18-760 Spring99 VLSI CAD Project 3: Global Routing

Out: Tue Apr 6, 1999. **Due: Thu, Apr 29, 1999**

1.0 Background

In Fall97 (the last time we taught 760) Project 3 was an annealing placer. We supplied netlists of connected gates, IO pins and wires (nets), and asked students to write code to place the movable objects on a coarse grid. To keep the complexity down, we used a very coarse grid, and allowed multiple gates to be placed at each grid cell, up to some capacity limit. For example, a 1000 gate netlist could have been placed on a 10x10 grid, with each cell allowed to hold 10 different gates.

This semester, you will write a companion **router** for these layouts. You will be using maze routing ideas as from the class notes, but since each "cell" of the grid can hold more than one gate, it can also hold more than one wire. Routers that work on this sort of task are called **global routers**. A "global" router does not embed the exact wire geometry of each net. Instead, it "plans" the path for a net, through a set of adjacent cells on this coarse grid, each of which has some capacity to hold multiple wires.

Your assignment is to implement a program that can take these 3 inputs from us:

- A **netlist file**, which describes all the gates, IO pins, and wires. This is actually the input to the Fall97 placer program.
- A **placement file**, which describes where each gate and IO pin is located on a simple grid. This is actually the output of the Fall97 placer program.
- A **routing capacity limit**, which is just two numbers for a formula that tells how many wires can actually fit in each cell in your grid, as a function of how many objects have been placed in that grid cell.

...and then **route** each net on this grid. You will then write a 3rd file as output, the **routing output file**, which tells which grid cells each wire uses.

Be sure to go look at the writeup for **Fall97** Project3 (also on the web site) to see how the placer worked. We use the same file formats and geometric model. Also go look at **/afs/ece/class/ee760/proj3** for benchmarks and instructions on what you need to run.

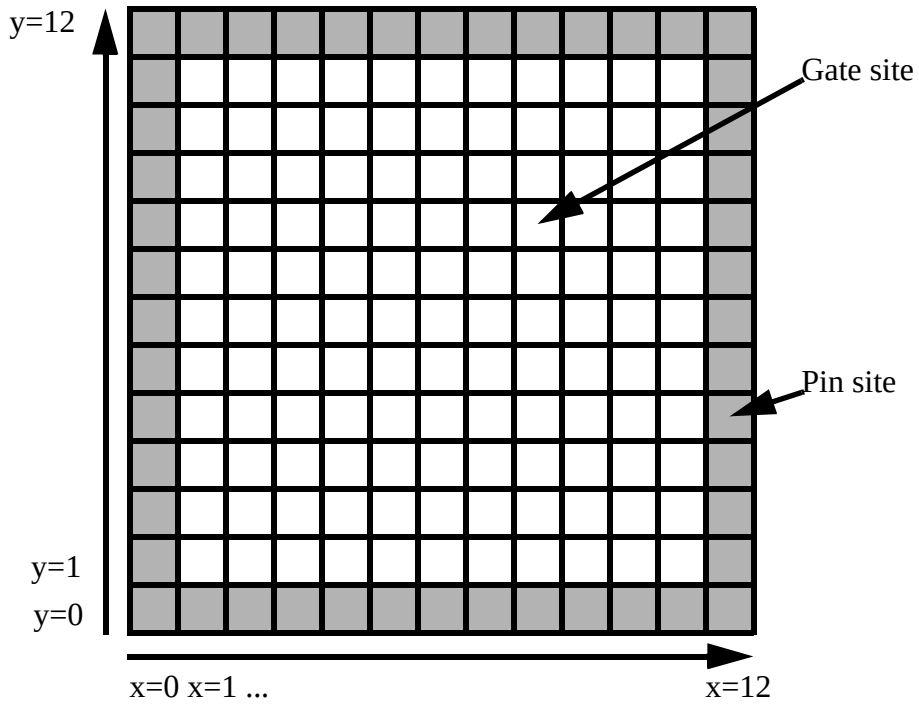
2.0 Modeling the Placement

We describe here the geometric model of the *placed chip* itself, since your router starts with a placed netlist.

2.1 Geometric Model of the Placement

For simplicity, we shall model all geometric objects in this problem as being rectangles whose dimensions are multiples of some unspecified “fundamental unit”. Everything looks like a sort of simple checkerboard, where the individual squares have certain properties.

In our model, the overall placed chip is a rectangle with *pin sites* at its periphery, as shown in the figure below.



Individual gates drop onto this grid at arbitrary locations (see next subsection). The chip itself has pins, but they are restricted to be around the outside of the rectangle. Our model uses a *coarse* grid--it is a sort of detailed partitioning model and not a complete, exact placement of each gate. We allow each gate site to hold some maximum number of physical gates. This number is the *capacity* of the site. We also allow each pin site to hold more than one physical pin. The gate capacity (how many gates can be put in a gate site) and pin capacity (how many pins can be put in a pin site) are part of the input netlist file. This is information the placer uses to decide where to put the gates and the pins.

In the example on this page, the chip is 13 fu X13 fu (fu=fundamental unit), with (x=0,y=0) modeled as being the cell at the lower left. There are on this chip 121 gate sites and 48 pin sites. If we allowed 10 physical pins per pin site, and 100 physical gates per gate site, this chip could have 10x48=480 actual pins and 121x100 = 12,100 gates if it was fully populated.

2.2 The Gates and IO Pins

For our problem, gates and IO pins are the placed objects. The placer starts by reading a large netlist file specifying gates and nets and pins. The placer then places gates in gate sites, and pins in pin sites, trying to achieve two goals:

1. **Minimize the wirelength:** We use the typical half-perimeter of the smallest bounding rectangle around all the placed terminals as the metric. (There are a few technical details to this, since many gates can go in one site; see the Fall97 writeup).
2. **Respect the capacity limits:** Our model of the chip allows multiple gates at each gate site, and multiple pins at each pin site. The way we suggested to do this with an annealer is to allow an *arbitrary* number of objects at each site, but add a term to the annealing cost function that adds in a *penalty* that is proportional to the amount of violation (overcapacity) at each too-full site. So, if the capacity is 10, and a site has 9 objects, it gets no penalty. If it has 11 objects it incurs a small penalty. If it has 100 objects it gets a BIG penalty.

Each gate is assumed to be a “small” unit-sized object. Each gate can be connected to several nets. Pins are likewise unit-sized objects that can be placed in pin sites. Most pins connect to only 1 gate but in general a pin can connect to several gates.

3.0 Modeling the Global Routing

3.1 Overview

See the Fall97 writeup for a discussion about how we modeled wire *length* in the placer. For your router project, the goal is to actually create a *path* for each wire, while again respecting a set of appropriate routing capacity constraints.

The basic global routing problem looks like this:

- **Nets**: you have to wire all the nets specified in the input netlist file. The placer has arranged the gates and (maybe) IO pins that are the terminals of this net on the coarse placement grid. As far as you are concerned, you just have to route each wire so that all the grid sites that hold its terminals are connected.
- **Capacity:** each cell in the grid has a capacity, which is the number of wires that can be routed through that cell. This capacity number is a function of the number of placed objects in that cell. So for example, if a grid cell has a capacity to hold 10 gates but 0 gates are placed there, it has *a large* capacity for wires. But if it can hold 10 gates and 9 gates are placed there, is has a *smaller* capacity for wires. Note that capacity also gets **updated** after each wire is routed. If we have a cell with a capacity of 10 wires, and we route a wire through it, then on the next net we route, this cell has a capacity of only 9 wires.

- **Routing cost:** the way that routing capacity information gets used in a maze router is to translate the capacity info into a **supply-and-demand** cost model. In a maze router, each cell in the grid can have a cost, which can be used to preferentially avoid certain grid cells (by giving them a high cost). In a global router, you use this idea by giving cells that have many wires through them, that are close to their capacity limit, a BIG cost. The idea is that you want the next path to avoid this cell if at all possible, and only use it if necessary. A cell that has few wires through it, and is not close to its capacity limits, gets a SMALL cost; it is OK to let the next path use this cell. Note that after each net is routed, all the costs in the grid get updated to reflect the altered capacity of each cell.
- **Obstacles**: in this form a routing, we don't really have "hard" obstacles. We only have cells that are over their capacity for wiring. For example, suppose a cell has a capacity for 20 wires, and your router has already put 20 wires through this cell. Do we get to route a 21st wire through this cell? **You** get to decide if you want to allow more wires to go in a cell that is at its capacity limit. One option is you give it a BIG cost, but you don't prevent the router from using the cell. Another option is you can treat this cell as an obstacle and prevent any subsequent wires from using it.

3.2 Modeling One Wire's Path

After placement, each gate and each IO pin is placed somewhere on the grid. So, the router must connect each wire by finding a path though the cells on the grid. The big difference between the lecture notes on maze routing and this router is that this one allows multiple wires to use each cell.

Let's look at 2 examples (see next page) to see how individual wires can use the grid cells to route. Suppose we have two nets in a simple problem:

- **Net N** is connected to 5 gates, numbered 1,2,3,4,5. Net N is also required to be connected to the chip pin "0" marked by the circle in the top row.
- **Net M** is connected to 3 gates, numbered a,b,c. It is not connected to any chip pin:

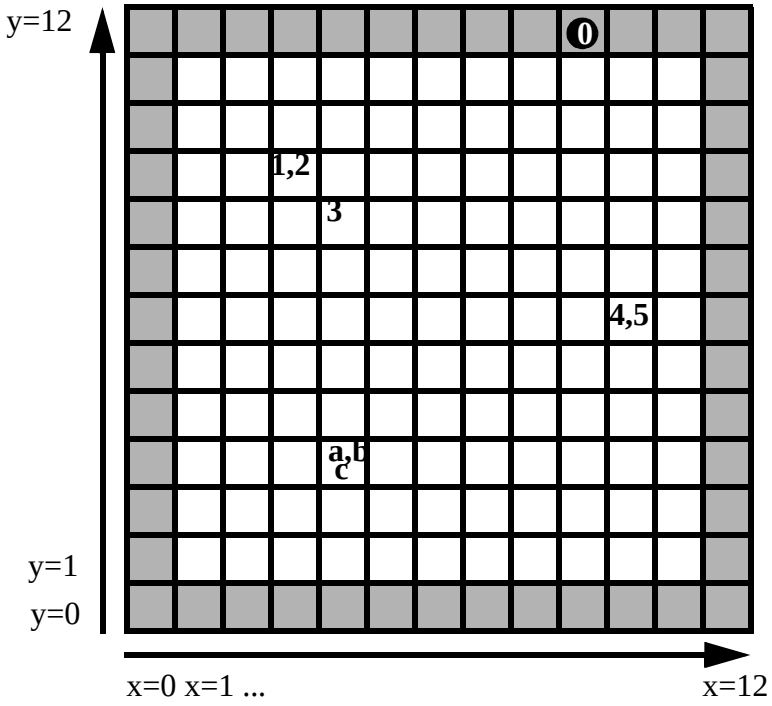
Consider net N first. Although it connects 6 objects (5 gates plus 1 IO pin), notice that the path we find for this wire only connects 4 different grid cells. This is because we assume that the wiring for the cells which hold multiple placed objects will take place "inside" those individual cells--so, we don't have to worry about it.

The extreme case of this is net M. All the objects to which net M is connected live in a single grid cell, so the ultimate "path" here is just this one cell. We refer to this as a "trivial" net. But note, you may see a lot of trivial nets in our benchmarks, since a good placer should be able to cluster connected gates in a single grid cell.

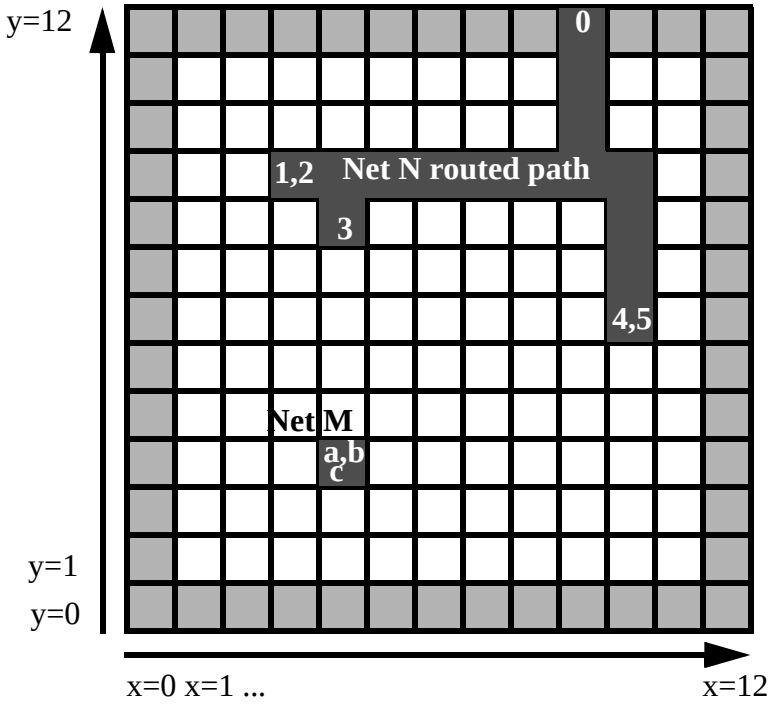
So, the summary about the geometric path for each wire is this:

- Before you start routing, make sure you "collapse" the terminals of each net down to just the unique grid cells, and route to only connect those grid cells.
- If your collapsed terminals are all in one grid cell, you're done--this is a trivial route and you don't have to do any real routing work here.

Gates and IO Pins for 2 different nets



Possible routes for the 2 nets



3.3 Basics for a Global Router

The big difference between a *detailed* router (like we did in the lecture notes) and a *global* router is that each cell is allowed to hold multiple wires. So, the fact that you used a cell to put a wire in does *not* mean that the next wire must avoid this cell. It just means you have to assign a sensible cost to each cell that reflects the supply of wiring space that *remains* inside it. The mechanism for doing this is to give each cell a cost this is inversely proportionally to how much "space" is left in the cell for future wires to use.

Let's look at a very simplified example, routing a few wires (**A B C**) in a very small 5x5 grid. Let's assume each cell has a capacity of 2 wires, and that the cost of each cell varies with the number of wires already using the cell, like this:

- If 0 wires are using this cell, the cost is 1
- If 1 wire is using this cell, the cost is 5
- If 2 wires are using this cell, the cost is 100; cost goes up by 20 for each succeeding wire.

Net A: we first route net A. Before we route it, we make sure to set all the cell costs. Since this is the start of the routing, all cells cost 1. The terminals of net A are shaded at left. Since there is really nothing to stop it, the net takes a very straightforward path during routing. Note that all the cells on the path get their cost updated after routing. These costs reflect the fact that we would rather not use these cells if don't have to, since they each have space for just one more wire.

Cost before routing A

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Path for net A

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Cost after routing A

1	1	1	1	5
5	5	5	5	5
1	1	1	1	5
1	1	1	1	5
1	1	1	1	5

Net B: we next route net B. Again, before we route it, we make sure we have updated all the cell costs. Terminals are again shaded at the left. Notice now that not all cells cost "1" anymore. And, some of the terminals are in cells that cost 5; this is always possible, since there is more than one gate in each cell of the grid, and also the gates have multiple terminals themselves.

Cost before routing B

1	1	1	1	5
5	5	5	5	5
1	1	1	1	5
1	1	1	1	5
1	1	1	1	5

Path for net B

1	1	1	1	5
5	5	5	5	5
1	1	1	1	5
1	1	1	1	5
1	1	1	1	5

Cost after routing B

1	1	1	5	100
5	5	5	100	5
1	1	1	5	5
1	1	1	5	100
1	1	1	1	5

Observe that the path for B is not the “obvious” straight vertical connection. The reason is that that path costs too much: it costs 5+5+5+5+5=25. The reason is all those cells are close to their capacity limit of 2 wires. The chosen path costs 5+1+5+1+1+5=18, which is longer but cheaper.

Net C: we next route net C. Again, before we route it, we make sure we have updated all the cell costs. Terminals are again shaded at left. Notice again that a lot of cells have BIG costs, that are intended to strongly deter their use, since they are at or near capacity. Observe again that the net took a somewhat long path to get a cheap cost of 100+5+1+5+1+1+1+1+5=120. The vertical straight connection would cost 215.

Cost before routing c

1	1	1	5	100
5	5	5	100	5
1	1	1	5	5
1	1	1	5	100
1	1	1	1	5

Path for net c

1	1	1	5	100
5	5	5	100	5
1	1	1	5	5
1	1	1	5	100
1	1	1	1	5

Cost after routing C

1	1	5	100	120
5	5	100	100	5
1	1	5	5	5
1	1	5	5	100
1	1	5	5	100

This little example illustrates several critical ideas:

- Each cell has a **capacity** for wires, which simply counts how many net paths can use that cell. Think of this as the available **supply** of wire spaces in this cell.
- Each cell is updated after each route so that it knows how many wires actually use the cell. Think of this as the routing **demand** for wires in this cell.
- Each cell has a **cost** that is a function of its capacity (supply) and current demand. In particular, each cell’s cost gets updated after each route finishes, since the demand has gone up in some cells.
- In general cell costs get bigger as demand gets bigger. Think of this in terms of economics: *scarce things cost more*. The idea is to deter the router from using cells that are close to their capacity, or over their capacity.
- You can decide, if you want, to set the cost of a cell to infinity when it hits capacity; this will preclude your router from ever using the cell. But, you may also fail to route the net at all, this way. Another strategy is to give cells at or over capacity a very BIG cost. This way the router will only use it if it’s truly desperate, and the nets will all route (although some are illegal here).

3.4 Modeling Cell Capacity: the Supply Side of the Router

We adopt a simple first-order model to determine how many wires can use a cell in our placed grid. Given the parameters W, A for a cell in the grid, the capacity is:

capacity = ceiling[W - AG]

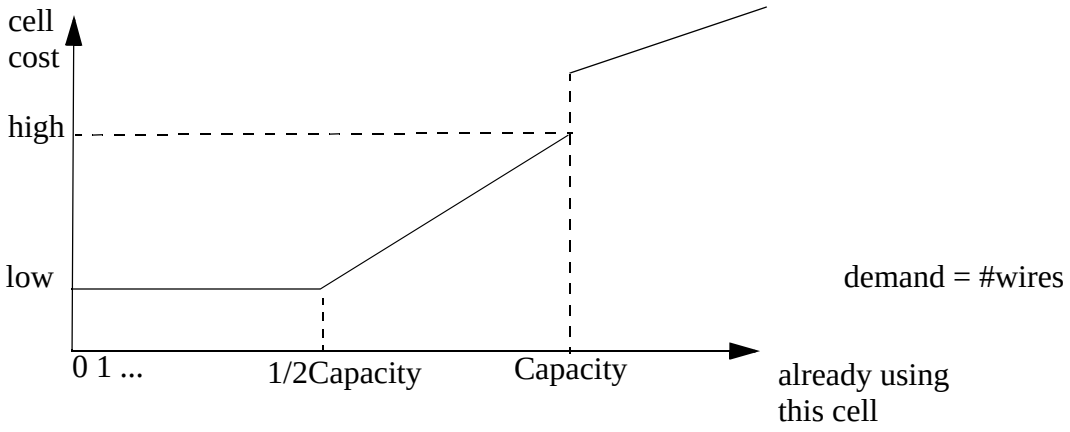
where W and A are parameters associated with each benchmark circuit, and G is simply the number of placed objects in the cell. Usually, G means “gates”, but at the edges of the placement grid, G means “IO pins”. We treat each cell identically with respect to the capacity for wires. Note that if we ignore the ceiling[...] part, the **(W - AG)** formula simply means this:

- If there are 0 placed objects in the cell, you get to have “W” wires use the cell.
- Each gate in the cell consumes “A” potential wire spaces

We use **ceiling**[...] since the “A” parameter may not be an integer. For example, gates may each have a very different number of pins of amount of internal wiring. Perhaps each gate in a netlist consumes, on average, 1.3 wire slots. **ceiling[W - AG]** just round this up to the next biggest integer, and thus ensures we get an integer capacity number, which is important.

3.5 Modeling Cell Cost: the Demand Side of the Router

Since this is, after all, a design class, you get to decide what you want to use here. We will note that cell cost functions tend to look something like this:



When the demand is much *less* than the supply (e.g., less than half the capacity number), the cost of each cell is essentially constant (e.g., “1”). But when the cell starts to gets congested (say, less the half the free wire spaces *remain*), the cost starts to rise. At the max capacity, the cost is very high. And, if you choose to allow routing with cells already over capacity, there is usually a jump in the cost so that cells over capacity are very expensive, and only get used if necessary. Sometimes you also see the “cost is rising” part of the curve as some nonlinear function, like a quadratic, which starts out flattish but rises very fast near the capacity.

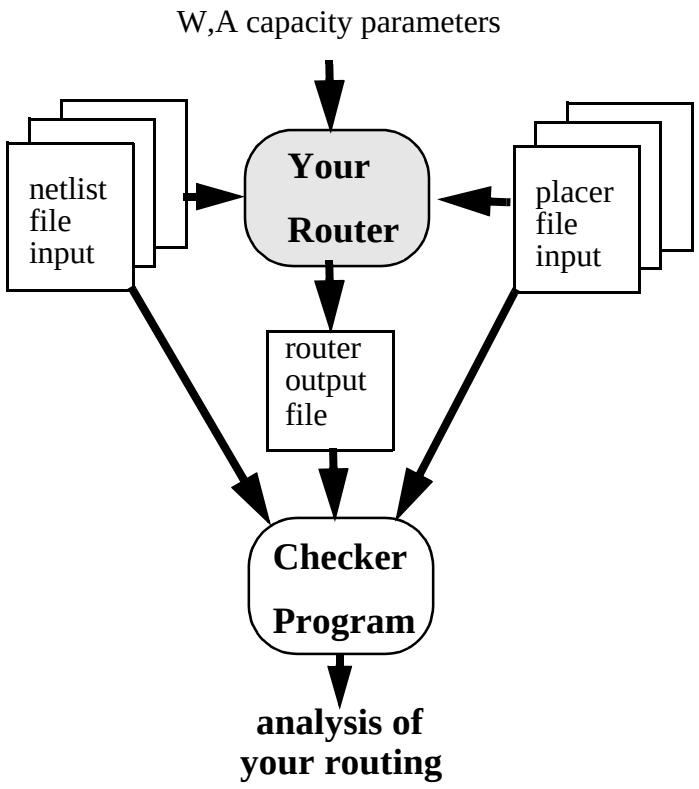
Playing around with this is one of the interesting parts of this assignment.

4.0 Project Function, Objectives and Constraints

Obviously, this layout problem is to be solved with certain functional goals, objectives to optimize, and constraints not to violate (if you can help it). Here we describe these.

4.1 Project Function

Here is the overall structure of the project:



We supply **netlist input** files that tell the geometry of the chip, all the relevant delay and cell site capacity parameters, and the specific paths the placer is supposed to optimize for. The placer reads this, and generates a placement and some output information in a **placement output** file. You read the netlist input file to determine the gates, IO pins and nets; you get to ignore the timing constraints for this project. Your router creates the placement grid, the appropriate cell costs, and routes each net (you hope...). Your router writes the **router output** file, which tells where each wire goes. A checker program we will supply reads these 3 files and lets you know how well you really did: how many wires did you route, what was the total wirelength, how many cells had their routing capacity violated, etc.

4.2 Constraints

Let’s begin with the constraints not to be violated.

1. Every wire in the input netlist should be accounted for. There are 3 ways for you to deal with a net. (1) You routed it. (2) You failed to route it. (3) You decided to ignore it. In other words, you can also decide that won't even *try* to route it, e.g., you can decide up front to punt on any nets with more than 50 terminals to connect. Your output file will specify which of these 3 options (routed, failed to route, ignored). You cannot *lose* any nets. Each net must be accounted for.
2. Each net that your output file says you routed must actually have a connected path that touches all the grid cells that hold the terminals (gates, IO pins) of the net. If you report that a net is partially routed, it must touch at least some of the terminals. It is an error to report a net completely routed if, in fact, your constructed path does not actually touch every net.
3. You should not put more wires in a cell than the cell's capacity, as determined by the W, A numbers for the benchmark. Of course, we actually **expect** that you **will** do this, but the ideal goal is not to.

4.3 Objectives

Objectives are the things you want to optimize. You can decide yourself on what your exact priorities are, but we will be looking at the following:

- Make the total wirelength short.
- Minimize the number of the cells that are over their wiring capacity, and in each such cell, minimize the number of wires over the capacity.
- Try to make your software run in a reasonable amount of time.

Note that our real ideal here is: route 100% of the nets, make them short, don't go over capacity in any cell, do it *fast*.

Note also that this is likely to be *hard*. Welcome to the real world of CAD...

5.0 Implementation Issues

Here we describe what the input and output files look like, what sorts of benchmark problems you have to run, and what sort of software help we will provide.

5.1 Input Netlist File Format

(Note that this file was really designed for the placer we did in Fall97; you can actually ignore a lot of this.) We keep formats simple. The input file is just lines of numbers describing the geometry of all the components, and the netlist. Here is a summary of the format:

```
Xchip Ychip GatesPerGateSite PinsPerPinSite
NumOfGates NumOfNets
1 NumOfNetsOnThisGate NetID NetID ... NetID
2 NumOfNetsOnThisGate NetID NetID ... NetID
...
lastgate NumOfNetsOnThisGate NetID...NetID
NumOfPinsConnectedToNets
1 NetID PinLocation
2 NetID PinLocation
...
NumOfTimingPaths CycleTimeTarget
1 ObjectsOnPath InPinID NetId GateId NetId GateID...NetID OutPinID
2 ObjectsOnPath InPinID NetId GateId NetId GateID...NetID OutPinID
...
```

- The first line tells how wide and tall the chip itself is (in fundamental units), and the capacities of each gate site and pin site. *You need the chip size, but you can ignore the site capacities.*
- The next line tells how many gates (numbered as consecutive integers starting at 1) and nets (ditto) in this problem. *You need both of these numbers.*
- The next **NumOfgates** lines each describe one gate and the nets it connects to. The first number is the gate ID (integer starting at 1), then how many nets this gate connects to, then the net ID (integer) of each of these **NumOfNetsOnThisGate** nets. (Note: there is no separate list of the nets and the gates on each one; you will need to build this as you read in this gate-oriented netlist.) *You need this info, since this really is the netlist.*
- The next line tells how many pins there are that must be placed along the edges of the chip on pin sites. *You need this info.*
- The next **NumOfPinsConnectedToNets** lines each list a pin (consecutive integers numbered from 1) and the NetId number (integer) of the net this pin connects to, and the constraint on which edge you can put the pin on. A **PinLocation** is one of the single characters “t” for top, “b” for bottom, “l” for left, or “r” for right edge. *You need the pin ID info and the net the pin connects to. You don’t need the edge constraint info.*
- The next line tells how many paths we want you to watch while placing, and what your target delay--chip cycle time--is. *You can ignore this.*
- The next **NumOfTimingPaths** lines each specify a path. Each path gets a number, starting from 1. The next field tells how many objects on the path: pins + gates + nets. Then the subsequent **ObjectsOnPath** numbers specify the actual elements of the path. The first element is a pin ID. Then there pairs of NetID number and GateID number, then the final NetID and final PinID. *You can ignore this too--we’re not dealing with timing in this benchmark.*

You should expect to read this information in one line at a time, and build up the data structures you will need to be able to find out quickly things like:

- What nets are on this gate?
- What gates are on this net?

- Where is this gate (what site?)
- Is this net connected to a pin? If so, where is the pin (which site?)
- etc etc etc.

5.2 Input Placement File Format

The placer code we did in Fall97 generated this as an *output* file. This file basically tells where everything got placed, and also gives a lot of info about the critical timing paths. For your router, this is an *input* file. Here is the format, which is again simple:

```
GateID1 ChipXSite ChipYSite
GateID2 ChipXSite ChipYSite
...
GateIDlast ChipXSite ChipYSite
NetID1 Length Delay
NetID2 Length Delay
...
NetIDlast Length Delay
PinID1 ChipXSite ChipYSite
PinID2 ChipXSite ChipYSite
...
PinIDlast ChipXSite ChipYSite
PathID1 Delay
PathID2 Delay
...
PathIDlast Delay
```

- The first **NumOfGates** lines each list a gate ID (consecutive integers from 1), and the relative location (X,Y) on the chip. Note that (ChipXSite, ChipYSite) is measured from the lower-left corner of the chip, numbered as (0,0). *You need this info, since this tells you where the gates got placed.*
- The next **NumOfNets** lines likewise list what the placer program thinks is the Length and Delay of each net. *You can ignore all this.*
- The next **NumOPinsConnectedToNets** lines list for each pin (consecutive integers starting at 1) the location of the pin site (X,Y) used for that pin. *You need this info, since this tells you where the pins got placed.*
- The next **NumOfTimingPaths** lines list for each constrained path (consecutive integers from 1) what you think the delay actually is for that path in your placement. *You can ignore this info.*

As you can see, the output file is pretty simple and basically just dumps the final placement info. You just need to read the (Xgrid, Ygrid) location for each gate and each pin, so you know where each terminal is on each net you want to route.

5.3 Router Output File: Checker Program

How do you know if your code is working? Obviously you will print stuff out, and you can also do some live graphics so you can watch it run.

But to help, we will also provide a CHECKER program that will read all the input files, and an output file format your tool must generate. CHECKER will then report a few useful statistics, like:

- Total wirelength of the nets you completely routed
- Any gross violations, like nets not accounted for, nets you said you routed and did not, etc.
- Any capacity violations: more wires than a cell can hold.
- A routing score that we will use later to try to see how different people's programs attacked the problem. This score will tentatively be a 4-tuple as follows:

  (**WiresUnrouted**, **SumWirelength, SumCellCapacityViolations, SumWireViolations**)

  The first number just counts how many wires you did not completely route. The second number will measure the routed wirelength by just adding up all the cells in all the paths of the nets you correctly routed. The third number counts how many grid cells have wires in excess of capacity. The fourth number counts the actual number of wires over the capacity limit, summing over all the over-full cells. A good rule of thumb here is that this tuple should come out like this:

  (**0, small, 0, 0**)

  Of course, this may not always be possible.

So, you have to write out a router output file that the CHECKER code will read as input, along with the actual netlist input file, to figure this stuff out. Here is the format:

```
NetID1 status NumOfCellsInPath X1 Y1 X2 Y2 ... XN YN
NetID2 status NumOfCellsInPath X1 Y1 X2 Y2 ... XN YN
...
NetIDlast status NumOfCellsInPath X1 Y1 X2 Y2 ... XN YN
```

- The file has **NumOfNets** lines in it., each of which describe how you routed (or did not route) the net. **status** is a single char, one of (c, f, i). "c" means you completed the net. "f" means you failed on this net. "i" means you decided up front to ignore this net. If the status is "c" then **NumOfCellsInPath** tells us how many grid cells are used in the path. The next **NumOfCellsInPath** pairs of integers **Xi Yi** each tell one cell in the path, in no particular order. If the status is "f" or "i" then don't put anything else on the line, i.e., the line should read "**NetIDx f**" or "**NetIDx i**" and nothing else.

As you can see, the router output file is very simple and basically just dumps the final route path info for each net.

5.4 Benchmarks

We will provide a suite of placement benchmarks for you to run your router tool on. These will come in three flavors:

- **Toy Problems:** small enough so that you can see what's happening when you run your tool and manually track down all the bugs in your first cut at the solution. These problems will have 10-100 gates and nets.
- **For-Credit Problems:** we will provide a few "real" problems that you must run your router on, and run the CHECKER program on, and turn in the results. These will be smallish industrial semi-custom netlists with a few thousand gates and nets.
- **We'll-Be-Very-Impressed-Extra-Credit Problems:** Same thing, only bigger. We'll provide problems in the 5000-15000 gate/net range. We'll be delighted to see what happens when you run your tool on these.

6.0 Algorithmic Formulation for Router

The idea here is to use what you know about maze routing to build this global router. You have a lot of flexibility here, but here are some ideas.

6.1 Core Routing Engine

This is really right out of the lecture. The "core" of your router is a maze router that can route multi-point nets in a rectangular grid with non-unit-cost cells. That's basically it.

The trick is that each cell has its cost changed as a function of where the wire went after you finished routing it.

6.2 Multipoint Nets

Yes, you have to be able to support multipoint nets, but as we said earlier, you can decide to punt if the number of terminals is too big (e.g., over 50 or 100, say).

You can do in this 2 different ways:

- **Crude:** this is a hack but its fairly easy. You chop the net up into separate 2 point nets, and then you just route each 2 point net separately. (Look up minimum spanning tree algorithms in your favorite data structures book for how to do this.) You have to make sure that when you're done, all the terminals are connected. This saves you from having to do "real" multipoint nets, where you relabel the current partial route as a source and expand it. You only have to route 2 point nets with this option, but your wirelength will suffer, since you cannot route any Steiner nets (i.e., you must connect terminal to terminal, you cannot connect to the middle of a wire). Note that when you dump the route path, some cells will be repeated, since they were included on different 2 point paths.

This is OK, but the CHECKER program will count the wirelength. You also have to update each cell's capacity and cost correctly. If 3 different 2 point paths use the cell, it gets charged for 3 separate wires against is capacity.

- **Optimal:** route the multipoint nets directly. This means you relabel each partial path as a SOURCE and expand from it to hit the next target. This is really not that hard to do, but you do have to be careful about that SOURCE labeling, and accounting for which cells got used in the final path.

One fussy point to mention is what happens if you can route *some* but *not all* the points of a net? For example, you have a 5 point net, and you can route the first 4 points, but not the last point. What do you do? In this case:

- You have to report this as a failed route. For simplicity, we're not supporting any "partial" net routing in this project.
- You have to undo whatever cost changes you did to the cells as part of this partial routing. For example, after the first 2-point connection got routed, you may have updated the costs of the cells in the grid used in that connection. When you fail on an entire net, you have to restore the costs and capacities back to what they were at the start of the net.

One final fussy point is a reminder: remember to *collapse* the net points down to only the unique grid cells before routing. A net can have many of its gates in the same grid cell, and you only need to route to that cell. In particular, note the following:

- You only need to update the capacities/costs of the cells in the grid. You don't worry about individual gates or IO pins.
- A cell is charged with 1 wire for each path that uses that cell. So, for example, even the trivial wiring of gates a,b,c from our prior example of Net M is charged with using 1 wire in the cell that all the gates reside in. This is clearly a trivial net, but it's still a real net and uses real wire. It just uses wire in exactly one cell.

6.3 Data Structures

You need a grid for the actual placement grid. You store the cell cost info, and the predecessor info, and the "I've been reached during search" flag in here.

You need some kind of cost-sorted data structure for the wavefront. A heap is probably best, but a big cost sorted hash array would probably also work here, depending on how you do the cell costs.

You will also need some basic bookkeeping structures, to keep track of what the nets are, what the gates are, what the pins are, and how they are connected (e.g., gates-on-nets, nets-on-gates, etc.)

6.4 Cell Cost

As we said earlier, it's your call here how to assign costs to the cells as a function of the supply and demand for wires.

6.5 Net Routing Order

You initially at least have to decide what order to try to route the nets in. Conventional wisdom is for you to create an estimate of the length of each net (e.g., 1/2 perimeter of the bounding box of the terminals) and use this to sort the nets. You can route short to long (this is common) or long to short (this also happens). Or you can route in random order (or the order you see the nets numbered in the input file). Some intelligent ordering will probably help your results.

6.6 File IO

Your program should handle its inputs and outputs more or less like this:

YOURROUTER W A netlistinfile placerinfile > routeroutfile

where "**W**" and "**A**" are the capacity parameters, **netlistinfile** is the netlist input, **placerinfile** is the placement input, and **routeroutfile** is the routed output file.

6.7 Ripup and Reroute

It's not as hard as it sounds with a simple global router. **Ripup** means that when you find that you cannot route a net (or a net is "hard" to route) you determine some other nets to remove, and you keep routing. Here is one suggestion for a nice ripup/reroute strategy:

- First, before you do any real routing, route each net individually, with infinite capacity on each cell. The idea is that each net WILL definitely route, and you can tell how long it would like to be, with no other nets around competing for space. Save these length numbers.
- Next, sort the nets on these lengths, shortest to longest, and put these nets in a queue. This is the routing queue, you pop nets off of this in LIFO style to get the next net to route.
- Use a cell cost function that allows nets to route even if they exceed capacity. This means every net will route, although you might violate some capacities to do it, or you might not like the path.
- After each net routes, you look at the "quality" of the path, and if you don't like it, you ripup some nets in the cells on the path that have the most wires in them. Two criteria for "poor quality" are (1) the net's actual routed length is vastly longer than its minimum length, and (2) the net uses cells that are over capacity.

- You get to decide which nets to ripup. You could actually ripup ALL the nets in each cell over capacity and reroute them, or just pick, say, a random subset of the nets in each of these cells. When you rip them up, you physically remove them from the routing grid, so you have go in and update the cell costs correctly. (This is the most mechanically messy part, but its not really that bad). Then, you push them back on the queue (in what order? your call.) and keep routing. The idea is that the router always pops the next net off the queue and routes it, but sometimes this process removes some nets and reinserts them in the queue. You keep routing until the queue is empty.
- The big thing you need to do is to avoid infinite ripup loops, where net A removes net B, and then net removes net A. Easiest thing to do here is to put a counter on each net that limits ripups. For example, no net can get ripped up more than 10 times.

This sounds a little complicated, but the code is not all that hard (though, there is certainly MORE of it). And, this is pretty representative of how real ASIC routers do it, to get really good results.

7.0 What To Do For Credit

Here are the constraints:

- **Groups:** you can work in groups of up to 2 people. Pick your partner carefully. Group dynamics is your business.
- **Code**: you can write in JAVA (which will be slooooow...) or C or C++ on your favorite UNIX box or PC. If you want to use something else, make sure you can read and write our mandatory netlist formats, and get at the data, and get your output to our UNIX CHECKER program. We are not going to provide any porting or translation service here for other than Solaris UNIX.
- **Graphics**: optional but a good idea. It's your business to decide if and what, but your code should could show something *illuminating* as it runs. We suggest at least that you dump a text grid after each net routes, showing where the net went, and also a grid showing how the cost changed. You can be fancy if you like: plot all the sites in artistic detail; show the supply/demand at each site as a color that suggests sparseness or fullness; etc. Debugging a router that uses maze search is vastly easier if you can *see your nets as they each route*, even if you have to just print them out on paper as it runs. Go look in /afs/ece/class/ee760/hw5/annealgraphics for an example of how to do this. This is the same annealer you are asked to augment in HW5, but with a set of graphics drawing code added. Lots of comments in the code show how it works. It's pretty simple if you just want to draw boxes and lines and text.
- **Demo:** you have to sign up for a live demo to the TAs at the end of the assignment. You come to us, you run your program on a new, small benchmark, and we watch your code run and ask you probing questions about what the heck it's doing. If you don't have graphics you at least have to print out some useful ASCII stuff as it runs.
- **Benchmarks:** you run your router on at least the minimum set of benchmarks (we'll specify what this is) and also on any of the "big" netlists if you feel ambitious, and you tell us what the CHECKER program said about your solution, and you also tell us how much CPU time it took and on what machine. In UNIX, if your program is called FOOROUTE, you do this with this command:

  time FOOROUT > FILE

  which will put a line of timing info into FILE when this thing is finished.
- **Write-up:** this is a big deal. You also submit a write-up (and the code) describing how you designed this, and how it works, and analyzing it.

The requirement of the write-up will be scored using the sheet on the following page. The big difference between this and prior writeups is that **this writeup is a web page**. That's right: a web page. You put it someplace we can see it, and grab it, and copy it, and we put it in the projects section of the 760 web page. You can write raw html, or use a web builder, or just use your favorite text processor (e.g., Microsoft WORD, Adobe Frame) and dump the output to a set of web pages. It's easy. You can be as fancy or as simple as you like.

18-760 Spring99 VLSI CAD
Proj. 3: Global Router [100 pts]

NAMES:

Problem Formulation [10 pts]: How did you decide to attack the routing problem? What where your objectives, what trade-offs or assumptions or simplifications did you choose, and why?

Algorithms & Data Structures [15 pts]: In detail, how did you solve this? Algorithm pseudo-code, data structure diagrams, analysis of complexity.

Live Demo [15 pts]: Did it work? Did it show something useful? Did the demonstrators answer questions in a lucid and coherent fashion?

Benchmark Results & Analysis [15 pts]: What did you run, what did CHECKER say about the result, is this good or bad, why are you getting these answers, did you make the right design trade-offs early in the design, etc.? **Analyze** what your tool can do. Include CHECKER outputs, plots of relevant details of the program doing the placement.

Web Write-up Style [20 pts]: Professional, neat, coherent, grammatically clean, etc. Think of this as a web page for the startup company you just launched, that you are referring your customers to, to try to coax another $1,000,000 out of a venture capitalist to fund your breakthrough routing technology.

Code Quality [5 pts]: Yeah, we want to see it. Commented, indented, structured, etc. Just stick it on the web page someplace, and put a link to it.

Ambition/Style [20 pts]: This is a subjective judgement on how *well* you achieved your stated goals. You can shoot for a really solid, simple, elegant solution that does fabulously well for the smaller benchmarks only. You can shoot for something more complex that can run the difficult big benchmarks, but maybe not get great answers. One component of this score is this “style” part--does the tool do something well? The other part is your comparative rank against all the other tools in class, using the output of the CHECKER program. Best-in-benchmark-category gets you a few more points here.