Global Routing Programming

1 Introduction

In this lab, you will learn about how a global router is designed. There will be a fair amount of C/C++ programming and debugging. You will be working with around 2.5K lines of existing source code, while designing some key components for the global router. Your experience in industry will be similar in spirit to this assignment. Make sure to start early and make incremental progress on this project.

The lab is organized into three parts, each with its own hand-in. In Part A you will write the component which reads in global routing nets and become familiar with the basics of global routing. In Part B, you will learn about the framework of SimpleGR, a simple implementation of global router. These two parts will prepare you for Part C, where you will design a global routing engine for SmipleGR that uses A*Search algorithm.

2 Logistics

You will work on this lab alone. Any clarifications and revisions to the assignment will be posted on the course Web page.

3 Handout Instructions

Download handout from course webpage.

1. Start by copying the file `simple-gr-project.tar.gz` to a (protected) directory in which you plan to do your work.
2. Then give the command: `tar xvf gr-project.tar.gz`. This will unpack `simple-gr` folder into the directory. Go into the folder, you will see : `handout.pdf`, `src.tar.gz`.
3. Next, give the command `tar xvf src.tar.gz`. This will create the directory `handout`, which contains the source code of SimpleGR, benchmarks files, and reference solutions for each benchmark. You will be doing all of your work inside this directory.

The folder contains the following items. You will find 4 benchmark files (.gr) under benchmark folder, C++ source code for SimpleGR under src folder, and solution files and congestion maps under goldenRef for your reference:

1. benchmark
2. src
3. goldenRef

4 Part A

Problem Definition

The global routing problem is defined as follows. There is a *grid graph* G that is composed of a set of cells (or vertexes) V and edges E. Each cell v_i in V corresponds to a rectangular box on the grid G, the cell is also called a *gcell*. Each *edge* e_{ij} in E corresponds a link connecting two adjacent gcells v_i and v_j. There is also a set of *nets* N, in which each net n_i contains set of *pins* P_i. Each pin is located in the center of a gcell. Pins of different nets can belong to the same gcell. However, if pins of one net belong to the same gcell, these pins are considered as just one pin. Nets whose pins all belong to the same gcell are ignored in the global routing problem.

The graph G is a three dimensional rectilinear grid. All gcells on G are the same size. A 3D coordinate x, y, z on G corresponds to one gcell. The x and y coordinates are the gcell's column and row indexes on the grid, while z is the gcell's layer. There is a function $f(x, y, z)$ to convert the 3D coordinate of a gcell v_i to its id i, as well as an inverse function $f^{-1}(i)$ that converts an id i back to the 3D coordinate x, y, z.

A net n_i is routed when a rectilinear Steiner tree on graph G is found to connect all the pins of the net. The *wire length* of a net is defined as the number of edges the rectilinear Steiner tree crosses to complete the route. The capacity c_{ij} represents the number of wire tracks available to pass through edge e_{ij}. The demand d_{ij} represents the number of wire tracks used by existing nets that pass through edge e_{ij}. *Overflow* of an edge is then defined as $O_{ij} = max(d_{ij} - c_{ij}, 0)$. A *solution* of the global routing problem is achieved when all the nets n_i in N are routed. A *solution* is considered *routable* if and only if it is *overflow-free* ($O_{ij} = 0$ for all pairs of adjacent v_i and v_j).

Objectives

In general a global router has three objectives. First is to minimize the overflows. Second is to minimize the total wire length of all nets. Third is to minimize of the total runtime. This assignment will evaluate you in all three aspects of the global router that you design.

Example

We now demonstrate a global routing problem with a simple example. The example is taken from the `benchmarks/simple.gr` testcase, which has just one net to route on a 2-layer 3×3 grid. To make things easier, only one overflow-free solution is achievable.

First let us take a quick look at the benchmark file and understand how it defines the global routing problem.

```
grid 3 3 2
vertical capacity 0 1
horizontal capacity 1 0
minimum width 1 1
minimum spacing 0 0
via spacing 0 0
0 0 10 10

num net 1
A 0 2 1
 5  5 1
25  5 1

4
1 0 1   2 0 1   0
1 1 1   2 1 1   0
0 0 2   0 1 2   0
1 1 2   1 2 2   0
```

The first line defines the size of the routing grid. Here we know that the grid has 3 gcells in each row and 3 gcells in each column, and that the grid has 2 layers. To simplify the problem, you may assume that all global routing problems you will face in this assignment have exactly 2 layers.

In global routing, each layer usually has a preferred routing direction, either horizontal or vertical, to force global wires to route along a singular direction on each layer. The `vertical capacity` and `horizontal capacity` fields define this characteristic, stating that all edges on layer 1 has 0 vertical wire track and 1 horizontal wire track, and that edges on layer 2 has 1 vertical wire track and 0 horizontal wire track. As a result, layer 1 allows only horizontal wires and layer 2 allows only vertical wires. You can also assume this layer direction is the same in all global routing problems you will face in this assignment.

The `minimum width` and `minimum spacing` fields describes the number of wire tracks demanded by any net to route over an edge on the specified layer. The routing demand over any edge equals to the sum of minimum width and minimum spacing of that edge's layer. In this case the routing demands on the 2 layers are both 1 (1+0). **Make sure you understand this concept because it will be used in your coding**.

The `via spacing` field can be safely ignored. Vias are metal interconnects completing the linkages among routing layers. They must be used to create turns since each layer only routes in one direction. In this assignment we will ignore all via spacing rules, and assume that there is no via capacity limitation. However, we also assume that **a via costs** $3X$ **the wire length of a normal segment** to penalize excessive zigzagging routes.

Then we are looking at four numbers: `0 0 10 10`, presenting: the global grid's starting X coordinate, Y coordinate, each gcell's width, and height. These coordinates and sizes are based on the "detailed" coordinates of this design, NOT the grid coordinates used by global route. "Detailed" coordinates are much more fine-grain than the global routing grid coordinates. They define the detailed locations of components

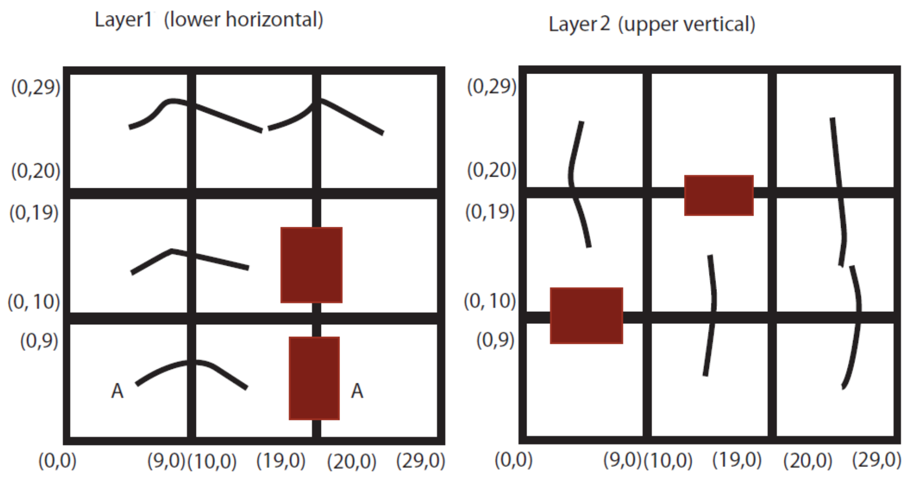

Figure 1: An illustrative example of benchmarks/simple.gr testcase.

on a design. However, routing with detailed coordinates can be computationally prohibitive for real world designs. In this testcase, recall that the global routing grid is 3×3. Given these four numbers we can derive that the detailed coordinate is 30×30 – a problem size increased by 100 fold.

The following part defines the net list of this design. `num net 1` tells us there is one net in the design. `A 0 2 1` gives us the net's name: A, database ID: 0, number of pins (always 2 in this assignment), and wire width (Ignored. We assume it is always overruled by the minimum width and spacing rules mentioned earlier). The next two lines are the detailed coordinates of the two pins. We can see that they are both on layer 1. Note that here the layer number starts from 1 (not 0).

The last part defines adjustments of edge capacities. Recall that `vertical capacity` and `horizontal capacity` fields already defined the capacity of all edges globally. What is different here is that a selection of edges are picked out and assigned with new capacities. The first line "4" simply suggests 4 edges are up for adjustments. `1 0 1 2 0 1 0` means edge connecting gcells $(1, 0, 1)$ and $(2, 0, 1)$ (both on layer 1) has zero or no capacity. This adjustment is typically used to describe a blockage that removes a part or all of an edge's capacity.

At last, we will use a figure to show this global routing problem. Figure 1 is a good visualization of the 3×3 global grid on 2 layers. The X and Y coordinates are detailed coordinates, which mark the placement of gcells on the design. The two pins of net A are marked on their gcells on the first layer (horizontal). Four blockages shown as red boxes indicate no track capacity on the edges covered by them. The design is routable. The curved lines crossing gcell boundaries marks the segments used for routing net A. The wire changes layers several times through vias (not shown), wandering through the maze. Eventually, the solution consists of 4 horizontal segments, 4 vertical segments, and 6 vias, for a total "wire length" of $4 + 4 + 6 \times 3 = 26$, where each via costing 3 unit length.

Deliverables

Use your knowledge of global routing to fill in some comments to help future coders understand how SimpleGR loads global routing data.

Locate the source file `IO.cpp` and find function `void SimpleGR::parseInput()`. Within this function, there are many comments that start with "`FIXME`", followed with a requirement. Your task is to replaced these comments with a proper description that fits the requirement. As you doing so, please pay attention to the following rules:

- Only work within the scope of the specified function.
- Do not make changes to the code.
- Make sure your comments are short and crisp, easy to read.
- Your `IO.cpp` file will be submitted as Part A deliverable.

Extra: A function `buildGrid()` is called by `parseInput()`. Find this function and read through it as an extra exercise. This exercise is not graded but will provide you with information of how routing grid data structures (`GCell` and `Edge` classes specifically) are built, which will become useful knowledge for later assignments.

Part B

This section will briefly introduce the general framework of SimpleGR, and get you ready for Part C. First, let us take a look at Figure 2, which shows the `main()` function of SimpleGR. From this function we can see the top level design flow of SimpleGR. After instantiation of a SimpleGR parameter and object, the first thing to do is loading a design into SimpleGR's database. This is done with the `parseInput()` function, which was commented by you in Part A. Then SimpleGR performs a 3-stage routing routine: initial route, rip-up and re-route, and greedy route. Once done, SimpleGR outputs the solution to a text file.

Now build your SimpleGR for the first time and give it a spin.

```
unix>  cd src
unix>  make all
unix>  ./SimpleGR -f simple.gr
```

You will notice the output of the router is divided into 3 stages. For this portion of the assignment, create a file titled README.txt. At the top, put your name and A-number. Answer the following questions in the README.txt file based on what you have learned in class, as well as what you have learned from inspecting the provided code.

- What is the purpose of each of the 3 stages in this routing algorithm? How are they different from each other? How do they work together as a flow?
- How does SimpleGR build a framework to allow the 3 stages? (Think high-level, it should be brief)

```
int main(int argc, char **argv)
{
  printTitle();
  // init parameters of SimpleGR
  SimpleGRParams params(argc, argv);
  // instantiate an instance of SimpleGR
  SimpleGR simplegr(params);

  // read in the nets and create the grid input file
  simplegr.parseInput();
  simplegr.printParams();

  // perform 3-stage global routing
  simplegr.initialRouting();
  simplegr.printStatistics();

  simplegr.doRRR();
  simplegr.printStatistics();

  simplegr.greedyImprovement();
  simplegr.printStatistics(true, true);

  // output solution file
  simplegr.writeRoutes();

  return 0;
}
```

Figure 2: **The main() function of SimpleGR.** See `src/main.cpp`

Part C

With the preparation of Part A and B, now you are ready to take on the task of building a maze router. In this section you will be working on `MazeRouter.cpp` file alone, to implement an A* search based maze router engine.

Coding Rules

You are free to make any modifications you wish, with the following constraints:

- Stay within `MazeRoute.cpp`. The only exception is when you need to make your own function, in which case you can edit the `SimpleGR` class in `SimpleGR.h`, and add your function *declaration* in the `private` section of this class.

Building and Running Your Solution

Doing the following will run SimpleGR for `adaptec1` benchmark, and generate a congestion map based, which is saved in `Congestion.xpm` file.

```
unix>  cd src
unix>  make
unix>  ./SimpleGR.exe -f adaptec1.simple.gr -o adaptec1.sol
unix>  ./mapper.exe adaptec1.simple.gr adaptec1.sol
```

5 Evaluation

The lab is worth 100 points:

- *Part A: 20 points:* Correctly comment the IO.cpp file
- *Part B: 20 points:* Correctly answering the questions in your README.txt file
- *Part C: 60 points:*

  Code execution does not result in segmentation fault, and successfully routes 100% of the nets (30 points);

  Correctly handle overflow, capacity, and boundary limitations (10 points)

  Implements A* algorithm (10 points)

  Produces results with total length for routed nets and wire length that exceed golden model results by no more than 1 % (10 points)

6 Handin Instructions

Make a directory with your name and A-number. Copy into that directory your IO.cpp file, README.txt, and your MazeRoute.cpp (and corresponding files if you edited the SimpleGR class). Also include a text file which shows the information printed to the console during program execution for all 3 of the example netlists (adaptec1, 2, and 3). Compress the directory using tar. See example below:

```
unix>  tar cvf JohnDoe_A00000000.tar JohnDoe_A00000000
```

This compresses the directory to a single file. Submit the .tar file on canvas.