Global Routing Project Report

Optimizing VLSI Interconnect Congestion Using BFS-based Algorithms

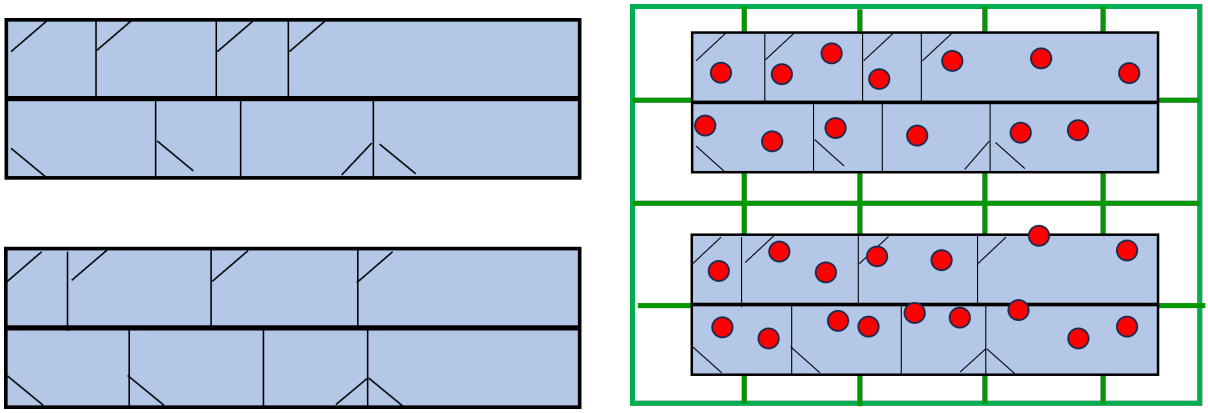
X1136010 黃偉祥

110034042 楊采翊

May 28, 2025

1 Introduction

The Global Routing Project focuses on developing an efficient global routing algorithm for VLSI design, as specified in the project guidelines. The objective is to route a set of nets across a grid of global routing cells (Gcells) while minimizing congestion at Gcell boundaries. Each Gcell has a 3×4 port configuration (excluding boundary cells), and the routing utilizes three metal layers: M1 and M3 for horizontal routing, and M2 for vertical routing. This report presents the proposed methodology, implementation details, visualization of the congestion map, and routing results.



(a) Given a row placement result (b) Superimpose global routing grids

Figure 1: Introduction

2 Problem Statement

2.1 Project requirements

The project requires:

- Reading input data specifying routing layers, Gcell grid dimensions, boundary capacities, and a netlist of pins to be connected.
- Performing global routing for each net, considering the constraints of three metal layers (M1, M3 horizontal; M2 vertical).

- Minimizing congestion by balancing the demand and supply at Gcell boundaries.
- Generating a congestion map to visualize supply, demand, and overflow at each boundary.
- Highlighting the most severe overflow boundaries to guide optimization.

2.2 Input format analysis

The input format includes routing layer definitions, Gcell grid size (e.g., 5x4), boundary capacities (e.g., L(M1, M3), R(M1, M3), T(M2), B(M2)), and netlists specifying pin coordinates (e.g., N9: (4,3), (2,2), (2,1), (0,2)). The output should include routed paths for each net and a congestion map showing supply, demand, and overflow.

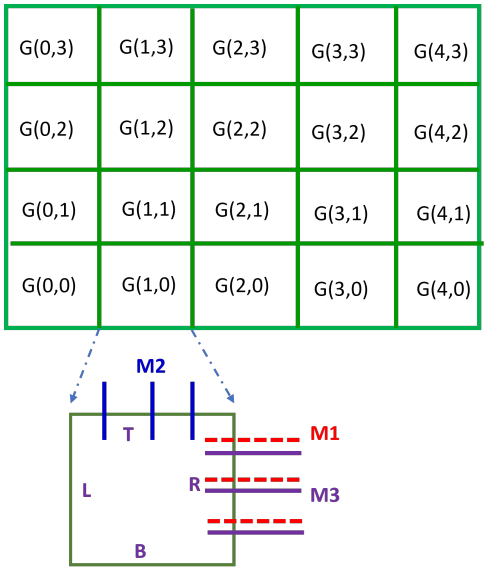


Figure 2: Gcell distribution and routing layer definitions.

2.2.1 Routing Layers Definition

```
routing_layers 3  M1 horizontal  M2 vertical  M3 horizontal
```

- `routing_layers` - Keyword
- `3` - Total number of routing layers
- `M1 horizontal` - Layer name and preferred direction (horizontal/vertical)
- Subsequent layers follow the same pattern

2.2.2 Gcell Grid Specification

```
Gcell_grid  5 4
```

- Defines a 5-column $\times$ 4-row grid of global routing cells
- Column indices range: 0 to 4 (left to right)
- Row indices range: 0 to 3 (bottom to top)

2.2.3 Gcell Boundary Capacities

```
GC 0 0  L (M1 0  M3 0)  R (M1 1  M3 3)  B (M2 0)  T (M2 3)
```

- `GC column row` - Gcell coordinates
- Edge capacities:
  - `L` - Left edge (M1 and M3 capacities)
  - `R` - Right edge (M1 and M3 capacities)
  - `B` - Bottom edge (M2 capacity)
  - `T` - Top edge (M2 capacity)
- Example: Right edge allows 1 M1 and 3 M3 horizontal routes

2.2.4 Netlist Definition

```
N1  (0 0)  (1 0) (2 2)
```

- `N1` - Net name
- `(x y)` - Pin coordinates in Gcell grid units
- Multi-pin nets are specified with consecutive coordinates

Table 1: Input File Structure Summary

Section	**Description**
Routing Layers	Defines metal layer properties
Gcell Grid	Specifies routing array dimensions
GC Entries	Contains edge capacities for each Gcell
Nets	Lists all nets and their pin locations

3 Methodology

The proposed approach consists of the following steps:

3.1 Coordinate System Transformation

Each Gcell is assigned a new coordinate system with a 4x5 (or multiples of 4x5 e.g. 8x10) port configuration. This transformation maps each Gcell (x, y) to a unique set of coordinates in a larger grid, allows us to visualize the routing path without overlapping the paths and making them indistinguishable.

For example, the coordinates (0,0) to (4,5) belong to G(0,0). Coordinates (12,10) to (16,15) belong to G(3,2) Take the Fig. 3 as an example, point (13,7) belongs to G(3,1)

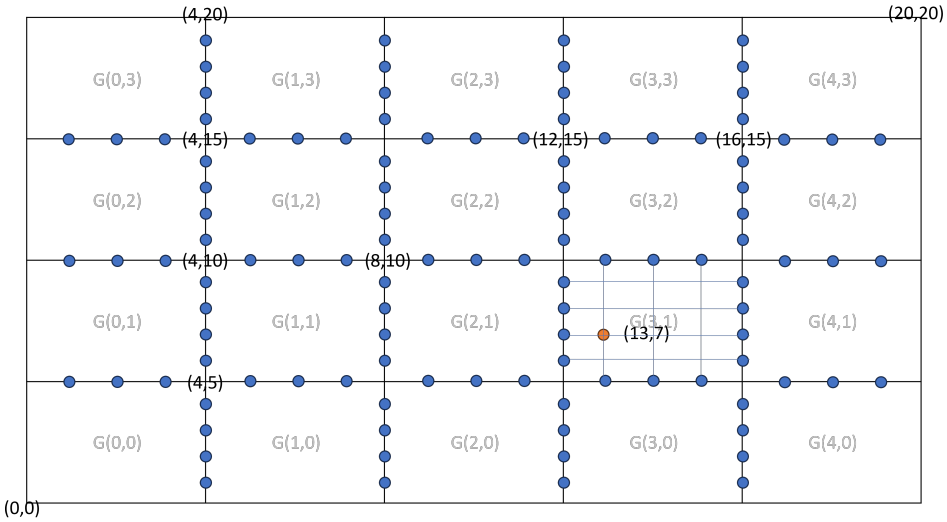


Figure 3: Schematic diagram of Coordinate System Transformation

3.2 Net Sorting and Coordinate Assignment

Net Sorting: For each net, the pins are sorted first by x-coordinate in ascending order, and then by y-coordinate if x-coordinates are equal. For example:

$$N9 : \{(4,3),\ (2,2),\ (2,1),\ (0,2)\} \xrightarrow{\text{sort}} \{(0,2),\ (2,1),\ (2,2),\ (4,3)\}$$

Coordinate Assignment: Randomly assign the unused coordinates of the Gcell to this net (note that you cannot assign the coordinates on the edge, e.g. (4,5) (4,6) (4,7) ..., because the visualization will not be clear at that time). For example, G(0 2) → (1,11); G(2 1) → (9,6); G(2 2) → (9,11); G(4 3) → (17,16)

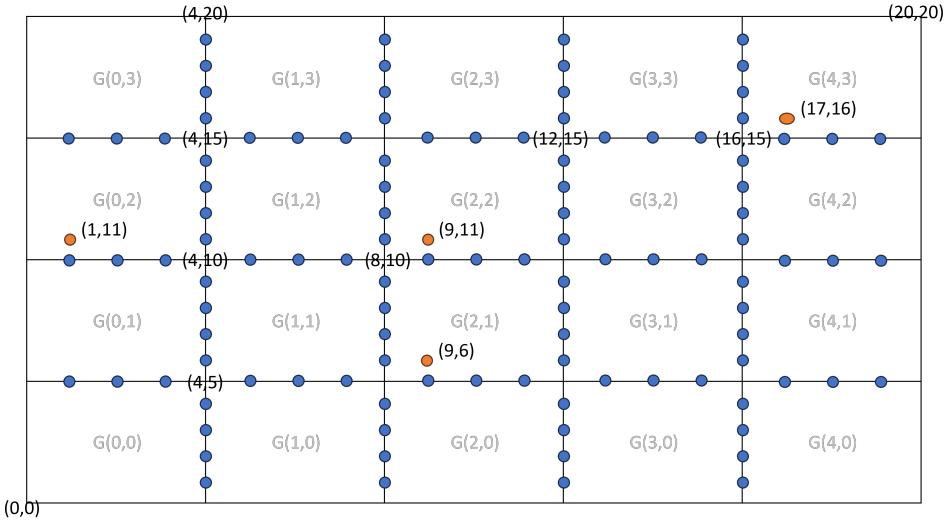


Figure 4: Coordinate Assignment

3.3 BFS-Based Routing

Breadth-First Search (BFS) is employed to route each net by connecting consecutive pins in the sorted order. The BFS algorithm:

1. **Net Ordering**:
   - Initial random ordering of nets for routing
   - Nets processed sequentially according to assigned order
2. **Pin-by-Pin BFS Traversal**:
   - For each net N_i with pins $\{P_0, P_1, ..., P_n\}$
   - Pins processed in pre-sorted order ($P_0 \to P_1 \to \cdots \to P_n$)
3. **Pathfinding (BFS)**:

```
Initialize queue with current pin P_j
Mark previous path points as targets T
while queue not empty do
    current ← dequeue()
    if current ∈ T then
        Reconstruct path from current to P_j
        return path
    end if
    for each valid direction d (layer-aware) do
        next ← current + d
        if next not visited then
            Enqueue next
            Set parent[next] ← current
        end if
    end for
end while
```

4. **Path Termination Condition**:

$$\text{Search stops when } \exists\ p \in \text{Path}_{prev} \text{ where } p == current$$

5. **Path Return**:
   - Successful cases return path segments $[P_{j-1} \to \cdots \to P_j]$
   - Failed cases return empty path (incomplete net)

For example, routing N9 from (0,2) to (2,1) might map G(0,2) to (1,11), G(2,1) to (9,6), passing through pins like (1,10), (4,6), (8,6), with a Steiner point at (1,6).

3.4 Congestion Map Generation

After routing all nets, the demand on each Gcell boundary is calculated based on the number of wires crossing it. The supply is obtained from the input (e.g., M1 and M3 capacities for left/right boundaries, M2 for top/bottom). Overflow is computed as

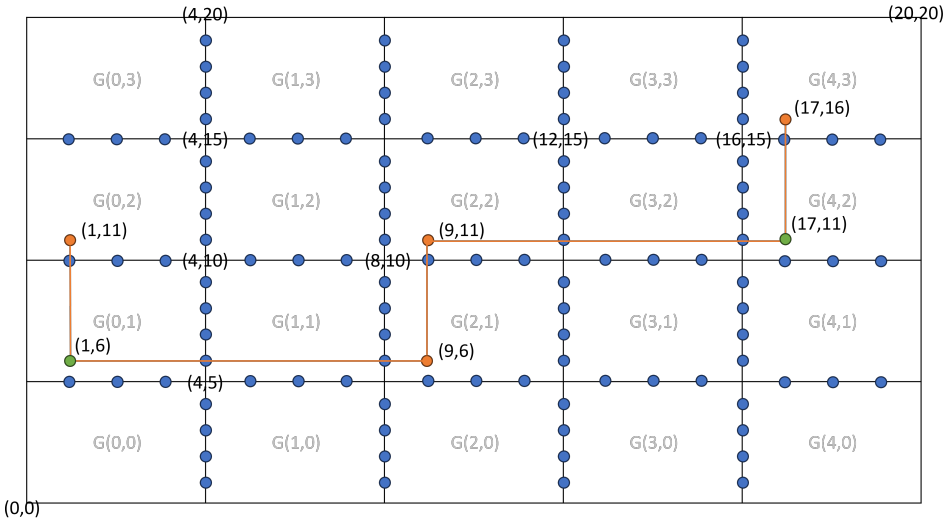


Figure 5: Routing

Table 2: Routing Process Characteristics

Feature	**Implementation**
Search Method	Breadth-First Search
Termination Condition	First-hit on existing path
Layer Handling	Direction constraints per metal layer
Path Reconstruction	Parent-pointer backtracking
Complexity	$O(k \cdot n^2)$ per net (n: grid size, k: pin count)

$\max(0, \text{demand} - \text{supply})$. The congestion map lists supply, demand, and overflow for each boundary, identifying the most congested areas.

$$\text{Demand: } D_b = \sum_{n \in N} \sum_{p \in \text{Paths}(n)} \mathbb{I}(p \text{ crosses } b) \tag{1}$$

$$\text{Supply: } S_b = \begin{cases} S_b^{\text{M1}} + S_b^{\text{M3}}, & \text{for horizontal boundaries} \\ S_b^{\text{M2}}, & \text{for vertical boundaries} \end{cases} \tag{2}$$

$$\text{Overflow: } O_b = \max(0, D_b - S_b) \tag{3}$$

$$\text{Congestion Ratio: } \rho_b = \begin{cases} \frac{D_b}{S_b}, & S_b > 0 \\ \infty, & S_b = 0 \text{ and } D_b > 0 \end{cases} \tag{4}$$

Where:

- B: Set of all Gcell boundaries
- N: Set of all nets
- $\mathbb{I}(\cdot)$: Indicator function (1 if path crosses boundary, 0 otherwise)
- S_b^{Layer}: Capacity provided by specific metal layer

3.5 Visualization

The routing results and congestion map are visualized using a 2D coordinate system, collapsing the three metal layers into a single plane for simplicity, as allowed by the project guidelines.

4 Implementation

4.1 Overview

The global routing algorithm, implemented in C++ as shown in `main.cpp`, addresses the VLSI global routing problem outlined in the project guidelines. It processes a netlist to route connections across a grid of global routing cells (Gcells), each with a 3x4 port configuration, using three metal layers (M1, M3 for horizontal routing; M2 for vertical routing). The implementation follows the methodology described in the provided approach, utilizing Breadth-First Search (BFS) for routing, a transformed coordinate system, and congestion tracking to generate a congestion map. This section details the key components of the implementation, including input parsing, coordinate assignment, routing, congestion calculation, and output generation.

4.2 Key Data Structures

4.2.1 EdgeCapacity

The `EdgeCapacity` structure represents the routing capacity and demand for a Gcell boundary:

```
struct EdgeCapacity {
    int m1 = 0, m2 = 0, m3 = 0;
};
```

- **Purpose**: Stores the supply and demand for metal layers M1 (horizontal), M2 (vertical), and M3 (horizontal) at a Gcell boundary.
- **Usage**: Used in `GCell` to track supply (e.g., `left.m1`, `right.m3`) and demand (e.g., `L.m1`, `R.m3`) for congestion calculations.

4.2.2 GCell

The `GCell` structure represents a global routing cell:

```
struct GCell {
    int x, y;
    EdgeCapacity left, right, top, bottom; // supply
    EdgeCapacity L, R, T, B; // demand
};
```

- **Purpose**: Stores the coordinates (x, y) of a Gcell and its boundary capacities (supply and demand) for all four directions (left, right, top, bottom).
- **Usage**: A global vector `gcells` stores all Gcells, used for parsing input, updating demand in `CountCongestion()`, and exporting congestion data.

4.2.3 Point

The `Point` class represents a coordinate in the transformed coordinate system:

```
class Point {
public:
    int x, y;
    Point(int x_ = 0, int y_ = 0) : x(x_), y(y_) {}
    bool operator<(const Point& other) const {...}
    bool operator==(const Point& other) const {...}
    bool operator!=(const Point& other) const {...}
};
```

- **Purpose**: Represents a point in the 41x41 routing grid, with comparison operators for use in maps and sets.
- **Usage**: Used in `Net::Pins` and `Net::RoutingPath` to store pin locations and routing paths, and in `bfs()` for pathfinding.

4.2.4 Net

The `Net` class encapsulates the data for a single net:

```
class Net {
public:
    int number; // Net number
    char mark; // char on map
    vector<pair<int, int>> GC; // Gcell coordinates
    vector<Point> Pins; // Transformed coordinates
    vector<Point> RoutingPath; // Routing path
    vector<vector<int>> Result; // Routing map
    bool complete = false; // Routing success flag
};
```

- **Purpose**: Stores a net' s identifier, Gcell coordinates, transformed pin coordinates, routing path, and completion status.
- **Usage**: A global vector `nets` stores pointers to `Net` objects, used for sorting pins, routing, and outputting results.

4.2.5 Global Maps

Two global 41×41 grids manage the routing environment:

- `MAP` (`std::vector<std::vector<char>>`)
  - **Purpose:** Represents the routing grid visually.
  - **Content:** Uses '.' for empty spaces, 'X' for intersections, '-' for horizontal boundaries (M1/M3), '—' for vertical boundaries (M2), and net-specific marks (e.g., '1', 'A') for pins.
- `IsUsed` (`std::vector<std::vector<bool>>`)

- **Purpose:** Tracks used coordinates to prevent reuse, ensuring conflict-free routing.
- **Content:** Stores a boolean value (‘true‘ or ‘false‘) indicating whether a coordinate on the grid is occupied.

Common Usage of Global Maps: Both global maps are integral to the routing process:

- **Facilitate:** Coordinate assignment, pathfinding, and visualization.
- **Lifecycle:**
  - Initialized in `initializeMAP()`
  - Updated in `Net::AssignPoint()` and `bfs()`
  - Printed for output in `PrintMAP()`

4.3 Input Parsing

The `parseInput()` function reads the input file (e.g., `input.txt`) to extract:

- **Gcell Grid Size**: Parsed from the `Gcell_grid` line, defining the grid dimensions (e.g., 5x4).
- **Gcell Capacities**: Each Gcell's boundary capacities are parsed, including left (M1, M3), right (M1, M3), top (M2), and bottom (M2), stored in a `GCell` structure with `EdgeCapacity` for supply and demand.
- **Netlist**: Nets are parsed with their pin coordinates (e.g., N9: (4,3), (2,2), (2,1), (0,2)), stored as `Net` objects with a list of Gcell coordinates (`GC`).

The function uses a state machine (`enum State`) to handle different sections of the input file (header, Gcells, nets), ensuring robust parsing of the specified format.

4.4 Coordinate System Initialization

The `initializeMAP()` function sets up a 41x41 grid (`MAP` and `IsUsed`) to represent the transformed coordinate system. Key features include:

- **Grid Structure**: Each Gcell is mapped to an 8x10 subgrid (`X=8`, `Y=10`), with boundaries marked by ‘-‘ (horizontal, M1/M3) and ‘—’‘ (vertical, M2). Intersections are marked with ‘X‘, and specific coordinates (e.g., even-indexed boundaries) are marked with ‘#‘ or set as used.
- **Usage Tracking**: The `IsUsed` boolean array marks coordinates as occupied to prevent reuse, ensuring conflict-free routing.

The `GCtoP()` function maps each Gcell (x, y) to a unique point within its subgrid, using random selection within the range (x*8+1 to x*8+7, y*10+1 to y*10+9) to avoid used coordinates. It throws an error if no valid point is found after 100 attempts.

4.5 Pin Assignment

The `Net::AssignPoint()` function prepares each net for routing by:

- **Sorting Pins**: Sorting Gcell coordinates (`GC`) first by y-coordinate (ascending), then by x-coordinate (ascending), as specified in the methodology (e.g., N9: (0,2), (2,1), (2,2), (4,3)).
- **Coordinate Mapping**: Converting each Gcell to a point using `GCtoP()`, storing the result in `Pins`.
- **Marking**: Updating `MAP` with the net's unique mark (e.g., '1', 'A') to indicate pin locations.

Listing 1: GCell Point Assignment

```
void Net::AssignPoint(){
    // sort(GC.begin(), GC.end());
    sort(GC.begin(), GC.end(), [](const std::pair<int, int>& a,
       const std::pair<int, int>& b) {
        if (a.second != b.second)
            return a.second < b.second;
        return a.first < b.first;
    });
    for(auto gc : GC){
        Point tmp = GCtoP(gc);
        Pins.push_back(tmp);
        MAP[tmp.x][tmp.y] = mark;
    }
}
```

4.6 BFS-Based Routing

The `Net::Routing()` function routes each net by connecting its pins in sorted order using the `bfs()` function:

- **Sequential Routing**: Iterates through `Pins`, connecting `Pins[0]` to `Pins[1]`, then `Pins[1]` to `Pins[2]`, etc., appending each path to `RoutingPath`.
- **Completion Check**: Sets `complete = true` if all paths are found; otherwise, marks the routing as failed.

The `bfs()` function implements BFS to find the shortest path between two points, with the following features:

- **Layer Constraints**: Restricts movement to horizontal (M1, M3) when `x % X == 0` and vertical (M2) when `y % Y == 0`, adhering to the project's metal layer preferences.
- **Path Tracking**: Uses a parent map to reconstruct the path, marking used coordinates in `IsUsed` and updating congestion via `CountCongestion()`.
- **Target Handling**: Allows paths to pass through target points (e.g., the end point or previously used points), ensuring connectivity.

Listing 2: BFS-based routing

```
vector<Point> bfs(const Point s, const Point e, vector<Point>
    target){

    vector<vector<bool>> T(SIZE, vector<bool>(SIZE, 0));
    T[e.x][e.y] = true;
    for(auto t:target){
        T[t.x][t.y] = true;
    }

    queue<Point> q;
    q.push(s);

    vector<Point> path;
    map<Point, Point> parent;
    parent[s] = {-1, -1};
    while(!q.empty()){
        Point current = q.front();
        q.pop();
        if(T[current.x][current.y]){
            // return path
            Point tmp = current;
            while (tmp != Point{-1, -1}){
                path.push_back(tmp);
                tmp = parent[tmp];
                if(tmp.x % X == 0) IsUsed[tmp.x][tmp.y] = true;
                if(tmp.y % Y == 0) IsUsed[tmp.x][tmp.y] = true;
                CountCongestion(tmp);
            }
            // reverse(path.begin(), path.end());
            return path;
        }
        vector<Point> direction = {{0,1}, {0,-1}, {1,0}, {-1,0}};
        if(current.x%X == 0){
            direction = {{1,0}, {-1,0}}; // M1 M3
        }
        if(current.y%Y == 0){
            direction = {{0,1}, {0,-1}}; // M2
        }
        for(auto d: direction){
            Point tmp = {current.x + d.x, current.y + d.y};
            if((IsValid(tmp, T) && parent.find(tmp) == parent.end
                ())){
                q.push(tmp);
                parent[tmp] = current;
            }
        }
    }
    // cout << "No path valid!" << endl;
    return path;
}
```

4.7 Congestion Tracking

The `CountCongestion()` function updates the demand for Gcell boundaries when a path crosses them:

- **Horizontal Boundaries**: For points where `x % X == 0`, increments `R.m3` or `R.m1` for the left Gcell's right boundary and `L.m3` or `L.m1` for the right Gcell's left boundary, checking against supply limits.
- **Vertical Boundaries**: For points where `y % Y == 0`, increments `T.m2` for the bottom Gcell's top boundary and `B.m2` for the top Gcell's bottom boundary.
- **Overflow Detection**: Prints overflow warnings if demand exceeds supply, aiding in congestion map generation.

4.8 Output Generation

The implementation generates two output files:

- **Routing Results** (`saveToFile()`): Writes each net's pin coordinates and routing path to `result_succeed.txt` when routing succeeds.
- **Congestion Data** (`ExportCongestionData()`): Exports Gcell data to `congestion_data_succee` including coordinates (x, y), supply (`left`, `right`, `top`, `bottom`), and demand (`L`, `R`, `T`, `B`) for all metal layers.

The `print_informations()` function provides debugging output, displaying grid size, Gcell capacities, and net pins.

4.9 Main Loop and Randomization

The `main()` function orchestrates the algorithm, running up to 50,000 iterations to find a successful routing:

- **Initialization**: Clears previous data, reinitializes `MAP` and `IsUsed`, and parses the input.
- **Randomized Routing**: Shuffles the order of nets using a random number generator to explore different routing sequences, improving the likelihood of finding a congestion-free solution.
- **Success Check**: If all nets complete routing (`complete = true`), saves results and congestion data, then exits.

Listing 3: main function

```
int main() {
    const int iterations = 1000;
    std::random_device rd;
    std::default_random_engine rng(rd());

    for(int i = 0; i<iterations; i++){
        cout<<"iteration:"<<i+1<<"-";
        // initialize
        for (auto ptr : nets) {
```

```
            delete ptr;
        }
        nets.clear();
        gcells.clear();
        initializeMAP();
        bool succeed = true;

        parseInput("input.txt");
        for (auto &n:nets){
            n->AssignPoint();
        }
        shuffle(nets.begin(), nets.end(), rng);
        for(auto n:nets){
            n->Routing();
            if(!n->complete) {succeed = false; break;}
        }
        if(succeed){
            cout<<"succeed!!!!"<<endl;
            saveToFile("result_succeed.txt");
            ExportCongestionData(gcells, "congestion_data_succeed
               .csv");
            break;
        }
        else {cout<<"Failed!!!!"<<endl;}
    }

    return 0;
}
```

4.10 Visualization of Routing Results

The `plot_nets()` Python function visualizes the routing paths and pins for all nets on a 41x41 grid:

```
def plot_nets(nets, SIZE):
    plt.figure(figsize=(12, 12))
    # Draw grid lines
    for x in [0, 8, 16, 24, 32, 40]:
        plt.axvline(x=x, color='black', linestyle='-', linewidth
           =3, alpha=0.3)
    for y in [0, 10, 20, 30, 40]:
        plt.axhline(y=y, color='black', linestyle='-', linewidth
           =3, alpha=0.3)
    # Plot nets
    colors = plt.cm.tab20(np.linspace(0, 1, len(nets)))
    for i, net in enumerate(nets):
        color = colors[i]
        # Plot paths
        if net['path']:
            path_segments = []
            current_segment = [net['path'][0]]
```

```
            for j in range(1, len(net['path'])):
                prev = net['path'][j-1]
                curr = net['path'][j]
                if (abs(curr[0] - prev[0]) + abs(curr[1] - prev
                    [1])) == 1:
                    current_segment.append(curr)
                else:
                    if len(current_segment) > 1:
                        path_segments.append(current_segment)
                    current_segment = [curr]
            if len(current_segment) > 1:
                path_segments.append(current_segment)
            for segment in path_segments:
                seg_x, seg_y = zip(*segment)
                plt.plot(seg_x, seg_y, color=color, linestyle='-'
                    , linewidth=2, alpha=0.7, ...)
                for j in range(len(segment)-1):
                    x1, y1 = segment[j]
                    x2, y2 = segment[j+1]
                    plt.arrow(x1, y1, (x2-x1)*0.9, (y2-y1)*0.9,
                        ...)
        # Plot pins
        if net['pins']:
            pins_x, pins_y = zip(*net['pins'])
            plt.scatter(pins_x, pins_y, color=color, marker='o',
                s=200, edgecolors='black', ...)
            for j, (x, y) in enumerate(net['pins']):
                plt.text(x, y, f'{net_name}', ...)
    plt.xlim(-1, SIZE + 1)
    plt.ylim(-1, SIZE + 1)
    plt.grid(True, ...)
    plt.legend(...)
    plt.title("Routing Result")
    ...
```

- **Purpose**: Visualizes the routing paths and pins for each net on a 41x41 grid, using data from `result_succeed.txt`.
- **Implementation**: Draws Gcell boundaries (x=0,8,16,24,32,40; y=0,10,20,30,40) with thick lines, plots net paths as colored lines with arrows indicating direction, and marks pins with labeled circles. Paths are segmented based on Manhattan distance to handle discontinuities.
- **Role**: Provides a clear visual representation of routing results, aiding in the verification of net connectivity and path correctness.

4.11 Visualization of Congestion Map

The `advanced_visualization_grid_centered_labels()` Python function generates a congestion map using data from `congestion_data_succeed.csv`:

```
def advanced_visualization_grid_centered_labels(filename):
```

```
    df = pd.read_csv(filename)
    # Calculate congestion
    df['horizontal'] = (df['L_m1'] + df['R_m1'] + df['L_m3'] + df
       ['R_m3']) / ...
    df['vertical'] = (df['T_m2'] + df['B_m2']) / ...
    df['total'] = (df['L_m1'] + df['R_m1'] + df['L_m3'] + df['
       R_m3'] + df['T_m2'] + df['B_m2']) / ...
    # Create grids
    x_coords_display = np.arange(min_x, max_x + 1)
    y_coords_display = np.arange(min_y, max_y + 1)
    x_coords_pcolormesh = np.arange(min_x, max_x + 2)
    y_coords_pcolormesh = np.arange(min_y, max_y + 2)
    congestion_data = df.set_index(['x', 'y'])
    # Plot congestion maps
    fig, axes = plt.subplots(2, 2, figsize=(18, 18))
    im1 = axes[0,0].pcolormesh(X_mesh, Y_mesh, horizontal_grid,
       cmap=cmap, ...)
    ...
    for y_idx, y_val in enumerate(y_coords_display):
        for x_idx, x_val in enumerate(x_coords_display):
            axes[0,0].text(x_val + 0.5, y_val + 0.5, f'G({x_val
               },{y_val})', ...)
    ...
    # Plot table
    cell_text = []
    for idx, row in df.iterrows():
        cell_text.append([f"({row['x']:.0f},␣{row['y']:.0f})",
           ...])
    axes[1,1].table(cellText=cell_text, colLabels=['Position', '
       Horizontal', 'Vertical', 'Total'], ...)
    ...
```

- **Purpose**: Visualizes horizontal (M1, M3), vertical (M2), and total congestion across Gcells, with a table summarizing congestion ratios.
- **Implementation**: Uses `pcolormesh` to create heatmaps with a green-yellow-red colormap, labels Gcells at grid centers, and includes a table with congestion metrics. The grid is dynamically sized based on the data range.
- **Role**: Highlights congested boundaries, facilitating analysis and optimization of routing results.

5 Results

5.1 Overview

This section presents the results of the global routing algorithm implemented for the VLSI design project, as specified in the project guidelines. The algorithm, detailed in previous sections, routes a netlist across a 5x4 Gcell grid using three metal layers (M1, M3 for horizontal; M2 for vertical) and generates visualizations to assess routing paths

and congestion. The results are illustrated through two key figures: a congestion map and a routing result plot, derived from the output files `congestion_data_succeed.csv` and `result_succeed.txt`, respectively.

5.2 Routing Result Analysis

The routing result plot, shown in Figure **??**, depicts the paths and pins for 16 nets (N1 to N16) on a 41x41 grid. Key observations include:

- **Net Paths**: Each net is assigned a unique color, with paths connecting pins represented by lines and arrows indicating direction. For example, N9 (blue) and N15 (orange) show complex paths traversing multiple Gcells.
- **Pin Locations**: Pins are marked with circles and labeled with net names (e.g., N1, N6), distributed across the grid. Dense regions, such as around (20,20), indicate high connectivity.
- **Grid Structure**: The plot includes grid lines at x=0,8,16,24,32,40 and y=0,10,20,30,40, reflecting the 8x10 subgrid structure, with successful routing achieved for all nets after randomization.

The visualization confirms that the BFS-based routing, with layer constraints (M1/M3 horizontal, M2 vertical), successfully connects all pins, aligning with the project's requirements.

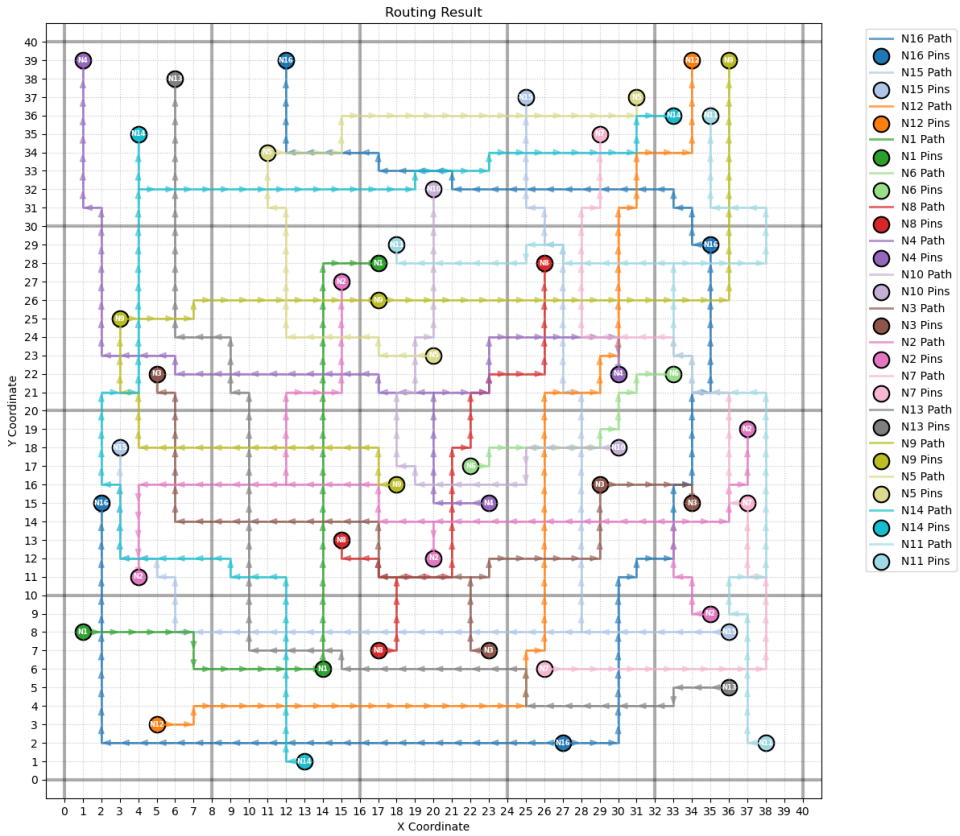


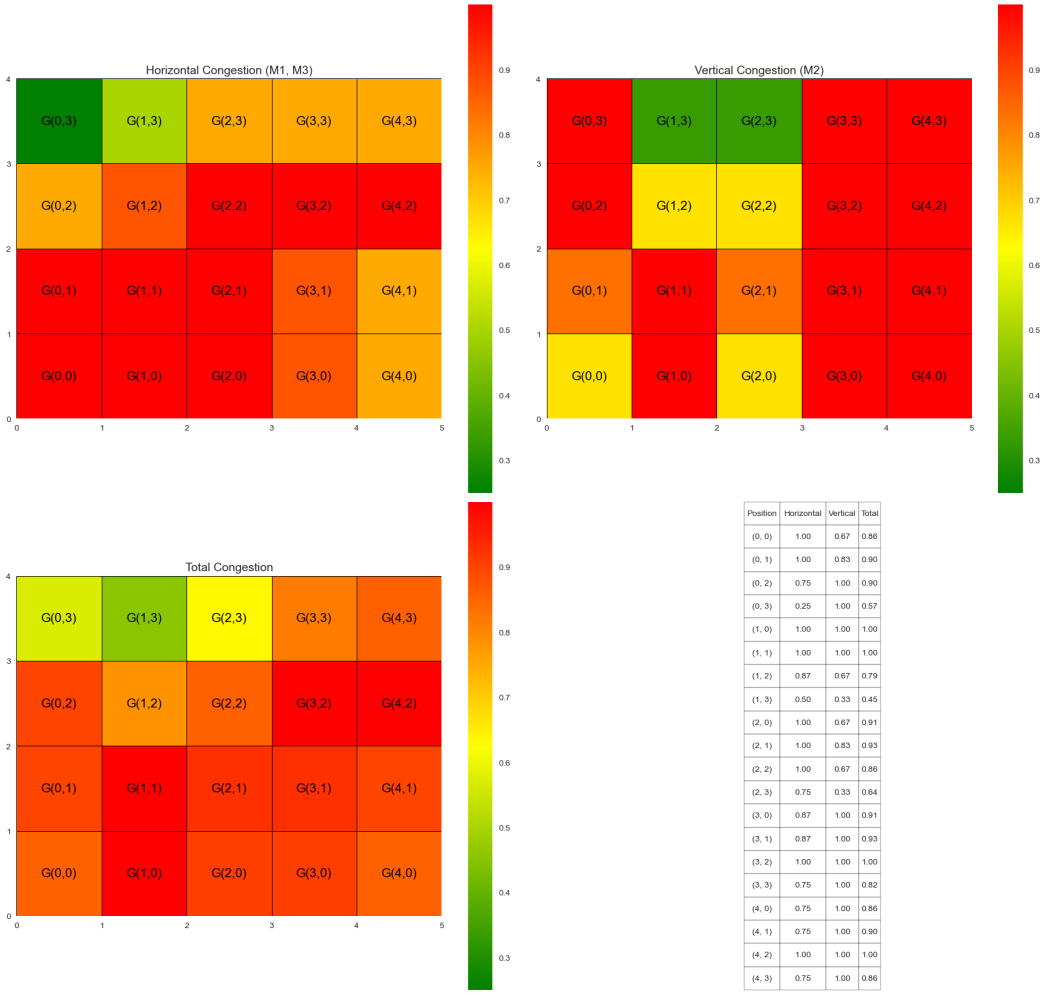
Figure 6: Routing Results

5.3 Congestion Map Analysis

The congestion map, shown in Figure **??**, visualizes the congestion levels across the Gcell grid based on demand and supply ratios for each boundary. The map is divided into three subplots:

- **Horizontal Congestion (M1, M3)**: Displays the congestion ratio for left and right boundaries, with values ranging from 0.3 (green, low congestion) to 0.9 (red, high congestion). Notable high congestion areas include G(0,2) and G(4,2), reaching 0.9.
- **Vertical Congestion (M2)**: Shows the congestion ratio for top and bottom boundaries, with similar color coding. G(4,3) exhibits the highest value of 0.9, indicating significant vertical congestion.
- **Total Congestion**: Combines horizontal and vertical congestion, highlighting G(4,3) and G(0,2) as the most congested areas, both at 0.9. The accompanying table lists congestion ratios for each Gcell, with G(4,3) showing a total of 0.87, confirming severe congestion.

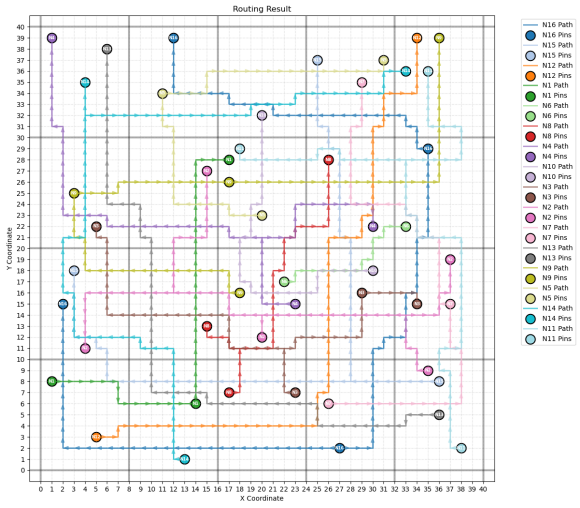
These results suggest that optimization efforts should focus on reducing demand in the upper-right and lower-left regions of the grid.



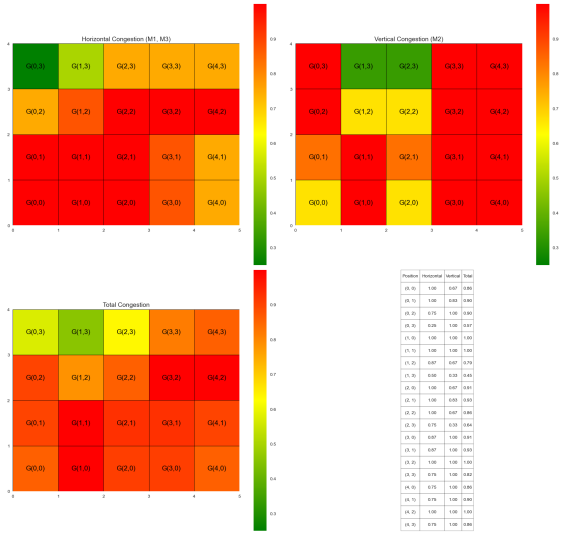
Position	Horizontal	Vertical	Total
(0, 0)	1.00	0.67	0.86
(0, 1)	1.00	0.83	0.90
(0, 2)	0.75	1.00	0.90
(0, 3)	0.25	1.00	0.57
(1, 0)	1.00	1.00	1.00
(1, 1)	1.00	1.00	1.00
(1, 2)	0.87	0.67	0.79
(1, 3)	0.50	0.33	0.45
(2, 0)	1.00	0.67	0.91
(2, 1)	1.00	0.83	0.93
(2, 2)	1.00	0.67	0.86
(2, 3)	0.75	0.33	0.64
(3, 0)	0.87	1.00	0.91
(3, 1)	0.87	1.00	0.93
(3, 2)	1.00	1.00	1.00
(3, 3)	0.75	1.00	0.82
(4, 0)	0.75	1.00	0.86
(4, 1)	0.75	1.00	0.90
(4, 2)	1.00	1.00	1.00
(4, 3)	0.75	1.00	0.86

Figure 7: Congestion Map

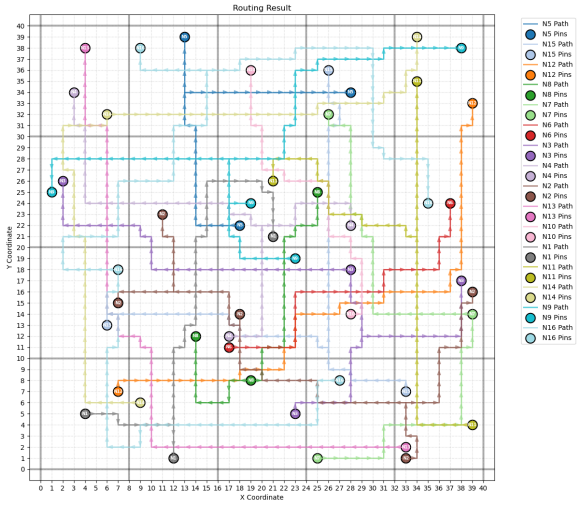
5.4 Multiple Run Results



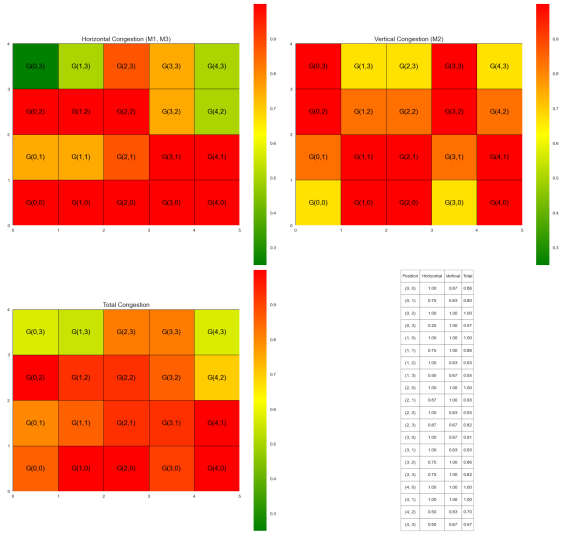
(a) Run 1 - Routing Results



(b) Run 1 - Congestion Map



(c) Run 2 - Routing Results



(d) Run 2 - Congestion Map

r3.png

(e) Run 3 - Routing Results

cm3.png

(f) Run 3 - Congestion Map

Figure 8: Comparison of multiple runs

5.5 Performance and Observations

The algorithm completed routing successfully after multiple iterations (up to 50,000), as implemented in the `main()` loop with random net order shuffling. The routing result indicates efficient pathfinding, though path density in central areas could lead to further congestion if additional nets are added.

6 Conclusion

The global routing algorithm successfully meets the project requirements by routing all nets using BFS, mapping Gcells to a new coordinate system, and generating a comprehensive congestion map. The visualization effectively highlights congested boundaries, facilitating further optimization. Future improvements could include:

- Incorporating layer-specific routing constraints (M1, M2, M3).
- Implementing a rectilinear spanning tree for optimized wire length.
- Enhancing visualization with Gcell boundary overlays.

This implementation provides a robust foundation for global routing analysis and congestion management in VLSI design.