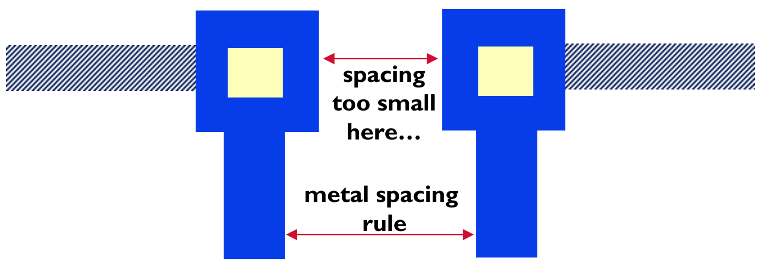



**Layer 1** **Layer 2**

© R. Rutenbar 2001

## Aside: About Vias

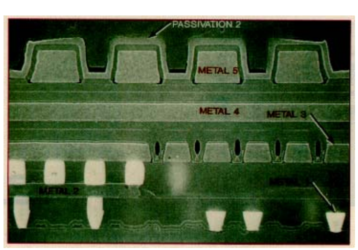
- Vias are how you move layer to layer
  - Electrical connects between separate layers of physical wiring
  - "Vertical" electrical connection
- Geometric issue #1: Size
  - On chips, vias may be wider than wire widths are
  - So you have to be careful where you assume you can put them
  - Since vias can "stick out," may force extra space between your wires



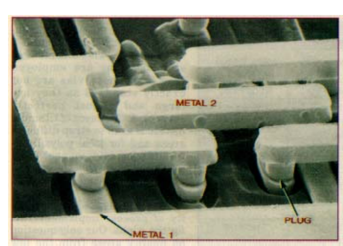



## Aside: About Vias

- Geometric Issues #2: Vertical stacking
  - Relevant in multi-layer metal process, and in PC boards
  - Can you put multiple vias connecting different sets of layers directly on top of each other, in a so-called **stack**?
  - In all modern processes yes, in older ones, no. Router has to handle this.

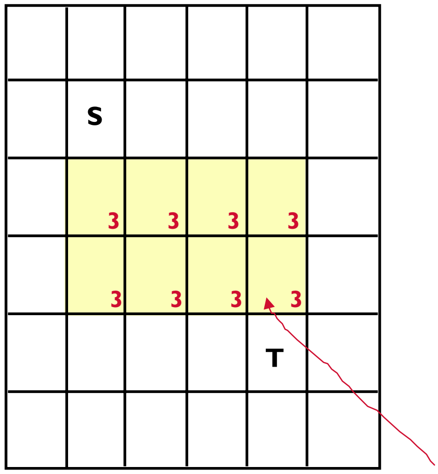



5-layer metal cross section from IBM PowerPC




Metal1 to metal2 vias

## Implementation Issues: Non-Unit Grids

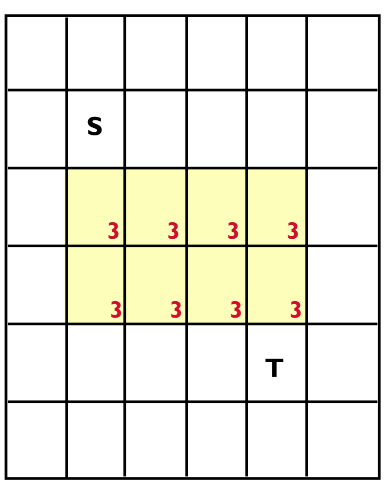



- Old problem
  - Each cell in grid cost the same to cross it with a wire
  - Cost ==1, unit-cost
  - Is this necessary?
- Now
  - Given grid, Source and target
  - Weights for each cell.
- Problem:
  - Find minimum cost path connecting source and target.

shaded cells cost 3

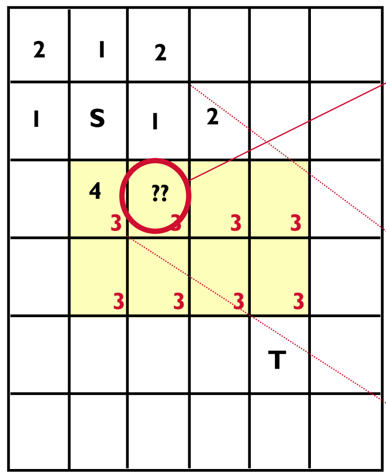


## Weighted Grids




- Many good applications
  - Make the router avoid electrically sensitive areas of IC
  - After global routing, weight cells with lots of potential wiring congestion higher, so router tries to avoid them
  - Can make different layers have different expense to use
  - Can make different vias have different expense to use
  - Can make different directions of expansion have different expense, eg, you want metal 2 mostly vertical, so left-right expansions cost more...
- Expansion...?
  - Always expand next cheapest partial path

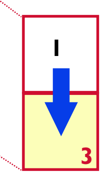
# Subtle Search Issues with NonUnit Costs




What cost does this get, and why?

Cost = 4 = cheapest path to this cell from the Source cell

We expand the cell to north to reach this cell, and we add this cell to the search wavefront at cost=4, reached from the north.




*Expand* this cell...

*Reach* this cell at cost = I + 3 = 4...



# Maze Routing: Mid-Point Summary

- What do we know?
  - Grid-based expansion, one net at a time
  - Can use costs in grid to get different effects
  - Can deal with multiple wiring layers, multi-point nets
- What don't you know?
  - Real implementation issues
  - Data structures for grid, for the search
  - Depth-first expansion techniques for speed
  - Subtle interactions between cost strategy and search strategy
  - Expanding a cell vs reaching a cell vs multiple-reaching....
- Next topics: *serious implementation issues.*

## Implementation Concerns

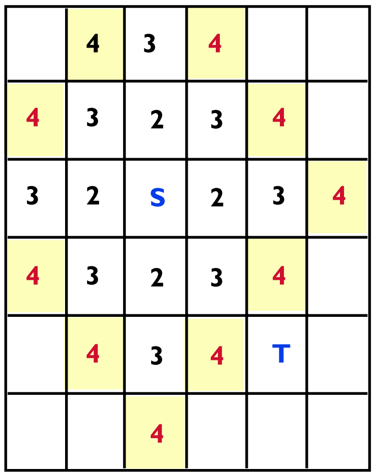
- Representation
  - How do we store the routing grid?
  - What do we need in each cell?
  - How do we represent the state of the advancing path search process?
- Algorithms
  - We have a serial computer: we can process one cell at a time...
  - ...so, which cell is next to "label" in the search process
  - Does the order matter?
  - How can we do this as fast as possible?



## Big Idea: Search Wavefront

- One big goal
  - Efficient storage: big layout needs a big grid
  - Want to put as little in each cell as possible
- Idea: Don't actually store path costs in grid cells
  - Big costs -> many bits per cell.
  - Only the cells most recently labeled during search will be used to expand the search for new paths
  - These cells constitute the search wavefront
- *Wavefront* is important:
  - Store wavefront list with all needed info about each wavefront cell.
  - Mark a few bits in the actual grid cells just to indicate how you found the path to this cell - i.e., remember the predecessor cell.

## Example Wavefront for Simple Search

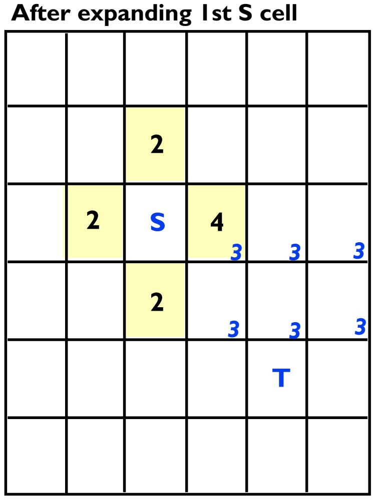



- Wavefront is...
  - The frontier of the active search for new paths
  - The neighbors of the new cells worth looking at to try to extend the evolving path search
  - The only cells we need to look at to decide how to continue the search process
- Implication
  - Don't store the path cost numbers *in* the grid
  - Just store the wavefront cells themselves in a special data structure

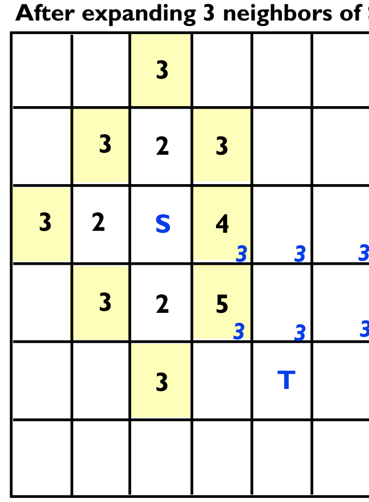


## More Complex Wavefront

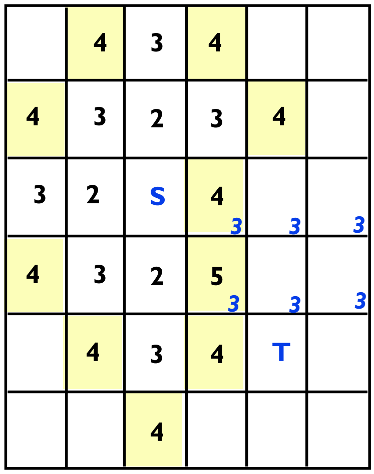
After expanding 1st S cell




After expanding 3 neighbors of S



## More Complex Wavefront




- **What wavefront is…**
  - Set of cells already reached in the expansion process…
  - …which have neighbors we have not already reached
  - Indexed by cost of cells reached (== costs of paths that start at source and end at this cell)
  - Expanded in cost order, cheapest cells before more expensive cells



## Outline of Expansion Algorithm

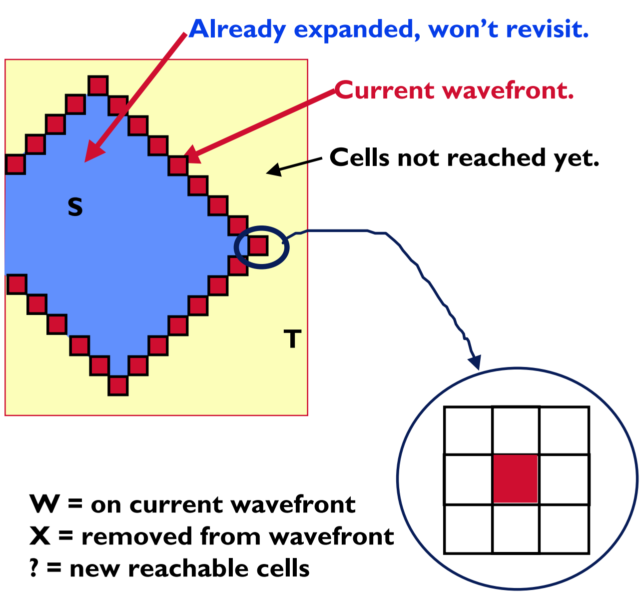
- **Cheapest-cell-first search**
  - Variant of Dijkstra's algorithm
  - Assume wavefront is a cost-indexed list of cells you have already visited during search process, and "labeled" with path cost
- **How does the wavefront grow?**
  - Pull out a cheapest cell C from the wavefront
  - Look at the neighbors N1, N2, … of cell C you have not visited yet
  - Compute the cost of expanding this path to reach these new cells N1…
  - Add these new cells N1, N2, … to the wavefront data structure (indexed by their cost)
  - Remove cell C from the wavefront
  - Repeat with the next cheapest cell on wavefront…

# Maze Router: Terminology

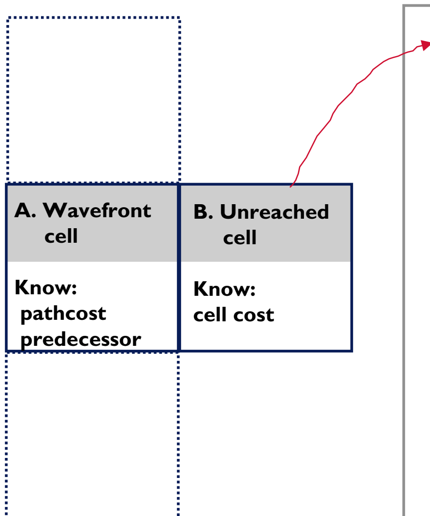
- We need some *terminology* or we'll get confused
- Wavefront
- Reached
- Expanded
- Dijkstra's Approach



# Illustrating the Terminology



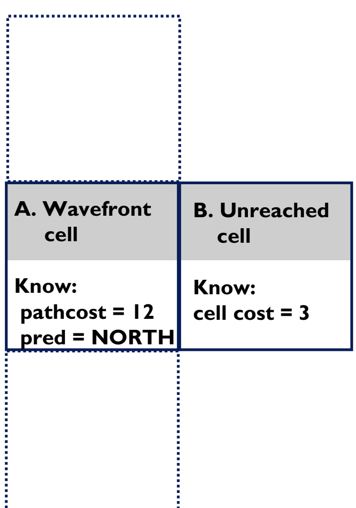
## Reaching a New Cell During Search




To reach this cell:



## Reaching A New Cell




- To reach cell B
  - Grab cell A, pathcost=12, from wavefront
  - See unreached neighbor B
  - Compute cost to reach B is 12 + 3 = 15
  - Add this cell to wavefront
  - Mark grid cell B as "reached" (only takes a few bits) so we don't try to put it on wavefront again (ie, reach it again)

## Basic Maze Routing Algorithm

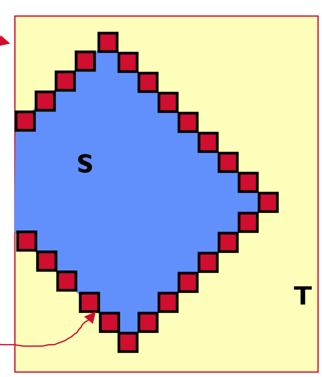
```
wavefront_structure = {  source cell }
while (we have not hit target) {
  if ( wavefront == empty )
     quit -- no path to be found

  C = get lowest cost cell on wavefront_structure
  if ( C == target )  }
     backtrace path in grid
     cleanup
     return -- we found a path
  }
  foreach ( unreached neighbor N of cell C ) {
    mark N cell in grid as reached
    compute cost to reach it = pathcost of C + cellcost of N
    mark N cell in grid with predecessor direction back to cell C from N
    add this cell N to wavefront
  }
  delete cell C from wavefront
}
```



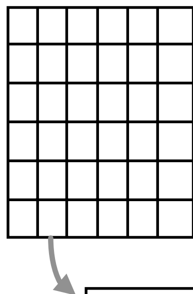
## Data Structure Issues

- 2 key structures
- Routing grid
  - Hold cells of area to route, costs of each cell, blockages
  - Mark these cells to know what cells you have already reached
  - Mark predecessor in here too
- Wavefront
  - Hold active cells to expand
  - Cell info has pathcost, predecessor information
  - Indexed on pathcost
  - Always expand cheapest cell (ie, cheapest partial path) next



# Data Stucture Implementation

**Grid of cells:**

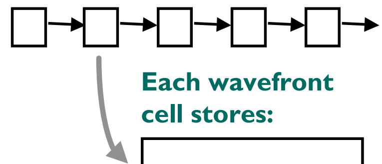


**Each grid cell stores:**

- int *cellCost*
- char *predecessor*
- boolean *reached*

**A 2-D array is just fine here**

**Wavefront list:**



**Each wavefront cell stores:**

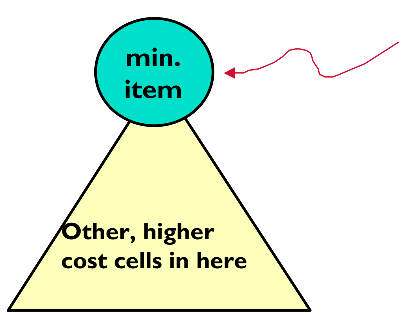
- grid **whichCell*
- int *pathCost*

**Need something clever here-- want fast insert/delete in cost here. Dumb linked list isn't going to be fast enough...**



# Wavbefront Option: Heap

- Store cells of wavefront in a heap
  - (also called a *priority queue* -- consult your favorite data structures book)
  - Classical data structure designed for fast insertion and retrieval of lowest cost data item.
  - All ops (add, delete, etc.) have O(logN) time complexity for N objects.
  - *Most routers do it like this.*



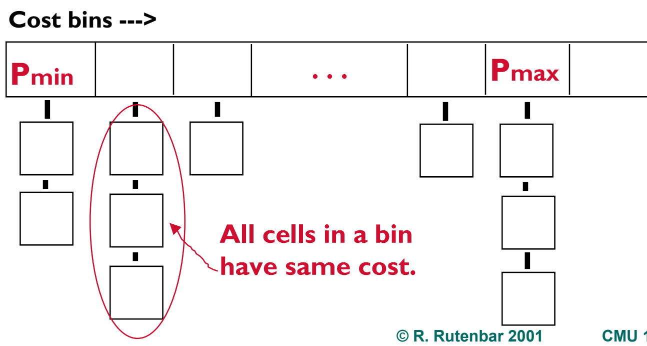

min cost item is always at the top.

Insert "bubbles" the items in heap around to ensure the cheapest item always on top.

Ditto for delete.

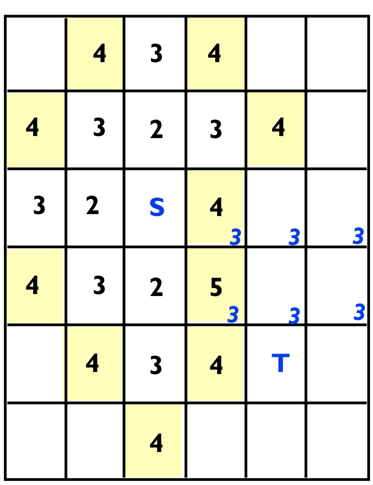
# Wavefront Option: Cost-Indexed Array

- **Basically a big hash table**
  - Index is path cost
  - In each bin of table we insert the cells in wavefront at that cost
  - Get almost constant-time insert/delete
- **Problem**
  - It's a BIG table -- 1 bucket for each possible path cost.






# Rememember How Expand Algorithm Works...

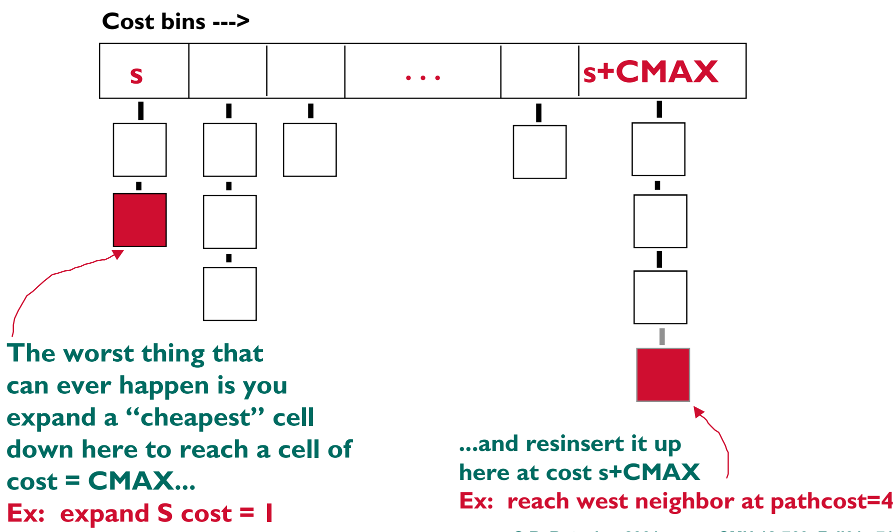



- ***Always* expand next cheapest cell**
  - Implication is you cannot have an arbitrarily wide "spread" in the values of "live" cells on wavefront
  - If CMAX is the maximum cell cost value you can see in the grid
  - ...then the biggest spread of cost values you can see in the wavefront is CMAX + 1
- **In this example**
  - Biggest spread in costs you can see is 4, since biggest cell has cost=3,

    3 + 1 = 4 different cost values possible in wavefront at any time

# Remember How Expand Algorithm Works

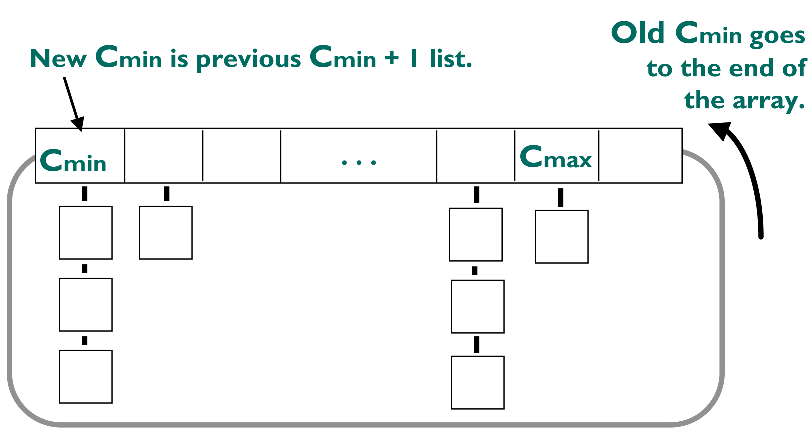
- Said another way…






# Wavefront Implementation

- So, you can do this with only CMAX+1 bins in array
  - It's a *circular* array: when you empty the lowest cost bin, it means you can reuse it for the *next* MAX pathcost value you need.



## Wavefront Data Structures

- Historically, saw both kinds of structures in real routers
  - Hashed cost array
  - Heap-based
  - Today, heaps seem to be dominating; just a lot more flexible on what they allow you to do with costs
- But, there are still some subtle interactions with the search algorithm to discuss
- Question
  - *What constraints have to met for this simple expand-cheapest-cell-next strategy to get the best path?*



## Plain Maze Routing Revisited

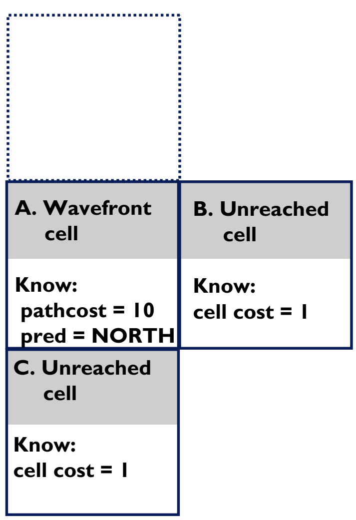
- Key assumptions
  - Always expand cheapest cell next
  - Expand each cell just once
  - Reach each cell just once
  - Guaranteed to find the min cost path (we hope...)
- Question rephrased
  - *What are the constraints on the cost function used for paths (ie, pathcost of a cell as it is reached) so that above stuff holds...?*

## Pathcost Constraints

- Basic constraint: *Consistency*
  - The cost of adding a cell to a path (reaching a cell) is independent of the path itself
  - It does NOT matter how you reached this new cell, it still adds the same cost to the path
  - Guarantees we reach it once (from a cheapest path) and thus expand it just once
- It's actually easy to create a cost function that is *inconsistent*, and violates all these nice properties



## Inconsistent Cost Function

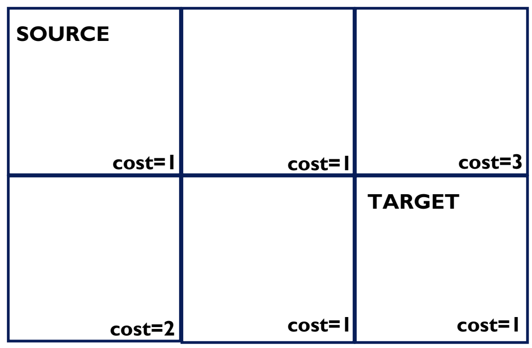



- Penalize paths with *bends*
  - Still store 1 cost inside each cell
  - But, now add another cost when you reach a cell that requires a turn (a bend) from the direction that reached the expanding cell

Suppose bend penalty = 2

## Inconsistent Cost Function

- Try this example with bend penalty = 2
  - Don't mark the "reached" bit in each grid cell when you reach the cell
  - Allow search to revisit previously reached cells...






## Inconsistent Cost Example

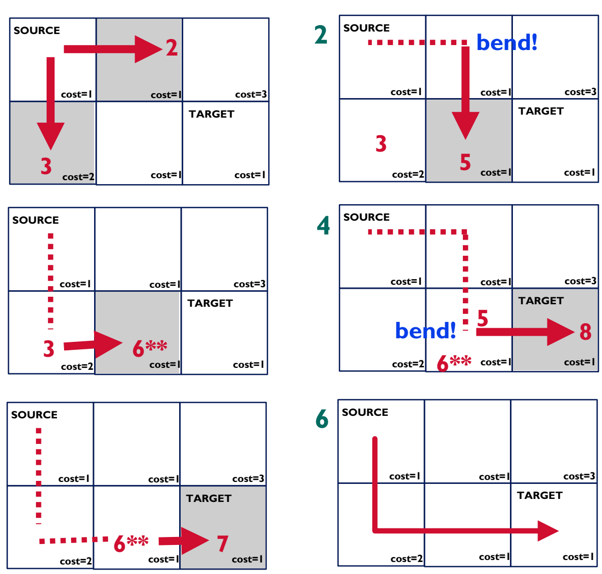
1

3

Revisit this cell, reach it again at more cost

5

Revisit target, but it's cheaper now



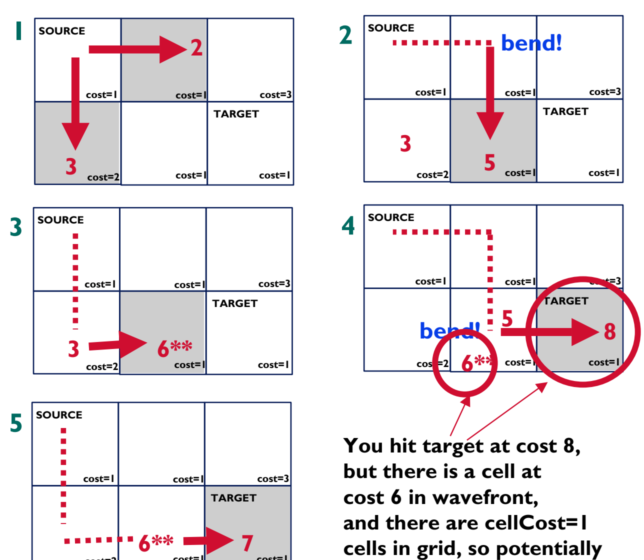
# Inconsistent Cost Function: Implications

- Notice what happened
  - Reached same cell, later, at a higher cost, but it was ultimately on the cheaper overall source-to-target path
- Implications
  - You will reach cells multiple times at different costs.
  - You will have same cell in wavefront multiple times at different costs.
  - Cannot guarantee you need only CMAX+1 hash bins in array
  - Can still expand cheapest first, but cannot quit when you reach target
- Termination of search?
  - Cannot quit until each cell in wavefront has a cost so big that it is NOT POSSIBLE to reach target any cheaper than current cheapest path
  - May reach, expand lot more cells with an inconsistent cost function...
  - ..but you can do a lot of cool things with such functions



# Termination of Search: Close Up Look

- Cannot quit until no cell in the wavefront has a cost that could lead to a cheaper path to target




You hit target at cost 8, but there is a cell at cost 6 in wavefront, and there are cellCost=1 cells in grid, so potentially possible to hit target at cost=7

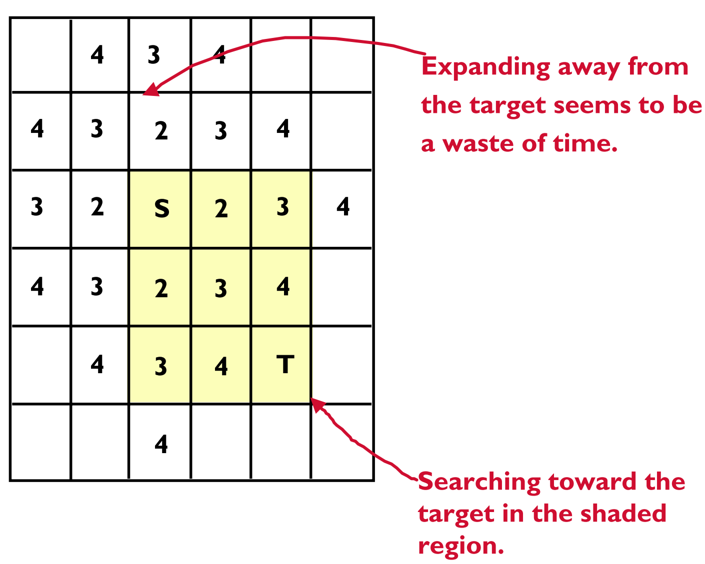
# Expansion Process, Revisited

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

- **Questions:**
  - How do you search **toward** the target?
  - Can we do this and still keep guarantees of reaching the target with the minimum cost path?



# Motivation for Smart Search

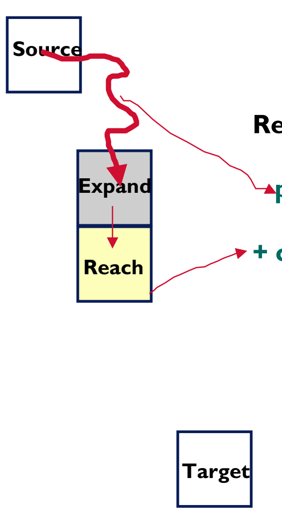


## Smarter Search: Rubin's Scheme

- Two parts:
  - Add predictor function to the cost.
  - Direct the search toward the target
- Plain maze router
  - You add a cell to the wavefront with a cost that measures partial cost of the path, source-to-target
- Rubin's Scheme
  - You add a cell to the wavefront with a cost the estimates the entire source-to-target cost of the path
  - Trick: estimate this as pathcost(source to cell) + predictor(cell to target)
  - (We will see this exact same idea again, when we do Static Timing analysis; this predictor will be called the ESPERANCE of a path…)



## Plain Maze Routing

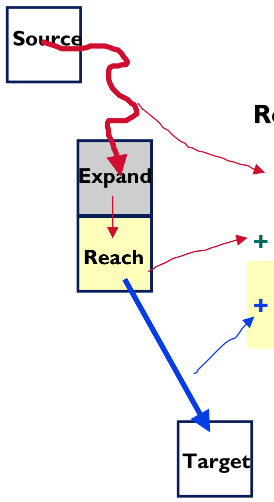



Reached cell goes on wavefront with this cost

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(reached cell to target)

A typical estimator is:

min cell cost * distance to target

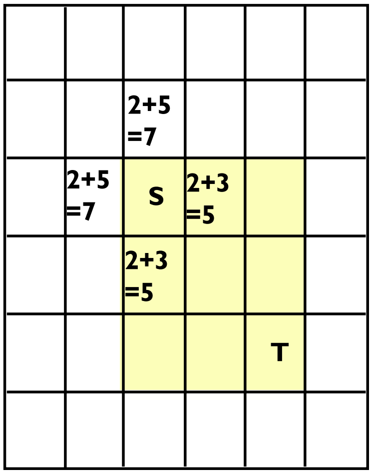
© R. Rutenbar 2001 

## Technical Results

- Depth first predictor
  - If the predictor is always a lower bound on how much pathcost you will really add to get to the target…
  - …you will still get the min cost path, guaranteed
- 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
  - It does this in a very geometrically stylized way
- Look at an example…

© R. Rutenbar 2001 

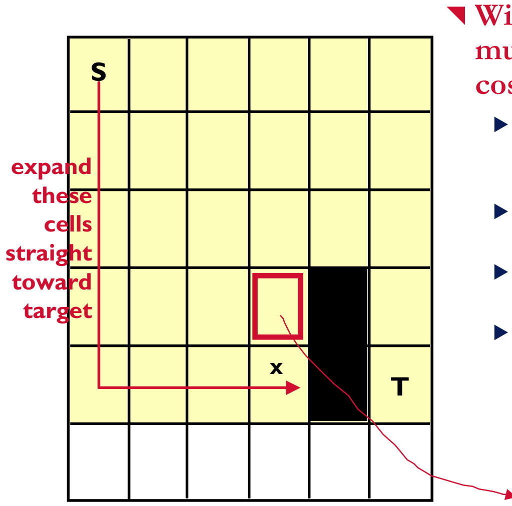
## Rubin Expansion Example




- **Observe**
  - It prefers to stay inside the bounding box of the source-target rectangle before it expands other cells.
  - How do we know which cell to expand in order inside the box?
  - Several heuristics which all basically say: don't turn unless you have to, and prefer to expand the cells that are actually closest to target, first



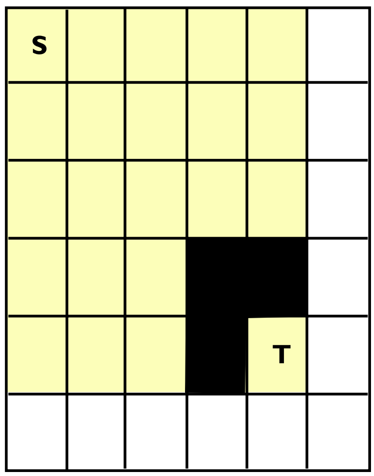
## Some Subtleties




- **Will again reach a cell multiple times with different costs**
  - Suppose you really expand cheapest first, and among those of same cost, closest to target
  - Can shoot directly toward target…
  - …but you reach cells early with suboptimal costs
  - Can reach time again later

You will first reach this cell as NORTH neighbor of cell x, but cost will be big because your path goes thru cell x, which is bad path here

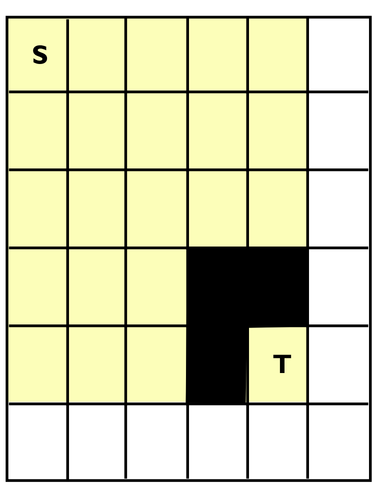
## Rubin Expansion Example




- Works great...
  - Until you get a case like this with the target blocked inside the source to target rectangle.
  - Problem now is that it explores the whole rectangle before it tries anyplace else.
  - Might it not be faster to search outside the rectangle if the rectangle is VERY big...?



## Another Tweak




- Can insist that cells closer to target *always* be cheaper
  - Add cell to wavefront as

    pathcost(src -> cell)

    +cost of cell

    + K • estimated cost(cell -> target)
- K is just a fudge factor
  - Forces cells closer to target to be cheaper.
  - Typically small (like 1.1)
  - Try K=2 in this example for effect
- Effects
  - Faster search, smaller search, but lose guarantees of minimum soln

## Area Routing By Maze Routing:  Summary

- Been around a *long* time
  - Very flexible cost-based search
  - Extremely flexible, can be recast to attack many problems
  - Zillions of tweaks for speed, space, etc.
  - Still widely used, but now often with rather more sophisticated representations of "space" than a 2D grid to handles gridless cases

- Remaining problems
  - Still routes one net at a time.  Early nets block later nets.
  - Lots of iterative improvement strategies here (I didn't talk about)
  - Great if there IS a path; if not, will spend a long time to prove to you that there is NO path