THE UNIVERSITY of WISCONSIN MADISON

ECE556 Project: Part 2

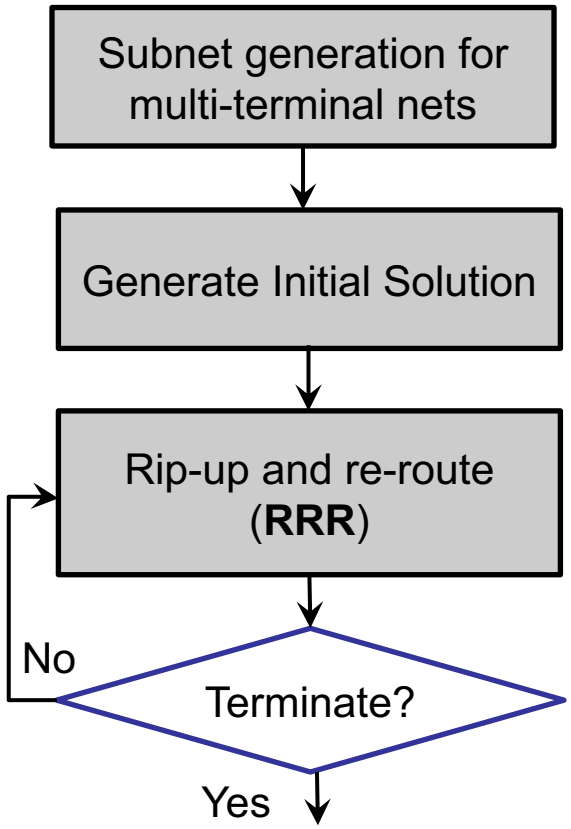
Azadeh Davoodi

Outline

- Routing framework
- Suggestion for teamwork
- Minimum project requirements
- Format of executable and required input arguments
- What to submit?
- Project grading

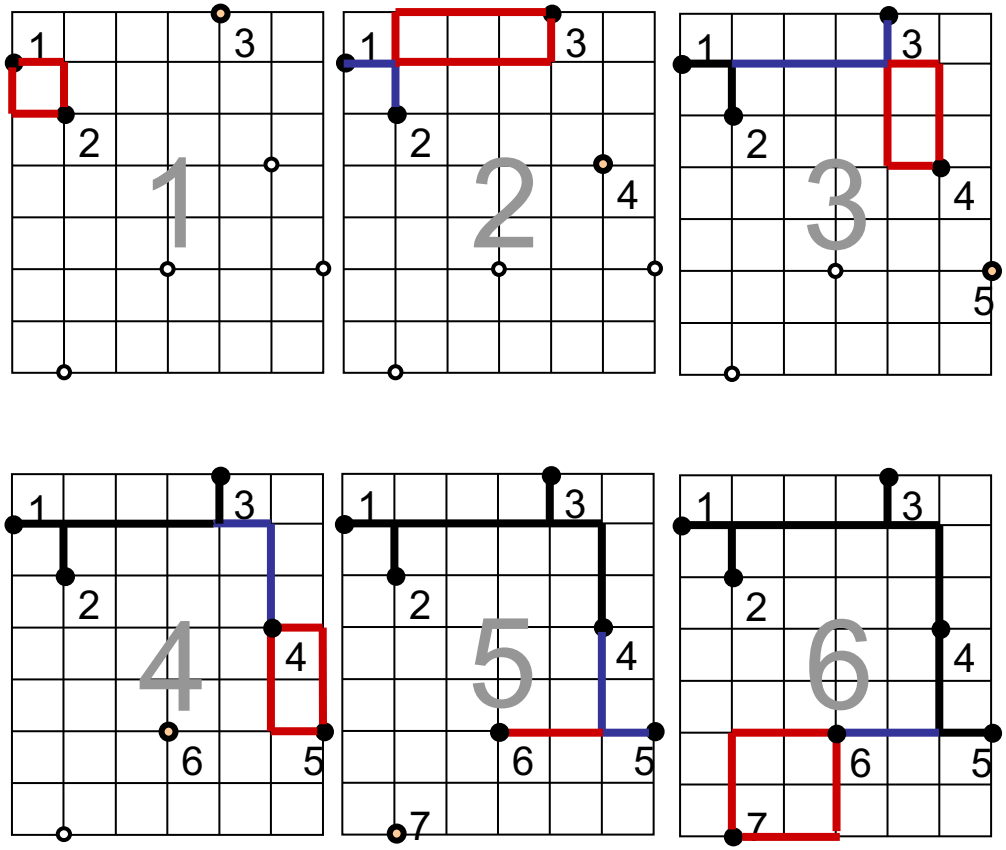
Routing Framework

- Components for project
  1. Subnet generation
  2. Generate initial solution
  3. Rip-up and reroute
     1. Edge weight calculation
     2. Net ordering
     3. Maze routing
  4. Termination condition



(1) Subnet Generation

- Generate two-terminal subnets for multi-terminal nets
- This step is essentially reordering of the entries in the pins array of the net data structure, assuming each pair of consecutive pins in the array make a subnet
- You can adapt a simpler variation of the heuristic for rectilinear routing for subnet generation (given in GlobalRouting slides)



(2) Generate Initial Solution

- This part was already done in the first part of the project
  - Note that your subnet generation in step (1) is now going to be applied before generating an initial solution

(3) Steps in RRR

3.0* At the first RRR iteration, compute all edge weights

- Assuming edge weights are updated in coarse-grained manner (*see slide 8 for option of fine-grained edge update)

3.1 Calculate new net ordering

- Note: ordering is defined for nets (and not subnets)

3.2 For each net in the order found from step 3.1

- Rip-uping up a net:
  - Rip-up means to remove the existing route stored for the net by decreasing the edge utilizations corresponding to the route
- Rerouting a net
  - Reroute means to generate a better route (of lower cost) for the net
    - Based on sequential routing of its 2-pin subnets (generating a “segment” corresponding to each subnet in the net data structure)
    - After routing each subnet, update the edge utilizations of the edges corresponding to the new route

(3.0) Computing Edge Weights

- The weight of an edge is based on negotiation-based routing and given by this formula:
- $w^k_e = o^{k-1}_e \times h^k_e$
  - where o^{k-1}_e is overflow given by : $o^{k-1}_e = max\ (\ u^{k-1}_e - c_e\ ,\ 0)$

  u^{k-1}_e : utilization of edge e is the number of routes passing from e based on the currently-stored routes (of all routed nets) and c_e is the capacity of edge e
  - and h^k_e : overflow history of edge e (set to 1 at the beginning of the first iteration k=1)

$$h_e^k = \begin{cases} h_e^{k-1} + 1 & \text{; if } o^{k-1}_e > 0 \text{ (edge e has overflow)} \\ h_e^{k-1} & \text{; else} \end{cases}$$

- History can be updated everytime the edge utilization is updated in your program

(3.0) Computing Edge Weights

- When to update the edge weights? Can pick one of these two options:
  1. Coarse grained: Update at the start of each RRR iteration, based on the currently-stored routes
     - Edge weights should be recalculated for each edge based on the stored edge utilizations
  2. Fine grained: Update every time the edge utilizations change
     - After each rip-up and after each reroute, update the edge weights of affected edges (those with change in their utilization)

(3.1) Calculate Net Ordering

- Given the updated edge weights at the beginning of each RRR iteration, for each net *n*, we calculate a cost as follows
  - *cost (n)* = $\sum_{\forall e \in route(n)} w^k_e$
  - where route(n) is the route stored for net n
  - which is either the initial solution (k=1) or the route from the last iteration of RRR (iteration k-1)
- Sort the nets in decreasing order of their costs to create the ordering
  - We will first route the nets with higher cost
- The above ordering is only for nets (and not subnets)

(3.2) Single-Net Routing

- Single-net routing is used during the RRR stage for each 2-pin subnet
- Implement any variation of single-net routing that you desire
  - Min-cost routing : most-time consuming
  - Min-cost routing with ”framing” option explained for Lee’s algorithm : helps accelerate the speed
  - A* (most popular one used in modern routers)
  - Pattern routing : easiest option
    - Explore among patterns such as L-shape, Z-shape, U-shape, etc.
    - Can integrate with framing idea to gradually increase the frame size as a function of iteration to allow searching in wider area with increase in iteration count

(4) Termination Condition

- Min time: 5min (must apply RRR)
- Max time: 30min
- Can be anything in between based on optimizing the quality metric below:

  Q = TOF(1+RUN/15)

  - where TOF is total overflow (sum of edge overflow) of your final, generated, routing solution as reported by the evaluation script
  - RUN is runtime of your executable in minutes on a free tux machine (best-tux.cae.wisc.edu)
  - Notice that 15 minutes is used as a reference for your runtime so any runtime beyond 15 minutes will result in scaling up TOF accordingly

Suggestion for Teamwork

1. Subnet generation (M1)
2. Generate initial solution (part 1)
3. Rip-up and reroute
    1. Edge weight calculation (M1)
    2. Net ordering (M2)
    3. Maze routing (M2)
4. Termination condition (M1)

Format of the Executable

- **Example: ./ROUTE.exe –d=0 –n=0 <input_benchmark_name> <output_file_name>**
  - Please make sure to follow the exact above format

Input arguments:

-d=0 -n=0 : no subnet generation, no net ordering, no ripup and reroute (this is just your part 1)

-d=0 -n=1 : default subnet generation from part 1, but apply net ordering, and apply ripup and reroute

-d=1 -n=0 : apply subnet generation, but no net ordering, and no ripup and reroute (same as part 1 but with your net decomposition enabled)

-d=1 -n=1 : apply all the features and ripup and reroute

Minimum Requirements of the Project

- Your router should have all the components defined in slide 3
- It should additionally have the expected behavior as defined below based on TOF (total overflow) as measured/reported by the evaluation script:
  - TOF in –d=1 –n=1 case is better than –n=0 –d=0
  - TOF in –d=0 –n=1 case is better than –n=0 –d=0 but worse than –n=1 –d=1
  - TOF in –d=1 –n=0 case is better than –d=0 –n=0 but worse than –n=1 –d=1

Project Grading

- Part 1: 20%
  - Full grade if passes evaluation script for the adaptec1 benchmark
- Part 2:
  - 60% : Based on meeting minimum requirements (defined in slide 13)
  - 20% : Based on value of “quality metric” of your submission (defined in slide 11)
  - The quality metric Q will only be measured if the minimum requirements are satisfied. If they are not satisfied, looking at the quality metric will be irrelevant, and no grade will be given for this portion.

What to Submit?

Group submission: deadline Sunday 4/30

1. A single file labeled "ece556-p2.tar" in the format .tar which includes the following files:
   - (1) ece556.cpp (2) ece556.h (3) main.cpp (4) Makefile (5) a README file
   - The tar file may include additional source files but may not include object file (.o) or executables
   - Include project team member names in the ReadMe file with a brief description of the responsibilities of each member
   - Submit only one tar file per group
   - What to include in the readme file?
     1. Fill the table below for adaptec1, adaptec2, adaptec3 using TOF and TWL reported by the evaluation script
     2. Responsibilities of ea

-d=0 -n=0		-d=0 -n=1		-d=1 -n=0		-d=1 -n=1	
TWL	TOF	TWL	TOF	TWL	TOF	TWL	TOF

What to Submit?

Individual report submission: deadline Friday 5/5

2. Additionally, each group member should submit a one-page PDF report

- The report should include:
  - What were your responsibilities? Please be as specific as possible. General statements such as we all worked together on X will result in asking for individual demos to explain what was done in-person by showing your code.
  - If you worked on improving Q, please explain what idea(s) did you explore beyond what was discussed at class.

- We are aware of some source codes for ECE556 posted online and will be carefully checking for authentic code. Any evidence of misconduct will be reported to the Office of Student Conduct & Community Standards, as given in the syllabus.