A Simple Global Router Project

1. Introduction

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

You will do the project on a Linux machine. We recommend to use g++ later than version 4.4 for this assignment. Now extract the .tar.gz file with the following command in terminal:

```
tar xvf gr_project.tar.gz
```

This will unpack the package and create folder **gr_project**. In this folder, you have the source code of a very simplified global router called SimpleGR. SimpleGR is derived from an advanced academic global router. SimpleGR largely resembles the design flow and global routing framework of an actual global router.

In addition to the source code of SimpleGR, you should also find a "benchmarks" folder which contains 3 workloads for the Global Router (**adaptec 1,2,3**). These workload are derived from actual industrial standard global routing benchmarks, and simplified to be parsable by SimpleGR. We also include a **simple.gr** workload with its solution file **simple.gr.sol**; they are used for demonstration and debugging purposes.

2. Objectives

In this project, your main objectives are listed as follows:

- Understand the design flow of SimpleGR.
- Implement the A* search for the SimpleGR routing engine.
- Experiment your design using the provided benchmarks.
- Verify the correctness of your design solutions against the golden results.

3. Instructions

The following important steps will help you navigate through the source code of the SimpleGR project, and get familiar with the provided utilities to assist your development.

1. Starting with the **main()** function.

The **`main()`** function is located in the route.cpp file. It is the only function of the file, describing the general flow of the global router from the top of the hierarchy. Some descriptions of this function are listed as the follows:

- Line 23-37: **`main()`** function parses the input arguments, and use these arguments to initialize a SimpleGR object.
- Line 40 – 74: **`main()`** uses several stages to route the design.
- Line 76-78: output the solution to a file.

2. Major functions.

The class SimpleGR is the main object that houses all global routing routines. From the top view, it uses the following functions to complete global routing:

```
void initialRouting(void);
void doRRR(void);
void greedyImprovement(void);
```

Under the hood, these functions use wrappers around the routing kernel to perform different maze routing tasks. One of most important wrapper function is:

```
CostType SimpleGR::routeNet(const IdType netId, bool allowOverflow, bool
   bboxConstrain, const CostFunction &f)
```

This function is called during all stages of global routing. It wraps around the `routeMaze(...)` `function`, **which is where most of your coding will happen.**

```
CostType routeMaze(const IdType netId, bool allowOverflow, const Point
   &botleft, const Point &topright, const CostFunction &f, vector<IdType>
   &path);
```

You can find this function at line 179 in SimpleGR.cpp file. Its input parameters are:

- **`netId`**: an IdType (defined in SimpleGR.h) variable that identifies the net that needs to be routed.
- **`allowOverflow`**: specify whether the newly routed path allows to cause overflow.
- **`botleft`**, **`topright`**: two point types (defined in SimpleGR.h) that specifies a bounding box of the routing region. Tiles on the edge of the bounding box are allowed for routing.
- **`f`**: the cost function to calculate an edge cost. Several cost function classes are defined between line 264 – 313 in SimpleGR.h.
- **`path`**: saves the edges of the routed paths. If no route is found then vector path should be empty.

Return value: A CostType (defined in SimpleGR.h) that equals to the accumulated edge cost of the found path.

Inside **`routeMaze()`** you are required to code the A* search kernel to enable maze routing, which loops until all "frontiers" in the priority queue are exhausted, or when the sink tile is found. The heuristic cost function for the A* search is instantiated for you as **`lb`** at line 195. You are also required to code the back-tracing procedure after search is concluded, and fill up **`path`** with all the edge IDs along the route.

You are encouraged to browse through the rest of the source code in the project to better understand how things work. Specifically, you need to understand the **priorityQueue** class claimed in SimpleGR.h and defined in Utils.cpp, between line 291 – 411.

3. Compiling the code

Do a "make" under the project root folder compiles the project. "make clean" cleans up the directory, leaving only the source code.

4. Run the SimpleGR.exe with benchmarks

Use the following command to run the global router, ignore the rest of the parameters of SimpleGR.

./SimpleGR.exe <benchmark file name> -o <output file name>

You can assign the output file name to anything you like, the file is simply a text file.

5. Check your results

Your output file should be compared with the output of a golden model. Use the Unix "diff" command to verify if your output file and the golden solution are at an exact match.
You can use the "mapper.exe" utility to generate a congested map of the solution. To do so, run mapper.exe with:

./mapper.exe <benchmark file name> <output file name>

If returned successfully, a "congestion.xpm" file will be generated under the folder. You can use an image viewer to open this file and visualize the congestion map of the solution.

4. Deliverable

You will hand in a report and your modified source code for this assignment. Your code should be able to compile, and the resultant global router binary should generate the same global routing solutions as the golden solutions.

Your report should include the following content:

1. A simple figure illustrating the general design flow of global routing, and a short paragraph explaining this figure.
2. A table reporting the total runtime and total wirelength of your global routing solutions with all 3 benchmarks
3. Figures of the congestion map of these benchmarks.

5. Hints

1. Go over the entire project before started coding. You will see some utilities have been provided for your ease. For example, the priority queue needed for A* search is already implemented.
2. Read about A* search http://en.wikipedia.org/wiki/A*_search_algorithm
3. The capacity of an edge is the width of that edge. The usage of an edge is the sum of the required widths of all the nets assigned to that edge. This includes the spacing between the nets. When overflow is not allowed, the total usage on an edge should not exceed the capacity of that edge. Calculate the usage of each edge by a single net with this equation:
   usage = max(min width of net, min width of layer) + min spacing of layer.
4. There are two layers on the routing grid, one goes only horizontally, the other goes only vertically.
5. Each net has and only has two pins.
6. For debugging, use the benchmarks/simple.gr input file. This input has a small 3x3 routing map with only one net. Your solution should match the benchmarks/simple.gr.sol file.

7. Refer to other parts of the code to determine how to reference edge IDs based on the associated tile IDs.
8. Edges Edges between layers do not have a max capacity, they are known as 'vias'.
9. A bounding box is two-dimensional rectangle, which restricts the path of a net to stay within its region. A net must NOT create any connectivity outside of the rectangle box.

6. Example problem

Here we will use the benchmarks/simple.gr input file as an example to explain how we describe a 3D routing problem with congestion to avoid on the routing map.

The figure below lists the routing problem described in benchmarks/simple.gr. This is a 3D routing graph with two routing layers. Each graph edge on a layer has a capacity of 1, with the first layer having **only** horizontal capacity, and the second having **only** vertical capacity.

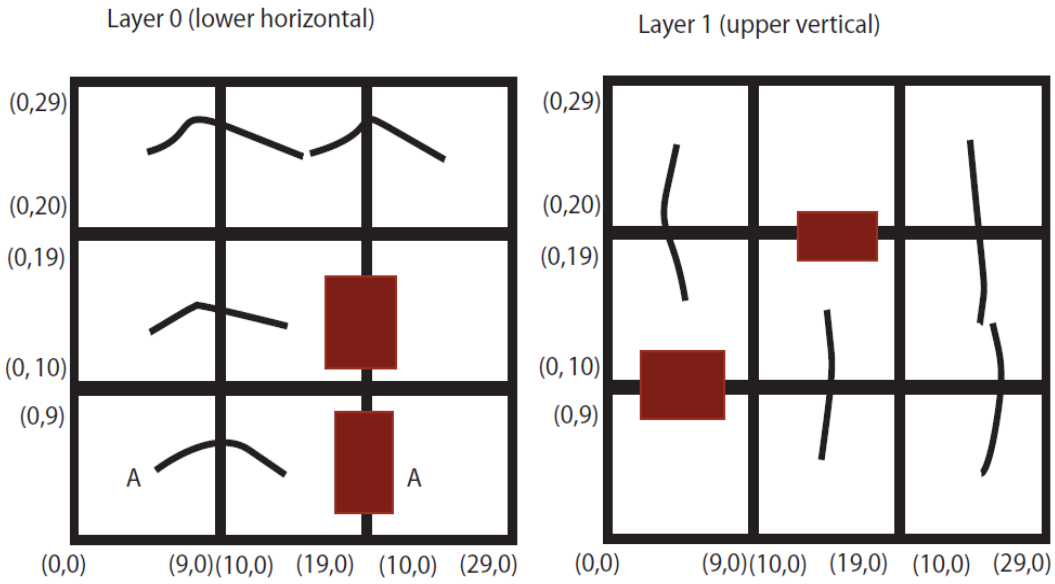
simple.gr

```
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
```

There is congestion in this graph; it is noted in the last 4 lines of benchmarks/simple.gr. These lines indicate that the capacity between a pair of tiles is reduced. The tiles will always be adjacent, and the number at the end of the line indicates the total capacity. The congestion is indicated in the figure below as red blocks.



Net A has 3 pins, 2 of which fall within the same global routing tile. That is why the net is a 2-pin net on the global routing scale. To route the net, it is necessary to via up and down several times, wandering through a maze. The solution consists of 4 horizontal segments, 4 vertical segments, and 6 vias, for a total "length" of 4+4+6*3 = 26, with each via costing 3 unit length.