Algorithm PA3 report

Ivan Chang

December 25, 2025

1 Data Structure

To efficiently handle the 3D Global Routing problem, we implemented several key data structures designed to optimize memory usage and access speed.

1.1 Graph Construction

The routing grid is abstracted into a graph $G = (V, E)$ stored as an adjacency list (`std::vector<std::vector<Edge>>`).

- **Vertices (V)**: Each GCell corresponds to a vertex.
- **Edges (E)**: Edges connect adjacent GCells in the x, y (planar) and z (via) directions.
- **Edge Weights**: Each edge stores a `baseCost`, which represents the physical wirelength (W_j, H_i) or the via cost (WL_{via}).

1.2 Net and Coordinates

Nets are stored as a struct containing the net name and two `Coord3D` pins. The `Coord3D` structure simply holds $\{layer, col, row\}$.

2 Cost Function

The core of the global router relies on a dynamic cost function that penalizes congestion. The cost of traversing an edge $e(u, v)$ is defined as the physical length plus the congestion penalty of the target vertex v.

2.1 Vertex Cost Formulation

In `router.cpp`, the function `computeVertexCost` calculates the penalty for occupying a GCell based on current overflow. The cost is exponential to heavily discourage using overflowing cells.

Let D_v be the demand and C_v be the capacity of vertex v. The overflow is $O_v = \max(0, D_v - C_v)$. The base vertex cost is:

$$Cost_{base}(v) = \alpha \times (2^{O_v} - 1) + \mathbb{I}(O_v > 0) \times P_{overflow} \tag{1}$$

Where:

- $\alpha = 1000$ (Scaling factor).
- $P_{overflow} = 100,000$ (A large fixed penalty added if any overflow exists).
- The overflow O_v is clamped at 20 to prevent integer overflow.

2.2 History Cost (Negotiation)

To resolve contention between nets that repeatedly compete for the same resources, we employ a history-based cost.

$$Cost_{total}(v) = Cost_{base}(v) + \beta \times h_v \tag{2}$$

Where:

- h_v is the history count, incremented by 1 in every iteration where GCell v is overfull.
- $\beta = 1000$ is the history penalty factor.

This ensures that if a cell remains congested over many iterations, its cost rises permanently, forcing nets to explore alternative, longer paths.

3 Routing Strategy

We implemented a Negotiation-based Rip-up and Reroute (RRR) strategy utilizing the A* search algorithm.

3.1 A* Search Algorithm

Pathfinding is performed using A* search.

- **Heuristic ($h(n)$)**: We use the 3D Manhattan distance between the current GCell and the target pin. This is admissible and consistent, ensuring optimal pathfinding relative to the current edge weights.
- **Cost ($g(n)$)**: The accumulated cost from the source.
- **Weight Calculation**: In the relaxation step, the weight to move from u to v is $EdgeLength(u, v) + Cost_{total}(v)$.

This allows the router to balance wirelength minimization (via edge lengths) and congestion avoidance (via vertex costs).

3.2 Rip-up and Reroute (RRR) Workflow

The routing process proceeds iteratively:

1. **Initial Routing**: Route all nets sequentially using A* on an empty grid (considering only physical capacity constraints).
2. **Iterative Refinement**:

- Calculate total overflow. If 0, the solution is valid and we terminate.
- Update history costs for all currently overfull GCells.
- **Net Selection**: Identify nets passing through overflowed GCells.
- **Ordering**: Shuffle the selected nets randomly, then sort them based on the number of overflow hits (nets causing the most congestion are routed first).
- **Reroute**: For each selected net:
  (a) **Rip-up**: Remove the net's demand from the grid.
  (b) **Update Graph**: Recalculate vertex costs with the updated demand and history.
  (c) **Route**: Run A* to find a new path.
  (d) **Commit**: Add the new path's demand to the grid.

3.3 Stagnation Control

To prevent the algorithm from getting stuck in local minima, we track a stagnation counter. If the total overflow does not decrease for 200 consecutive iterations (`stagcnt >= 200`), we force a full reroute of *all* nets, not just the congested ones, to perturb the system significantly.

4 Results and Analysis

4.1 Experimental Results

The router was tested on the provided benchmarks. Below is a summary of the performance metrics.

Case	Status	Overflow	Cost	Time
Case 1	PASS	0	70800	0m0.004s
Case 2	PASS	0	11785600	0m0.011s
Case 3	PASS	0	68233400	0m0.108s
Case 4	PASS	0	12314800	0m5.941s
Case 5	PASS	13	144668840	4m45.348s
Case 6	PASS	13	243300290	4m46.724s

Figure 1: Routing Results Summary

4.2 Analysis

- **Effectiveness of History Cost**: The inclusion of the history variable h_v was critical. In early iterations, nets oscillated between two equally congested paths. The history cost broke this symmetry by permanently penalizing the frequently contended nodes.
- **Exponential Penalty**: Using $2^{overflow}$ provided a strong gradient. It allowed the router to distinguish between "slightly congested" (overflow 1) and "heavily congested" (overflow 5), guiding paths toward the former if necessary.

- **Runtime vs. Quality**: The A* heuristic significantly sped up the search compared to Dijkstra for it only explores promising paths. However, the overall runtime is still dominated by the number of iterations needed to resolve congestion.