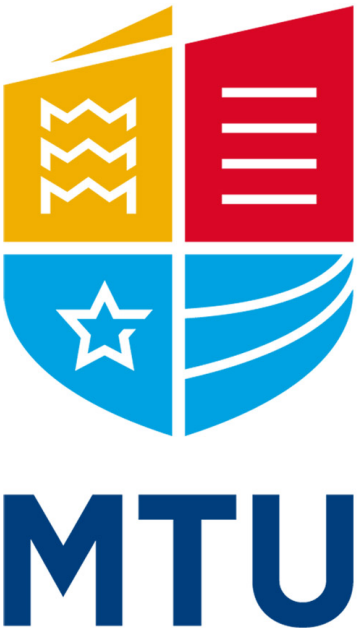


Algorithmic Techniques for Minimizing Congestion in VLSI Global Routing

by

Yi Ming Tan

This thesis has been submitted in partial fulfillment for the degree of Bachelor of Science in Software Development

in the
Faculty of Engineering and Science
Department of Computer Science

May 2023

Declaration of Authorship

This report, Algorithmic Techniques for Minimizing Congestion in VLSI Global Routing, is submitted in partial fulfillment of the requirements of Bachelor of Science in Software Development at Munster Technological University Cork. I, Yi Ming Tan, declare that this thesis titled, Algorithmic Techniques for Minimizing Congestion in VLSI Global Routing and the work represents substantially the result of my own work except where explicitly indicated in the text. This report may be freely copied and distributed provided the source is explicitly acknowledged. I confirm that:

- This work was done wholly or mainly while in candidature Bachelor of Science in Software Development at Munster Technological University Cork.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Munster Technological University Cork or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Yi Ming Tan

Date: 19 September 2022

MUNSTER TECHNOLOGICAL UNIVERSITY CORK

Abstract

Faculty of Engineering and Science
Department of Computer Science

Bachelor of Science

by Yi Ming Tan

Even today, the physical design of IC (Integrated Circuits) remains one of the most challenging but exciting areas in the field of EDA (Electronic Design Automation). Among many steps in physical design, routing is one of the most crucial ones. Routing is the process of creating physical connections based on logical connectivity. There are mainly three stages in the routing operation: Global Routing, Track Assignment and Detailed Routing. Global Routing is a step in which a coarse route for each net is generated while attempting to optimize some objective function, such as total wire length and circuit timing.

In this final-year project, the aim is to explore various algorithmic techniques used in Global Routing to optimize congestion and minimize the total interconnect/wire length on a chip, namely: Lee Maze Router, Dijkstra Algorithm, and A* Search Algorithm. We will be comparing the performance in terms of these algorithms' time and space complexity. Finally, we will plot a heat map to visualize the congestion of the layout.

Acknowledgements

I want to thank my ex-manager Mohamed Wahbi, and other colleagues from my 2022 Qualcomm internship for suggesting this topic idea. I would also like to thank my project supervisor Paul Davern for his continuous help and support throughout this academic year in making this research report come to fruition. Lastly, I want to thank all my friends and family for their endless encouragement.

Contents

List of Figures

List of Tables

Abbreviations

IC	**I**ntegrated **C**ircuit
EDA	**E**lectronic **D**esign **A**utomation
VLSI	**V**ery **L**arge **S**cale **I**ntegration
LEF	**L**ibrary **E**xchagne **F**ormat
DEF	**D**esign **E**xchange **F**ormat
PnR	**P**lacement and **R**outing
CAD	**C**omputer-**A**ided **D**esign
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
MST	**M**inimum **S**panning **T**ree
RMST	**R**ectilinear **M**inimum **S**panning **T**ree
SMT	**S**teiner **M**inimum **T**ree
RSMT	**S**teiner **R**ectilinear **M**inimum **T**ree
DAG	**D**irected **A**cyclic **G**raph
LVS	**L**ayout **VS** **S**chematic
DRC	**D**esign **R**ule **C**heck
HPWL	**H**alf **P**erimeter **W**ire **L**ength

Dedicated to my family and friends that believe in me and have been supportive throughout my whole life

Chapter 1

Introduction

Very-Large-Scale Integration (VLSI) is a field of electrical engineering that involves designing and fabricating complex *integrated circuits (ICs)* that contain millions or billions of transistors and other components. These circuits are used in various electronic devices, including computers, smartphones, and other electronic systems. Modern VLSI is tiny; flagship smartphones in 2022 are powered by VLSI chips that are only 4nm. To give some context into how small that is, you can fit 1.25 trillion 4nm chips on a silicon wafer around the size of a fingernail.

At every stage of the ASIC design cycle, there will be EDA tools that will help to implement ASIC design more efficiently. The goal is to make the chip low in delay, cost and power. When we talk about integrated circuits, we refer to nanometer-sized electrical chips. The complexity of VLSI makes the manual design approach impractical and essentially impossible. Design automation with EDA tools is the order of the day. So, engineers must follow this design flow methodology and use EDA tools to yield a design with the highest precision and accuracy. So, design accuracy must be achieved before fabrication; we cannot afford mistakes at the production stage because the costs would be highly exorbitant.

Each step in the design flow is an exciting and complex problem. In this paper, we will only be focusing on the routing stage. Routing is the "R" in PnR. PnR is the short form for Placement and Routing. Placement is the step in figuring out where those millions of gates will be placed on the chip. Routing comes after placement. Routing is the process of connecting millions of wires to all the gates on the chip. There are mainly two steps in routing: Global Routing and Detailed Routing.

Global routing is a critical step in the VLSI design process, as it determines the overall interconnectivity of the circuit and plays a significant role in determining the performance and efficiency of the final IC. It does not put any physical wire down; it only gives a coarse route to where the wire should go. In other words, global routing is a hierarchical decomposition of routing. Instead of decomposing the entire chip surface into a gigantic grid of tiny cells where each cell can only fit one unit of wire, the global router decomposes the chip surface into a coarse grid of cells, where each can fit perhaps a couple of hundred of wire units, depending on the capacity constraints.

In Detailed Routing, wires will be embedded physically on the chip. Since the global router from the previous has already defined a coarse routing region for the wires, the detail router will only run within the defined region rather than the entire chip, making it more efficient. In short, the detailed router follows the solution obtained in global routing to find the exact routing solution.

Overall, the goal of global routing in VLSI design is to find an optimal solution that meets the performance and efficiency requirements of the circuit while also considering other factors, such as manufacturability and reliability. The quality of global routing will affect timing, power and density in the chip area. So, the global routing step is crucial in the design cycle. The field of global routing in VLSI design is constantly evolving, with new algorithms and approaches being developed and evaluated to address the increasing complexity and variability of modern VLSI designs.

1.1 Motivation

VLSI is an important topic now because of its wide-ranging applications, the increasing complexity and performance requirements of modern VLSI designs, and the impact of VLSI technology on our everyday lives. It is also a very competitive field, with companies and research organizations constantly seeking new and innovative ways to improve the performance and efficiency of ICs.

Global Routing is one of the most critical roles in VLSI design as it determines the overall interconnectivity of the circuit and plays a significant role in determining the performance and efficiency of the final IC.

1.2 Contribution

The main contribution of this project is to allow new VLSI students or anyone interested in Global Routing to understand how Global Routing works on a high level. This project

will provide a user-friendly interface to allow anyone to route a netlist input and generate an output of the routed netlist. Moreover, it is to provide visualization of the output results in hopes of aiding them in understanding how the wires are routed, what areas are more congested, etc.

1.3 Structure of This Document

The structure of this document is as follows:

- Chapter 1 contains an introduction to this project, the motivation behind the project, the main feature and requirements of the project and the main contributions of this project.
- Chapter 2 contains research on the main topic of the project, a literature review and aims to describe what has and is being done.
- Chapter 3 outlines what the problem is that this project is trying to solve, what the objectives are, what are the functional and non-functional requirements of the project.
- Chapter 4 describes the implementation approach and overall system architecture of the project, use cases, risk assessment, methodology, detailed timeline for the implementation phase, evaluation plan and a prototype of how the project is expected to look like.
- Chapter 5 consists of the detailed implementation work of this project. It discusses the difficulties encountered and the actual solution approach.
- Chapter 6 consists of the testing and evaluation of this project. It discusses the metrics used for the systematic testing as well as the final results.
- Chapter 7 consists of the project's solution review, project review, conclusion and future work of this project.

Chapter 2

Background

2.1 Thematic Area within Computer Science

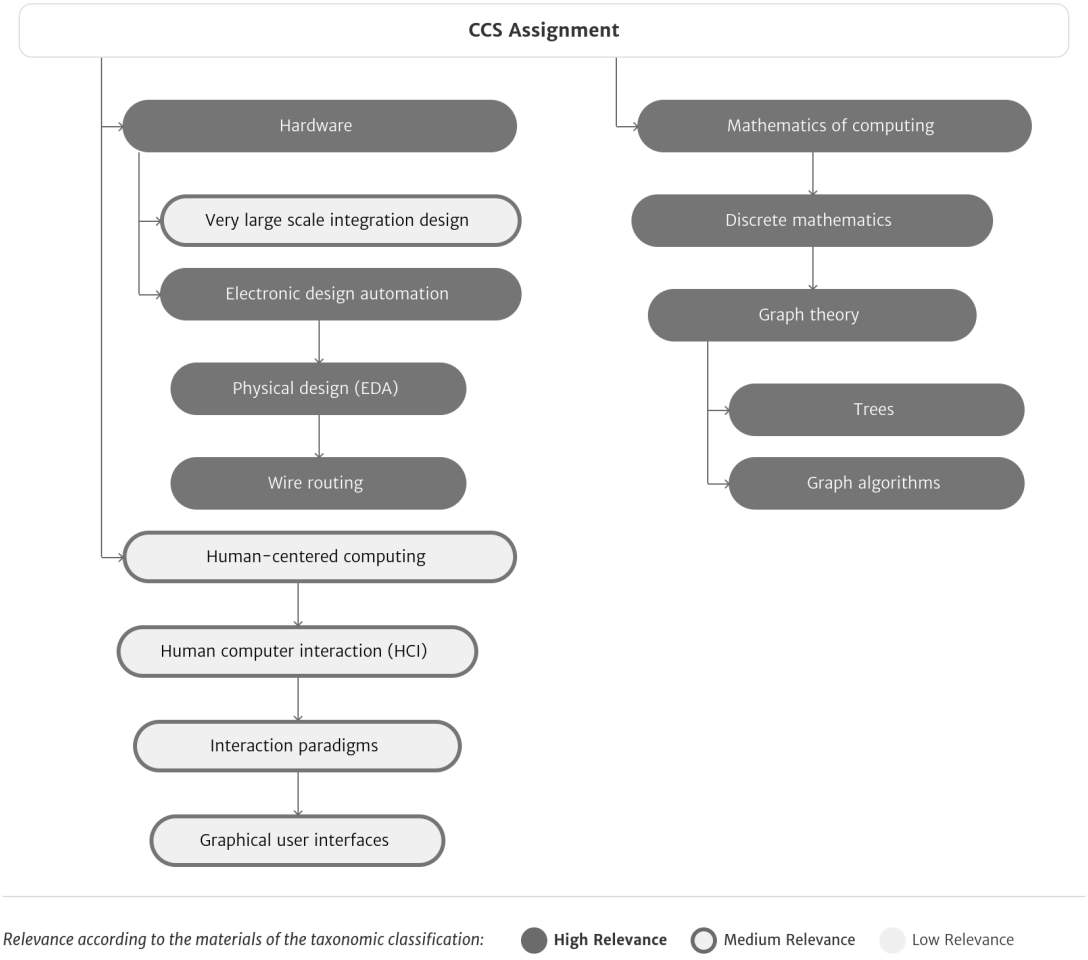


FIGURE 2.1: A chart of Computing Classification of this project

The core topic of the project is to create a resourceful and informative GUI application tool to visualize and analyze the different algorithms used to implement a VLSI global router. The application will parse the netlist input files and represent the layout in a graph data structure. Then, the application will run the different algorithms to route the entire netlist and evaluate the performance and accuracy between them. There are several core areas that this project falls under:

1. *VLSI Wire Routing* - this area covers the concepts and terminologies of VLSI and wire routing
2. *Graph Algorithms* - this area covers all the various routing algorithm concepts and implementations
3. *Graphical User Interfaces* - this area covers the design and implementation of the GUI application

The main areas of Computer Science this project falls under are:

1. *Mathematics of computing* - this area covers all *Discrete Mathematics* topics, including *Graph Theory*, which is where graph algorithms and trees fall under
2. *Hardware* - this area covers VLSI and EDA topics, namely *Physical Design*, which is where wire routing falls under
3. *Human-centered computing* - this area covers the Graphical User Interfaces topic

All of the mentioned Computing Classification topics and sub-topics will be discussed and expanded on in greater detail in the rest of this chapter.

2.2 A Review of all Core Topics in VLSI Routing

2.2.1 EDA Terminology

Electronic design automation (EDA) is at the core of all technology today. Given an electronic system modelled at the system level, EDA tools support engineers on almost every aspect of the *Integrated Circuit (IC)* flow, from high-level design down to fabrication. Below is a short list of common terms used in EDA, along with their brief definitions.

Physical Design is the process of geometrically arranging the placement of circuit components and their connections within the IC layout. The electrical and physical

properties of the components are retrieved from the library files and technology information, while the layout topology is retrieved from the netlist. The physical design process results in a geometrically and functionally accurate representation in a standard file format such as GDSII Stream [1].

ASIC is an acronym for *Application Specific Integrated Circuits*. It can be either a *full-custom* or *semi-custom* integrated circuit, such as a cell or gate array, made for a *specific* function application. Full-custom design is a design with the fewest constraints and usually results in highly optimized electrical properties. Semi-custom design uses pre-designed or pre-fabricated elements, which is more frequently used as it has lower design complexity, time-to-market and cost [1].

LEF is an acronym for *Library Exchange Format*. It is an ASCII data format file used to describe a standard cell library. It includes all the design rules for routing and the abstract of the cells as well as library data such as layer, via, and macro cell definitions. [2, 3]

DEF is an acronym for *Design Exchange Format*. It is an ASCII data format file containing the design-specific information of an IC. It represents the netlist and circuit layout. It is always used in conjunction with LEF files to represent the complete physical layout of an IC while it is in the design process. DEF files are usually generated by *Place and Route (PnR)* tools to be used as input for post-analysis tools [3].

Cell is an instance of a design or a library primitive inside a design. A cell is usually referred to as a gate in digital circuits. Generally, it is used to refer to either standard cells or macro cells.

Standard Cell is a cell with a pre-defined functionality. It is a cohort of transistors and interconnect structures consisting of a boolean logic or a storage function.

Macro Cell is a cell with no pre-defined dimensions. It could contain millions of transistors inside. Some examples of macro are SRAM and CPU core.

Global Bin also known as a *GCell*, is a single routing region routing, or *grid cell* in a grid-graph-represented layer.

Pin is the electrical terminal for connecting a component. It could be the input or output of a component.

Metal Layer is a process level where design components are placed and routed on. There are many metal layers depending on the foundry and technology node.

Via is a connection between metal layers to connect routing wires to different metal layers.

Net is a wire that connects a set of pins or terminals to have the same potential. A net can either be a 2-pin net or a multi-pin net.

Netlist is a description of the circuit, usually written in languages like Verilog or VHDL. Netlists can be organized in two ways: (1) *pin-oriented*: each component on the circuit has a list of associated nets [Figure 2.2 center], or (2) *net-oriented*: each nethas a list of associated components [Figure 2.2 right] . In short, a netlist is a list of all nets[1].

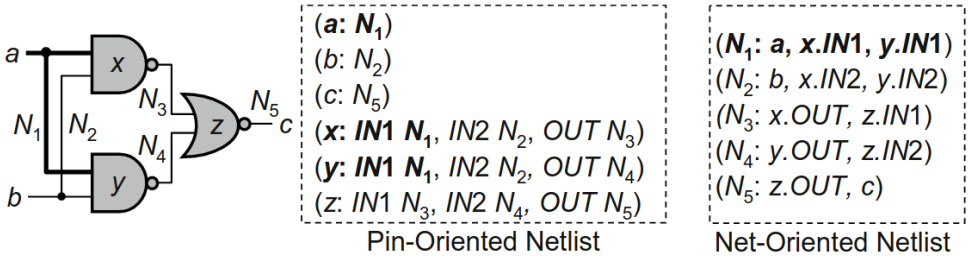


FIGURE 2.2: Pin-oriented (center) and net-oriented (right) netlist examples for the circuit (left)

Net Weight is a value representing the priority of a net. It is an essential concept in routing.

Euclidean Distance is the length of the line between P_1 and P_2 [Figure 2.3]. The formula for the Euclidean distance is

$$d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2)}$$

Manhattan Distance is the sum of the horizontal and vertical displacements between P_1 and P_2 [Figure 2.3]. This distance metric is used in VLSI routing algorithms. The formula for the Manhattan distance is

$$d_M(P_1, P_2) = |x_2 - x_1| + |y_2 - y_1|$$

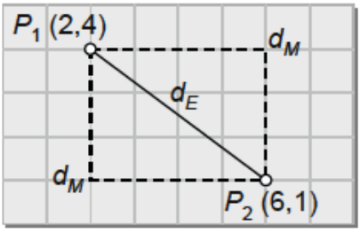


FIGURE 2.3: The Euclidean distance between P_1 and P_2 is 5, whereas the Manhattan distance between P_1 and P_2 is 7.

2.2.2 VLSI Design Flow

The VLSI circuit design process includes highly complex steps to finish the flow. It is an iterative cycle. At the final stages of the flow, right before fabrication, various tools and algorithms operate on detailed information regarding each circuit element's geometric shape and electrical properties. Each step of the flow has a dedicated EDA tool that will cover all aspects of the particular task perfectly.

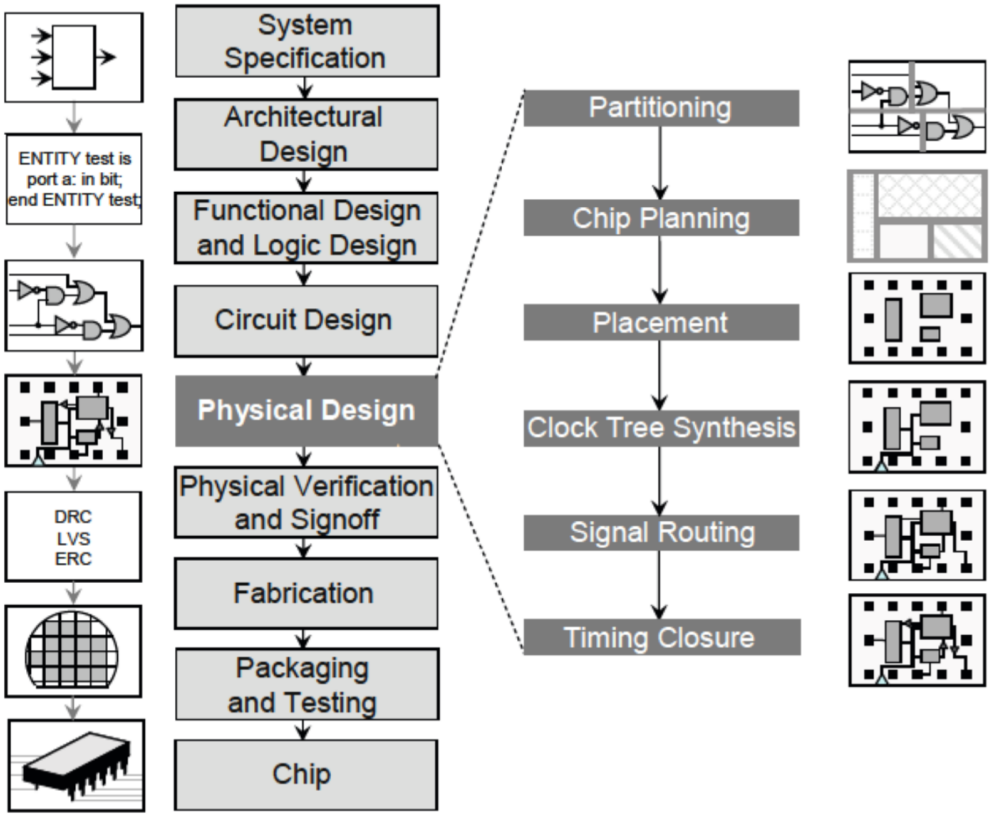


FIGURE 2.4: Major steps in VLSI design flow, with an emphasis on physical design steps

1. **System Specification:** The high-level specifications requirements of the system are defined at this step.

2. **Architectural Design:** Design engineers start working on the architecture design based on the system specification and requirements. Architectural design includes integrating analogue and mixed-signal blocks, power requirements, memory management, internal and external communication, and choice of process technology and layer stacks. [4]

3. **Functional and Logic Design:** After the architecture is set, the functionality and the high-level behaviour of each module in the design are defined. Logic

design is performed at the *register-transfer level* (RTL) using a *hardware description language* (HDL). The two most common HDLs are *Verilog* and *VHDL*. Logic synthesis is the next step, where the described functionality is mapped to a gate-level netlist and specific circuit elements using synthesis tools. Verification steps are performed here to ensure RTL design is done as planned, and the netlist is correctly generated. [5]

4. **Circuit Design:** The circuit is then designed based on the logic design generated previously. Boolean expressions are converted into a gate-level netlist. Circuit simulation is done in this step to verify the timing behaviour of each component of the system. SPICE is a common program tool for running circuit simulations.

5. **Physical Design:** The netlist is converted into a physical geometric representation. This complex step is divided into multiple key sub-steps:

   (a) **Partitioning:** Circuit is decomposed into multiple smaller sub-circuits or modules, which can be designed or analyzed individually.

   (b) **Floor Planning:** This step determines the shapes, orientations and arrangements of sub-circuits or modules and the positions of external ports and IP or macro blocks.

   (c) **Placement:** This step finds the spatial locations of all of the cells within each block, and integration of analogue blocks or external IP cores is performed.

   (d) **Clock Tree Synthesis:** builds the clock tree and meets the defined timing, area and power requirements.

   (e) **Routing:** This step connects all the components together

      i. **Detailed Routing:** assigns exact routes to specific metal layers and routing tracks within the defined global routing regions

      ii. **Global Routing:** constructs a routing plan for all the nets assigned to routing regions. Here, the objectives are to balance the congestion level across all routing regions and minimize the total wire lengths, making the detail routing process easier.

   (f) **Timing Closure:** Optimization of the circuit performance by using specialized placement and routing techniques.

6. **Physical Verification and Signoff:** Here, physical verification checks such as *Layout VS Schematic* (LVS) and *Design Rule Check* (DRC) are performed. DRC checks if the given layout satisfies the design rules provided by the fabrication team. LVS checks if the functionality and netlist of the layout and the schematic match or not.

7. **Fabrication:** Tapeout is reached after all the stages in the design and verification process have been completed. The design is now ready for fabrication. The final clean layout, formatted in *GDSII*, is then handed off to the silicon foundry. ICs are manufactured on round silicon wafers during the fabrication process, and many different photomasks are used to produce the ICs.

8. **Packaging and Testing:** Each of the silicon wafers contains hundreds of chips. After dicing up the wafer, the chips are packaged and sealed and finally tested to ensure the design requirements are still met.

2.2.3 Time and Space Complexities of Algorithms

Big O time complexity is the language and metric used to describe and gauge the efficiency of any algorithm. Simply put, it tells the rate of increase in time required by the algorithm to run as a function of some natural measure of problem size. Complexity is represented *asymptotically*, with respect to the size of input using the big O notation [6].

For instance, in the block placement process, the time to place n blocks can be denoted as $O(n)$. The more blocks, the more time is needed to run the algorithm. If the expression were instead $O(2^n + n + c)$ where c means constant time, the run-time complexity would be denoted as just $O(2^n)$ because 2^n is the fastest growing term.

Time is not the only important factor in an algorithm. The amount of memory or space used is also a crucial factor when assessing the efficiency of an algorithm. If the algorithm creates an array of size n, the space required will be denoted as $O(n)$; if the algorithm creates a $2d$ array of size $n * n$, this will require $O(n^2)$ space [7].

[Figure 2.5] depicts some typical big O rates of increase.

2.2.4 Graphs

In Computer Science, *graphs* are data structures representing connections of *nodes/vertices*. The relationship between two neighbouring nodes is called an *edge*. In short, a graph is an interrelationship of nodes connected by edges. A *path* between two nodes is an ordered sequence of edges from the starting node to the destination node. A *cycle/loop* is a closed path that starts and ends at the same node.

A graph $G(V, E)$ is comprised of two sets – a set of nodes/vertices (V) and a set of edges (E). A degree of a node/vertex is the number of its incident edges [8].

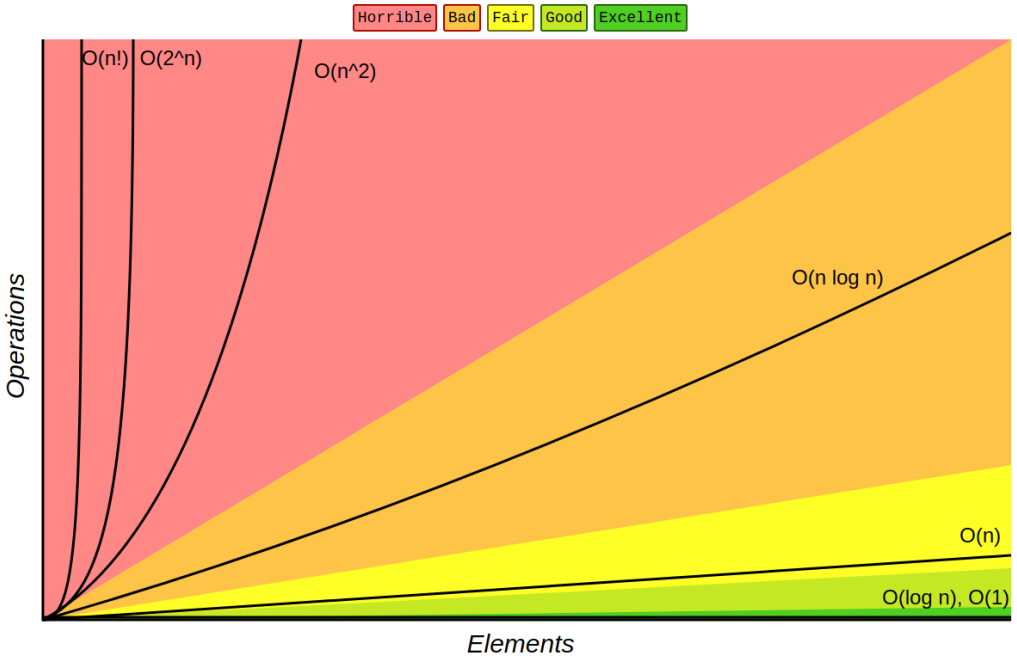
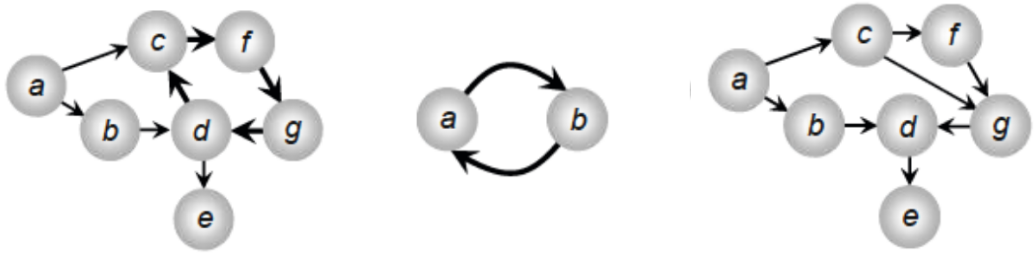


FIGURE 2.5: From this chart, $O(2^n)$ is faster growing than $O(n)$ and $O(1)$

A *weighted graph* is one in which the edges of the graph have a *cost*, known as *weight*. Depending on the use case, the weight can represent ideas like distance, time, or fees. On the other hand, an unweighted graph is simply a graph with a uniform weight across the entire graph.

Graphs can be divided into two categories: *undirected* and *directed*. An undirected graph is a graph in which the edge between any two nodes is bidirectional, while a directed graph is where the edge between two nodes has a single specific direction. A directed graph is *cyclic* if it has at least one directed loop. Otherwise, it is called a *directed acyclic graph (DAG)*. Refer to [figure 2.6] for some examples.

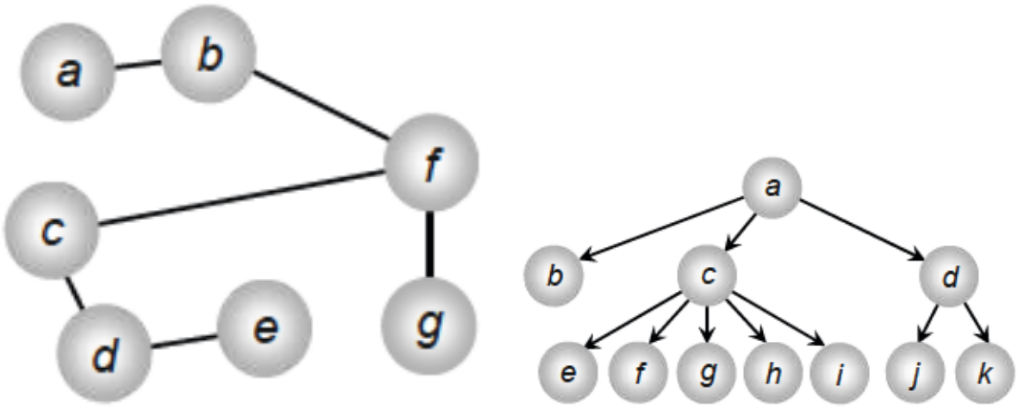


(a) (b) (c)

FIGURE 2.6: 2.6(a) A directed graph with one cycle *f-g-d-c-f*. 2.6(b) A directed graph with one cycle *a-b-a*. 2.6(c) A DAG, no cycle

A *tree* is another type of graph in which n nodes are connected by n - *1* edges. Since it is a graph, there are also undirected and directed trees. In an undirected tree, a node

with only one incident edge is called a *leaf*. In the undirected tree from [figure 2.7(a)], the leaf nodes are *a*, *e* and *g*. In a directed tree, there must exist a *root* node. The root has no incoming edges, only outgoing ones; a leaf in a directed tree is a node with no outgoing edge. In a directed tree, there must exist a unique path from the root to any other node in the tree [9]. In the directed tree from [figure 2.7(b)], the root node is *a*, and the leaf nodes are *b* and *e-k*.



(a) (b)

FIGURE 2.7: Undirected Graph and Directed Graph

Graphs are generally used to visualize layout topologies and solve real-world problems, physical design algorithms in our case. One example use case of DAG is in VLSI routing [9], where nodes represent the pins, and the edges represent the routed wires. A signal generated flows from an output pin of a gate to an input pin of another gate and not the other way around.

2.2.5 Spanning Trees and Steiner Trees

A *spanning tree* in a graph *G* is a connected acyclic sub-graph of *G* that is also a tree by itself and contains all the vertices of *G*. It consists of *n - 1* edges where *n* is the number of vertices in the graph *G*. Spanning trees play an essential role in designing efficient path routing algorithms such as the Steiner tree problem and the travelling salesperson problem [9, 10].

A *minimum spanning tree (MST)* of a weighted graph *G* is a spanning tree with the minimum sum of edge costs. In essence, a minimum spanning tree has two main properties: (1) it includes every vertex in the graph, and (2) the total cost of all the edges is the minimum. There are several well-known algorithms for solving the minimum spanning tree problem: Boruvka's algorithm, Prim's algorithm, and Kruskal's algorithm. [9]

A *rectilinear minimum spanning tree (RMST)* is an MST where the weight of the edge between each pair of vertices corresponds to the *Manhattan* (rectilinear) distance metric.

A *Steiner tree* is a tree of minimum weight that contains all vertices. In addition to the existing set of vertices, Steiner trees have *Steiner points* [2.8, 2.9(b)]. Edges in a Steiner tree can connect to Steiner points and the original vertices. The Steiner points reduce the total weight of the tree even more so than a Rectilinear Minimum Spanning Tree. If implemented in the Manhattan plane using only horizontal and vertical directions, then the Steiner Minimum Tree is called a *rectilinear Steiner minimum tree (RSMT)* [9].

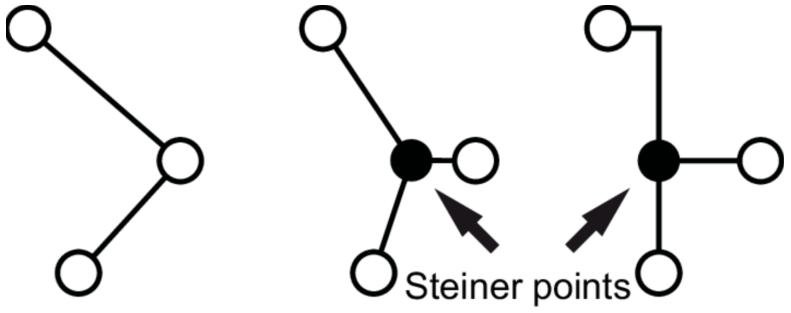
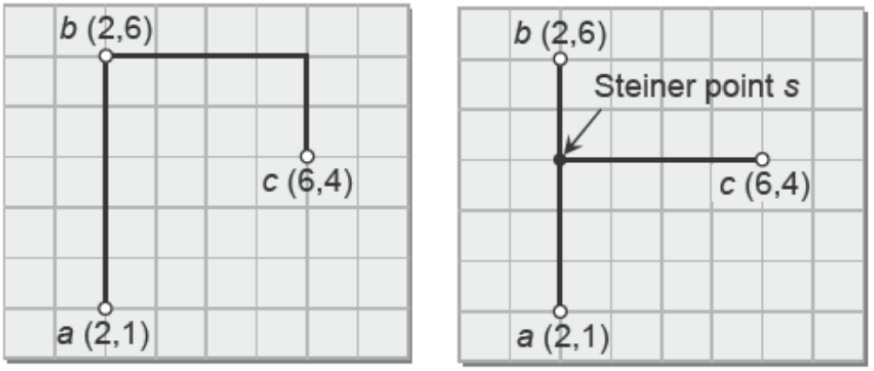


FIGURE 2.8: Steiner Points

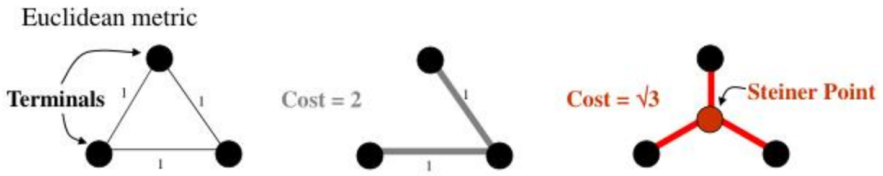


(a) rectilinear minimum spanning tree (b) rectilinear Steiner minimum tree

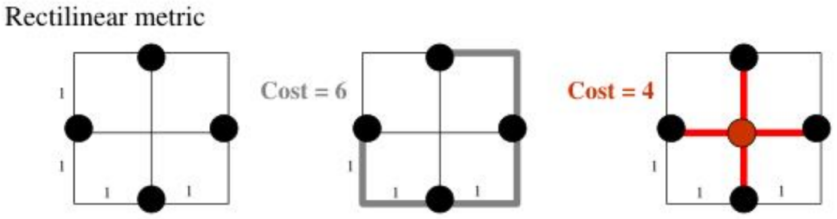
FIGURE 2.9: 2.9(a) Rectilinear minimum spanning tree (RMST) connects points a - c with the cost of 11. 2.9(b) Rectilinear minimum Steiner tree (RSMT) connects points a - c with the cost of only 9.

2.2.6 Rip-up and Reroute

Router algorithms that route one net at a time face a fundamental issue: they might produce conflict between two nets and cause the optimal path of early routes to block the optimal path of later routes. These failed interconnections typically add up to 5% of the total number, and manual routing was traditionally required [11]. So, this new rip-up and reroute idea exists to solve the issue. This novel idea is initially proposed by



(a) Euclidean

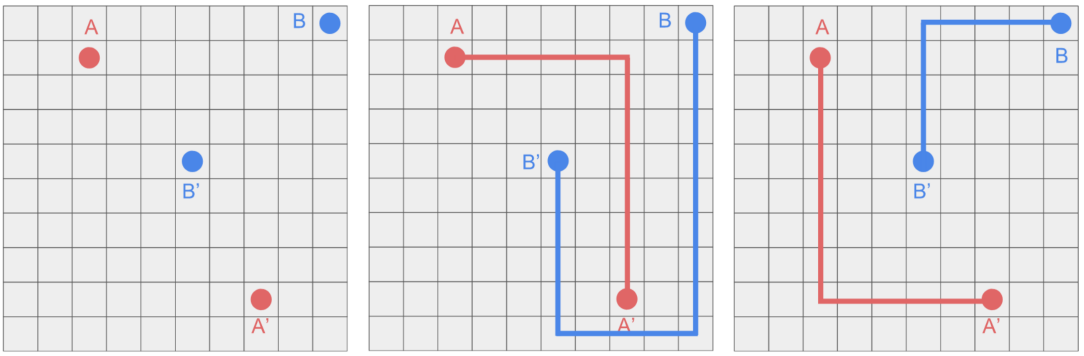


(b) Rectilinear

FIGURE 2.10: 2.10(a) Euclidean version of spanning tree (middle) and Steiner tree (left). 2.10(b) Rectilinear version of spanning tree (middle) and Steiner tree (left)

I. Shirakawa and Shin Futagami back in 1983 [12]. It aims to route previously impossible connections as well as to reduce unnecessary route detours [13]. The basic idea of this technique is to handle and rearrange the routing order so that the total wire length is minimal.

In some cases, switching the order of net routing can produce a more optimal solution. Figure 2.11 demonstrates the problem of unnecessary route detours. In subfigure 2.11(b), when net A is routed first, it will cause net B to be routed less optimally. By changing the order and routing net B first, the router will find the optimal solution shown in subfigure 2.11(c).



(a) Two nets A and B, both with two pins each, namely A and A' and B and B' respectively.

(b) Possible routing solution for nets A and B, routing net A first. Note that net B has sub-optimal wire length.

(c) Another possible routing solution for nets A and B, routing net B first. Note that now both nets have optimal wire length.

FIGURE 2.11: Net Ordering example to get optimal wire length.

2.3 A Review of Routing Algorithms

2.3.1 Lee Maze Router Algorithm

The Lee Maze Router is one of the earliest algorithms for solving path connection or maze problems, proposed by a computer scientist named C. Y. Lee in 1961 [14]. The algorithm uses *Breadth-First-Search (BFS)* technique to find all paths from source to destination in path-length order [13, 15, 16]. This algorithm guarantees to compute the path of minimum length even though it is not efficient [17]. It was first created to route wires on a single layer, and then later found that it is also very straightforward to route on multiple layers. The general idea of how it works can be summarized into three steps [11]:

1. **Wave propagation phase** - The BFS step, keep expanding the search until the target cell is reached
2. **Backtrace phase** - To backtrack all the way to the source cell, to get the full path in order
3. **Label clearance phase** - To clear all the previously marked cells during propagation phase

In 1978, J. Soukup proposed an improved version of the maze router [18]. He named it *Fast Maze Router*, instead of only relying on BFS [16], when a node expands and finds a neighbouring node that lies close to the target than itself, it does a *Depth-First-Search (DFS)* in that neighbour node's direction and adds all newly discovered nodes to the wavefront collection. So, this implementation combines both BFS and DFS approaches, leading to better and faster performance.

In a nutshell, the original Lee's maze router algorithm was a simple and effective method for solving mazes. It was not as efficient as more modern algorithms, such as A* or Dijkstra's algorithm. Still, it was a useful starting point for developing more sophisticated maze-solving techniques.

2.3.2 Dijkstra's Algorithm

Dijkstra's Algorithm [19] is a popular algorithm for finding the shortest path between two nodes in a graph. It was first described by Dutch computer scientist - Edsger Dijkstra in 1959. The algorithm works by starting at the source node and exploring the neighbouring nodes in increasing the order of distance from the source. At each step, the

algorithm updates the distances to the adjacent nodes and adds the nearest unvisited node to the list of visited nodes. This process continues until the destination node is reached or until there are no more unvisited nodes. This is often used in routing and as a subroutine in other graph algorithms.

The brief step-by-step guide of the algorithm is as follows [20]:

1. Mark all vertices as unvisited.
2. Set the current distance of the starting vertex with 0 and the other vertices with infinity.
3. Set the starting vertex as the current vertex.
4. For each of the unvisited neighbours of the current vertex, calculate their distances by summing up the current distance of the current vertex and the weight of the edge that connects to the neighbour.
5. Compare the newly calculated distance with the current distance assigned to the neighbour vertex. If the new distance is less than the current distance, replace the current distance with the newly calculated distance.
6. Mark the current vertex as visited after analysing all the neighbouring vertices.
7. End the algorithm program if the target vertex is marked as visited.
8. If not, choose the unvisited vertex with the least distance, set it as the new current vertex and repeat the process from step 4.

One of the key features of Dijkstra's algorithm is its use of a priority queue to store the distances to the neighbouring nodes. This allows the algorithm to quickly find the next unvisited node with the smallest distance, which makes it much more efficient than other algorithms that use a simple queue or list to store the distances.

The time complexity of Dijkstra's algorithm is $O((V + E)logV)$, where V is the number of nodes in the graph and E is the number of edges. This means that the algorithm's running time is proportional to the number of nodes and edges in the graph, and it also depends on the size of the priority queue used to store the distances to the neighbouring nodes.

The space complexity of Dijkstra's algorithm is $O(V)$, which means that the algorithm uses a fixed amount of space that is independent of the size of the input graph. This is because the algorithm only stores the distances to the neighbouring and visited nodes, and it does not require any additional data structures to keep track of the entire graph.

Overall, Dijkstra's algorithm is considered relatively efficient for solving shortest-path problems, especially compared to other algorithms with a higher time and space complexity.

2.3.3 A* Search Algorithm

A* is a popular algorithm used in pathfinding and graph traversal. It is an informed search algorithm, meaning that it considers the specific goals of the search in addition to the data in the graph. This makes it more efficient than other algorithms, such as breadth-first search. The A* algorithm uses a heuristic function to guide the search process. This function considers the distance between the current node and the goal node and any other relevant information, such as the terrain or obstacles in the environment. The algorithm then uses this information to prioritize the paths that are most likely to lead to the goal [21].

At each step of the search process, the A* algorithm selects the path with the lowest heuristic value and expands it. This means that it considers all possible paths that can be reached from the current node and then calculates the heuristic value for each of those paths. The path with the lowest heuristic value is then selected, and the process is repeated until the goal is reached.

One of the key advantages of the A* algorithm is its ability to find the optimal path. This is because it uses the heuristic function to guide the search, ensuring that the algorithm always considers the paths most likely to lead to the goal. In contrast, other algorithms, such as Dijkstra's algorithm, can sometimes get stuck in suboptimal paths, leading to longer search times and less efficient solutions.

In 1984, a computer scientist named G. W. Clow applied the A* algorithm to the problem of VLSI routing [22]. CLOW used the A* algorithm to find the optimal paths for routing the connections on the chip. This involved defining a heuristic function that considered factors such as the distance between the components and obstacles on the chip. The algorithm then used this information to guide the search for the optimal routing paths.

CLOW's work demonstrated the effectiveness of the A* algorithm for VLSI routing. It showed that the algorithm could find the optimal paths quickly and efficiently, making it a valuable tool for improving the performance of VLSI circuits.

2.4 Modern Global Router Examples

This section will cover the different implementations of modern global routers. The following examples are participants and winners of *International Symposium on Physical Design (ISPD)*. The ISPD provides a premier forum to promote new research ideas in all aspects of physical design. It hosted Global Routing contests in 2007 and 2008, open to all research groups.

2.4.1 BoxRouter 2.0

Minsik Cho from the University of Texas launched an improved version of BoxRouter global router, known as BoxRouter2.0 [23]. This updated version offers many enhancements, making it the global router for ultimate routability. In ISPD07 Global Routing Contest, BoxRouter won 2nd place in 3D Routing Contest and 3rd place in 2D Routing Contest. BoxRouter2.0 has two critical steps: (1) 2D global routing and (2) layer assignment. The main highlight features are as follows:

- Implementation of dynamic scaling for robust negotiation-based A* search. It stops the router from spinning out of control by balancing past costs and present congestion costs, ensuring consistent routability improvement over iterations.
- Performing topology-aware rip-up method which only removes some of the wires in the congested areas without changing the net topology.
- Proposition of *integer linear programming (ILP)* for via/blockage-aware layer assignment to reduce overall runtime and handle blockages.

2.4.2 MaizeRouter

Michael D. Moffitt from the University of Michigan submitted his MaizeRouter implementation to ISPD Global Routing Competition in 2007 [24]. Michael's MaizeRouter won 1st place in 3D Routing Contest and 2nd place in 2D Routing Contest. This is impressive as it is a massive leap in progress compared to publicly-available routing tools because it doesn't use popular algorithms such as ILP formulations, multicommodity flow-based techniques, and congestion-driven Steiner Tree generation. The foundation of the MaizeRouter algorithm is built instead with edge-based operations, consisting of:

- Extreme edge shifting, which is a technique targeted to reduce routing congestion efficiently.

- Edge retraction, which is the counterpart to extreme edge shifting to reduce wirelength.

2.4.3 NTHU-Route 2.0

The National Tsing Hua University (NTHU) launched a new version of its popular VLSI routing tool, known as NTHU-Route 2.0 [25]. This updated version offers several significant improvements over the previous state-of-the-art router: NTHU-Route, making it an even more powerful and effective tool for designing VLSI circuits. This router won 1st place in the ISPD 2008 Global Routing Contest. It solved seven out of 8 ISPD07 and 12 out of 16 ISPD08 benchmarks with no overflow. The main features are as follows:

- New history based cost function, which is divided into 3 sub-cost functions, which are: (1) base cost function, (2) congestion cost function, (3) via cost function.
- New ordering methods for congested regions identification, and for rip-up and reroute
- Two techniques for runtime reduction . The first being computing and storing the sum of the base cost and congestion cost for every single edge and update them when needed rather than re-computing the cost on the fly when needed, this will significantly improve the performance. The second is to change the implementation of the edge's look-up table from a balanced search tree to a hash table, this greatly improves efficiency and use less memory consumption.

2.4.4 CS6135 Homework 5: Global Routing

On the academic side, this is an assignment sheet from *National Tsing Hua University (NTHU)* CS6135 VLSI Physical Design Automation course. This course is designed by Ting-Chi Wang, one of the authors of NTHU-Route 2.0 (2.4.3)[17]. The assignment is similar to the Global Routing Contests hosted by ISPD in 2007 and 2008. It is about creating a global router algorithm. The basic premise of the assignment is as such: Given some 2-pin netlist input files with a specific format, generate a global router to route the input netlist and dump the output with a specific format. The two objectives are to minimize the total overflow and wire length, with two constraints being the horizontal and vertical channel capacity.

Lei Hsiung was a student in this course at NTHU. According to his CV on his personal website (https://hsiung.cc/), he ranked top five performance in placement and routing projects. An assignment report is shared in a publicly available GitHub repository

[26]. In the report, he gave an overview of the strategy for designing the global router algorithm and compared the performance with other peers from previous years.

In this final year project, we will be implementing a global router algorithm in Python using Lei's strategy, as outlined in Figure 2.12. The fundamental strategy is as follows:

1. Parse the input netlist file, and store the data in some data structures.
2. For each of the 2-pin net from the netlist, use BFS to find the path between the 2 pins.
3. If overflow exists:
   (a) If we have reached the max number of rip-up and reroute attempts:
      i. Jump to (4a)
   (b) Else if we have not reached the max number of rip-up and reroute attempts:
      i. Perform rip-up and reroute.
      ii. Retrieve all the nets that require rerouting.
      iii. Jump to (2)
4. Else if overflow does not exists:
   (a) Dump the result of the Global Route output.
   (b) Terminate program.

In our project implementation, we will write the code in Python as opposed to C++ in his implementation. Some of the reasons include that we will be using Python Flask to host a server and Python Matplotlib to plot visualizations later. Python is also my preferred language. So, implementing the algorithm with Python is more appropriate here. At step 2, we can use any path-finding algorithm to route the two pins, which means we can use Dijkstra's Algorithm or A* algorithm for this project.

For our project, there will be a lot more components implemented. For more information regarding the project, all its components and architecture can be found in Chapter 4.

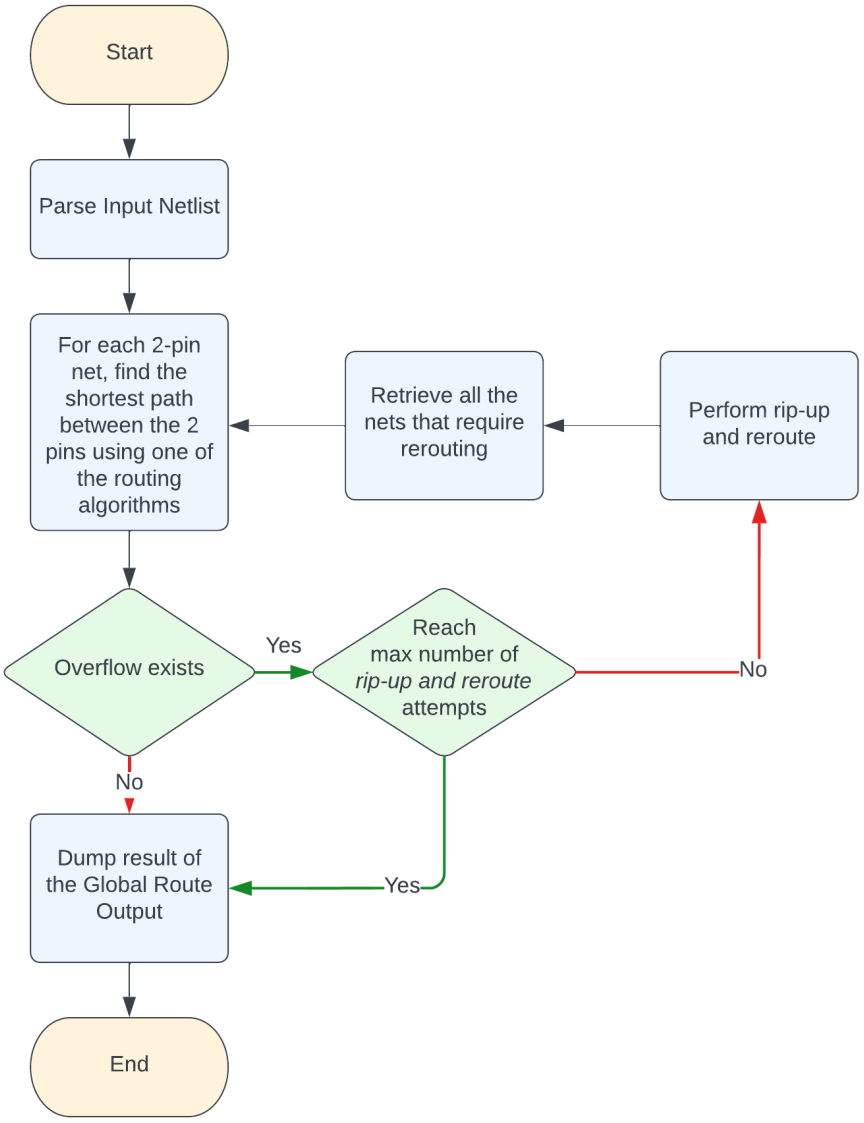


FIGURE 2.12: General flow of the Global Router Program

Chapter 3

Algorithmic Techniques for Minimizing Congestion in VLSI Global Routing

3.1 Problem Definition

In VLSI, after the location of each cell of an IC design is determined at the placement stage, the cells need to be connected properly during the routing stage. The routing stage is categorized into two steps: (1) Global Routing and (2) Detailed Routing. During global routing, nets are connected on a coarse grid graph with horizontal and vertical track capacity constraints. Next, detailed routing follows the routing region solution defined in global routing to find the exact routing solution. The quality of the global routing solution will affect timing, power and density in the chip area. So, global routing is a very critical stage in the chip design flow.

In most modern IC designs, engineers work with a large number of nets in an extremely limited area. Typically, this large amount of nets requires tens of millions of wires, which translates to kilometres of wires on the tiny surface of the chip. For example, Qualcomm's Snapdragon 8 Gen 2 technology node is just under 4 nanometers [27]. It is just physically impossible to manually route kilometres of wires in these nanometer-scale chips. The components, such as standard cells and macros, are typically packed together densely without any free extra space, so precision and accuracy are crucial here. That is why we need to create clever algorithms and use software tools to do this task accurately and efficiently.

Solving the global routing problem accurately is difficult and complex. We must first think of an appropriate data structure to represent the geometry structure of the routing regions of the layout most efficiently. Then we will need to define the scale and properties of the routing regions for the layout and assign each net in the netlist to a routing region.

Next, in every given routing region, there is a maximum horizontal and vertical capacity for the number of wires that can be placed there. These upper-bound numbers are not to be oversubscribed. Otherwise, it will lead to overflow, meaning more wires than the track capacity allows. With this constraint, it will force some routes to detour and, thus, make the estimation of the final total length of wire segments extremely difficult. Furthermore, we must figure out how to apply the routing algorithm to multiple metal layers, as most chips in the industry have tens of metal layers connected with vias.

Not only that, we need to think about how to route not just two-point nets but multi-point nets because, in most cases, a net will have more than two pins in the real-world industry. Also, for real-world industrial routers, the change in the direction of the wires is more expensive than just routing it straight. Bends are generally bad and inefficient in routing; we only want to bend the wires if we have to. We would always prefer the wires to be routed straight rather than zigzag.

Since bigger chips have an enormous number of nets, not every net gets to take a straightforward path to connect as there are some constraints to be satisfied, such as the horizontal and vertical track capacities, via costs, bend penalty costs, and many more. Another critical issue is that the early nets routed may block the path of later nets, and in many cases, the optimal route choice for one net may block the route for another.

3.2 Objectives

To represent the routing regions of the layout in this project, we will use a *grid graph*, defined as $G = (V, E)$, where vertices $v \in V$ represent the *Global Bins* and the edges represent the connections of any two adjacent global bins.

Next, we can represent the number of available routing tracks e as $supply(e)$, which is the supply of the edge. Furthermore, the number of nets that pass through e can be represented as $demand(e)$, which is the demand of the edge. So, the overflow on an edge e can be defined as the difference between the demand and supply as shown below:

$$overflow(e) = \begin{cases} demand(e) - supply(e) & : if\ demand(e) > supply(e) \\ 0 & : \ demand(e) \leq supply(e) \end{cases}$$

There are two main optimization objectives of global routing algorithms. The primary objective is to minimize the total overflow on all edges. This can be represented in the formula below:

$$minimum\left(\sum_{\forall e \in E} overflow(e)\right)$$

The secondary objective is to minimize the total wire length used on all edges of all nets, this can be represented in the formula below:

$$minimum\left(\sum_{\forall n_i \in N}\sum_{\forall e \in E_i} length(e)\right)$$

In this project, we will make the following simplifications to our Global Router grid-graph model:

- Only taking two metal layers into account (vertical and horizontal).
- Only considering 2 pins per net.
- Only considering fixed wire width and padding, which is 1 unit.
- Asserting that $length(e)$ is always 1 unit.
- Nets are already mapped to a routing region.

3.3 Functional Requirements

3.3.1 LEF & DEF Files Parsing

The system will parse both LEF and DEF files to retrieve all the coordinates of all the pin positions, the vertical and horizontal capacity constraints, and determine the grid size of the layout. Then, it will generate a modified version of the netlist in a format mentioned above.

3.3.2 Modified Netlist Input Validation and Parsing

The modified input netlist file will be validated and parsed by the system before the routing algorithm starts processing it. If the input file here does not follow the specified format, it should notify the user with an appropriate error message.

3.3.3 Routing Algorithm Selection and the Routing Process

The system should be able to provide the user different algorithm options to route the nets. Once an algorithm is selected, the system will start the routing process. All nets will be routed with two main objectives: (1) Minimum Congestion and (2) Minimum Wire Length. One thing to note is that a different algorithm will produce a different route solution and result.

3.3.4 Generation of Congestion Level Information based on Routed Output

The system should validate and parse the system processed and generated output file that contains the information of all routed paths. This is so that it can create a new file that contains information about the congestion level for each grid cell.

3.3.5 Storage for Routed Solution Outputs

Once the routing process is done, they system should give the user the option to save and store the routed solution output file in the system. The routing process could be long and the user might not want to wait for the routing process of the same layout every time, so the system should be able to store the routed output internally securely. Whenever the user wants to check on the routed output or the congestion map again, they can easily do so by selecting the routed output file they previously generated.

3.3.6 Analysis on the Routed Output Solution

After the routing process is done for a netlist, the system should provide the user a comprehensive analysis on the output result. The three main metrics to display for the user are:

- Total Wire Length: The total length of wire used to route the entire netlist
- Process Runtime: The time taken to run the entire routing process
- Number of Overflows: The total number of grid cells that has overflow

3.3.7 Display the routed paths for the netlist

The system should provide a visual representation of the routed output like the example above 4.6. Moreover, it should provide the user some basic interaction functionality such as zooming in and out of the grid to get a better understanding of how the the global routes are laid out for all components on the layout.

3.3.8 Display a congestion heat map for a netlist

The system should provide the functionality to display a congestion heat map with appropriate labels and legends to help to user visualize the congestion levels of the entire layout. It should create the congestion map based on the generated previously congestion level info file.

3.4 Non-Functional Requirements

3.4.1 Secure Netlist Input File Upload

Since this is the only input from the user into the system, file input validation has to be done correctly to make sure the system is not vulnerable to malicious attacks such as server-side scripting, XSS, CSRF, and arbitrary code execution attacks. Proper security checks will be in place to mitigate prevent these attacks.

3.4.2 Design Layout Representation with Graph Data Structure

The design layout will be represented as a graph data structure during the routing process. The routing algorithms will treat the grid cells as vertices, the pins as source and destination vertices, and the boundaries between any two adjacent grid cell as edges.

3.4.3 Web Application Responsiveness

The web application will be designed in such a way that it will automatically adjust for different screen sizes and viewports. This will allow the application to look good on desktops, tablets, and/or phones.

3.4.4 Web Application UI/UX

The web application will provide a user friendly and intuitive *User Interface/ User Experience (UI/UX).*

3.4.5 Web application Reliability

The web application will be hosted online and will be up online 24/7. It should be reliable and not face any down time issues.

3.4.6 Secure Storage of Routed Output Files

Since users can store their routed solutions on the system internally, there should be a strict security measure in place to keep their files and data safe and secure.

Chapter 4

Implementation Approach

This chapter will go through the high-level architecture overview of the system and the implementation approach. It will also cover the details of various software technologies that will be involved in this project. Furthermore, a detailed implementation and evaluation plan schedule and methodology are also discussed in the chapter. Lastly, a prototype discussion will also be included in this chapter.

4.1 Architecture

The diagram figure 4.1 displays a high-level overview of how the system is constructed and its data flow. The authentication layer is the first gateway into the system, so users must authenticate themselves before using the web app. Here, we will use Google's Firebase Authentication to handle all authentication scenarios, such as login, account registration, and resetting password. More details on Firebase Authentication later in the subsection 4.1.2.3.

Once the user has authenticated themselves, they will have access to the system. The back end of the system is a Python Flask application. The reasoning behind choosing Flask as the back-end framework is described in subsection 4.1.2.1. Flask handles all the API function components, which include:

- Parsing and validating the netlist input file
- Route the netlist with using Dijkstra's Path Finding Algorithm
- Route the netlist with using A Start Path Finding Algorithm
- Generate output netlist file

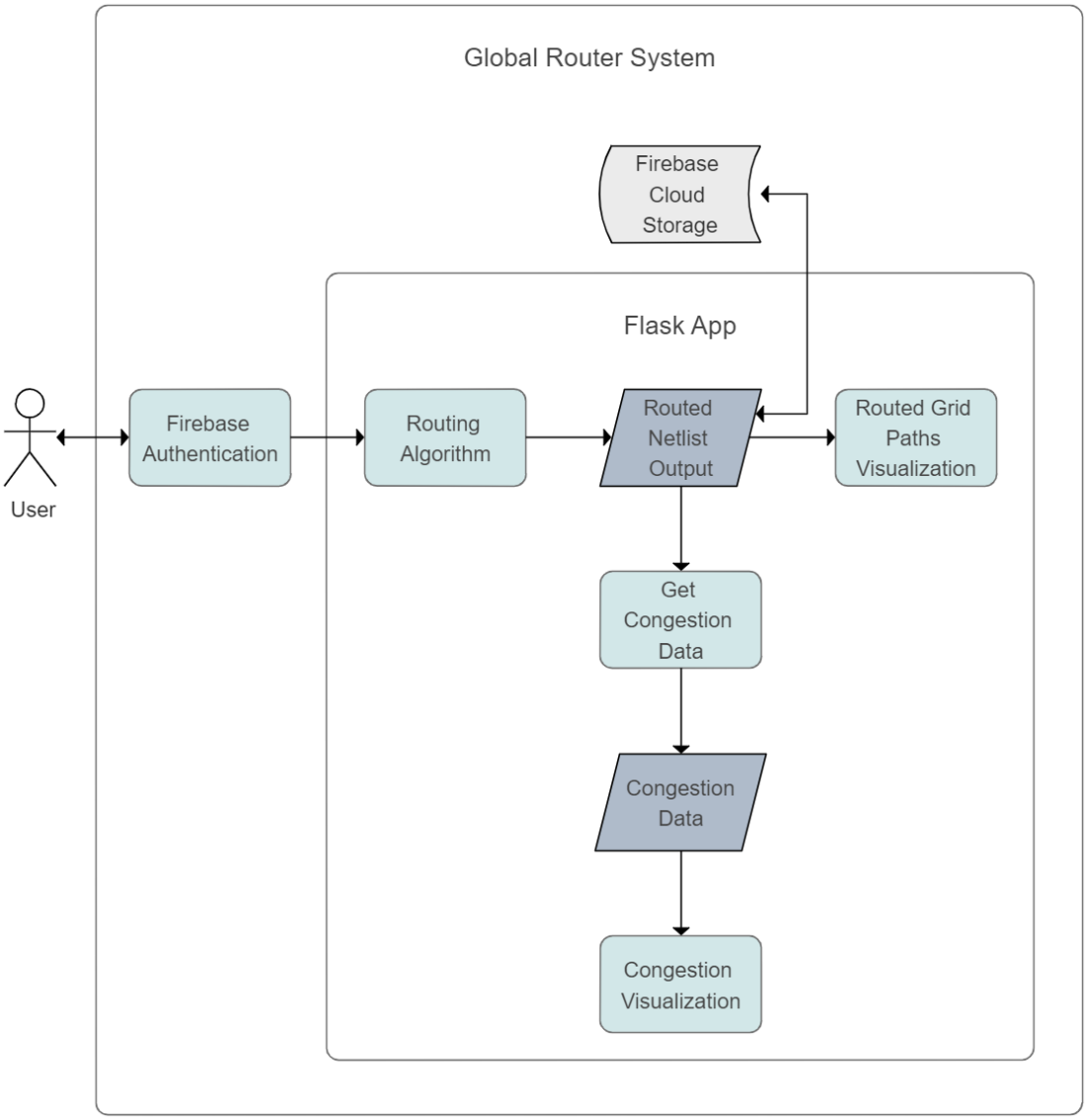


FIGURE 4.1: System Architecture

- Saving/retrieving routed output file to/from Firebase Cloud Store
- Generating congestion level heat map visualization
- Generating routed grid paths visualization

As for the application's front end, we will use Jinja2 Template Engine from Flask to render the HTML pages. To keep it minimal and neat, we will use Bootstrap5 library to handle all the styling and behaviour of the UI components. More details on Bootstrap here in the subsection 4.1.2.5

4.1.1 Global Router Model

Diving deeper into the system's global router model, the implementation strategy will be similar to Lei Hsiung's approach in [26]. The overall program flow has been discussed in this section 2.4.4. This section will define the appropriate data structures to represent the components in the global router model. Figure 4.2 shows an overview of the classes, methods, and their relationships to visualize what the system is made up of.

4.1.1.1 Router Class

Firstly, the *Router* class will store the general information about the router, which may include:

- List of nets
- Overall demand level on the grid
- The total overflow in the grid
- The total wire length used in the grid

Some Router methods might include:

- Parsing the input netlist file
- Finding a path between two points, here we can use any path finding algorithm to route. We will focus on using Dijkstra's Algorithm and A* Algorithm
- Rip-up a net and reroute it if overflow exists
- Check if there is any overflow in the grid graph
- Generate output of congestion data
- Get Set methods of the attributes mentioned above

4.1.1.2 Net Class

A Router class stores a netlist, which is a list of Net objects. A Net class is needed to store the information about a net, which may include:

- Name of the net

- ID of the net
- Number of pins on the net
- A list of all pins on the net
- The path solution to connect all the pins

Some Net methods might include the basic Get Set methods of the attributes mentioned above.

4.1.1.3 Path Class

Since a Net object stores its path solution, a Path data structure is required to store the following data:

- List of GBox coordinates that represent the path
- Total cost of the path, or known as the wire length of the path
- The end node object

A Net object should include methods to retrieve the coordinates list and total cost.

4.1.1.4 Node Class

Finally, a linked list data structure is needed to keep track of the Path, this is so that we can do the backtracking node by node and get the full path. This is what the Node class is for, it will encapsulate the following information:

- The previous node pointer
- This cost to route
- The xy coordinates of the current node

A Node object should also include basic methods such as the Get Set methods.

4.1.2 Technology Stack Overview

In this section, we will go over a brief overview on the various technologies and library frameworks we will use to build this project's web application. The reasoning behind these choices will also be thoroughly discussed.

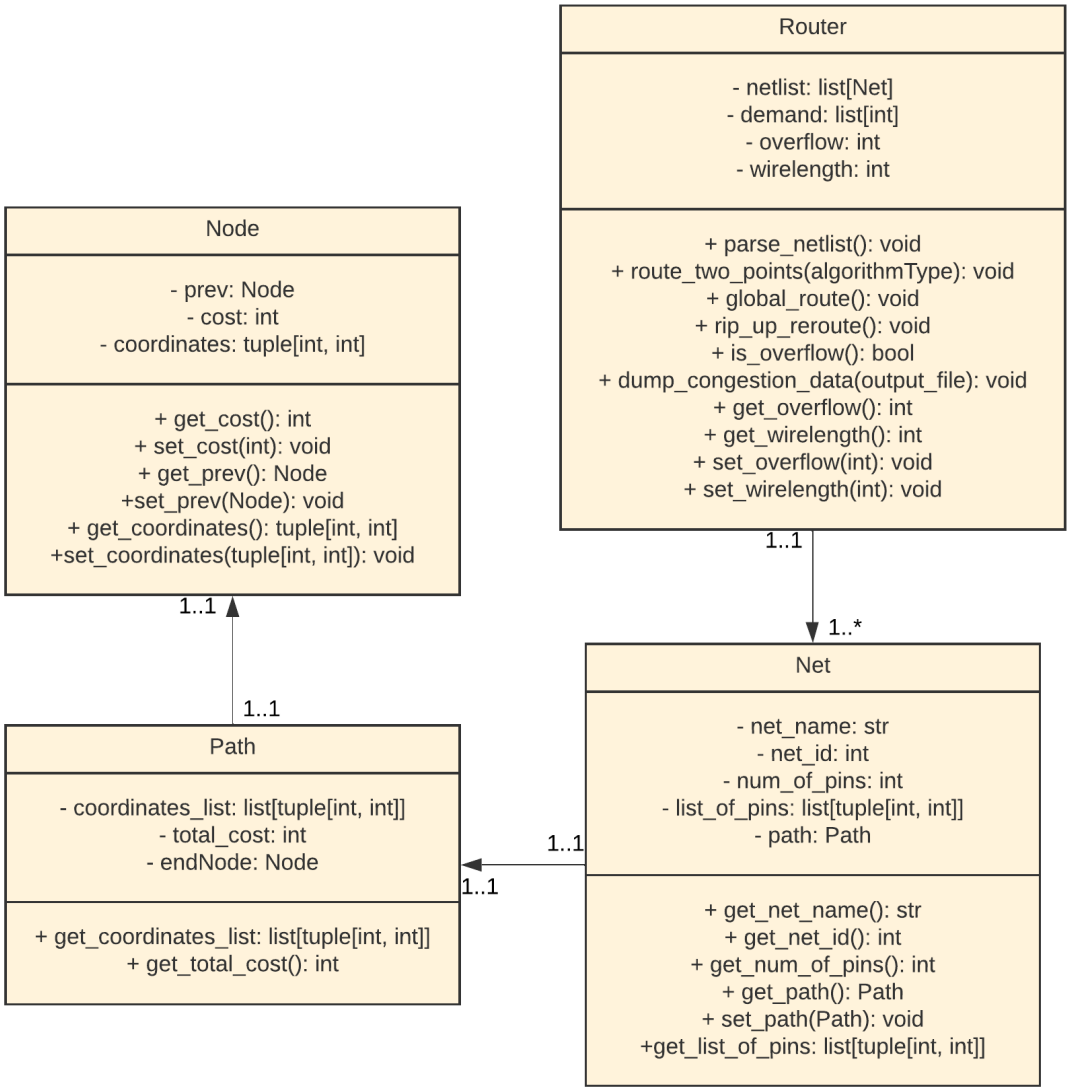


FIGURE 4.2: System Class Diagram

4.1.2.1 Python Flask

Flask is a lightweight *Web Server Gateway Interface (WSGI)* web application framework written in Python. It is considered the best and the most popular web application framework for serving light-weight web apps [28]. Some of the main advantages of using Python Flask include:

- It is a lightweight and modular design.
- Contains a built-in development server and a fast debugger.
- RESTful request dispatching.
- Jinja2 Template Engine usage, which allows passing of Python variables into HTML templates.

- Provides support for secure cookies.

Jinja2 is a powerful and flexible template engine that allows developers to create complex and dynamic templates for their Flask applications. It supports features such as template inheritance, filters, and macros, which make it possible to create sophisticated and reusable templates.

Flask is perfect for our application because the system architecture is straightforward. It will handle all the function APIs, the endpoint routing logic, and the Firebase connection to enable authentication and storage. It will also use Jinja2 as the template engine to render dynamic HTML pages.

4.1.2.2 Python Matplotlib

Matplotlib is a free and open-source library for creating static, animated, and interactive visualizations in Python. It is by far the most commonly used library to use when it comes to plotting graphs and charts using Python. Figures in Matplotlib can be made interactive, meaning that zooming and panning are possible [29].

There are several reasons why Python matplotlib is an excellent choice for data visualization in this project:

- **It is powerful and flexible:** Python matplotlib offers a wide range of plotting and charting capabilities, including support for different chart types, customizable markers and lines, and various color schemes. This makes it possible to create a wide range of different visualizations to suit your specific needs.
- **It is easy to use:** Python matplotlib has a simple, intuitive API that makes it easy to create charts and plots. It also includes many convenience functions and built-in themes, which can save you time and effort when creating visualizations.
- **It is well-documented:** Python matplotlib has extensive documentation, including tutorials, examples, and API reference materials. This makes it easy to learn how to use the library and find the information you need when working on a specific visualization.
- **It is widely used and supported:** Python matplotlib is a widely used and well-established library, with a large and active community of users and developers. This means that you can find a wealth of resources and support online if you need help with a specific problem or have questions about how to use the library.

Another interesting feature is that plots generated by Matplotlib can be integrated into Python Flask. So, we will be using this library to create the layer congestion heat map and any other plots then display them through our web application.

4.1.2.3 Firebase Authentication

Firebase Authentication is a cloud-based service provided by Google's Firebase platform that makes it easy to authenticate users in a mobile or web application. It supports a variety of authentication methods, including email and password, phone number, and popular third-party providers such as Google, Facebook, and Twitter [30]. We will use this authentication solution to securely save user data in the cloud and provide a personalized experience across all of the user's devices. This means that the user will have to create an account with credentials to use the application. Once logged in, they can upload their input netlist file to start the routing process and then save the routed output file to the system. All of the user's data and files are stored securely.

4.1.2.4 Firebase Cloud Storage

Firebase Cloud Storage is a cloud-based storage service provided by Google's Firebase platform. It is used for storing and managing user-generated data, such as images, audio files, and video files[31]. It is particularly well-suited for applications that require scalable, secure storage and seamless integration with other Firebase services, such as Firebase Authentication. We will use this platform to store the user's routed output files. Once the system generates the routed output file, the user will have the option to store the output file on the Cloud Storage, so they don't need to re-run the algorithm again for the same input netlist. The user can view all their stored output files at a glance and choose any output file to see the routed grid paths visualization and its corresponding congestion heat map.

4.1.2.5 Bootstrap 5

Bootstrap is an open-source front-end development framework. Its primary purpose is to help build and design websites quickly and easily. It is the most commonly used framework to build clean and responsive websites or web applications that look great and consistent on all sorts of screen resolutions. Bootstrap provides web developers with various UI components, such as buttons, tables, containers, navigation bars, etc. It saves the developers' precious time and effort from writing a lot of custom *Cascading Styling Sheet (CSS)* and JavaScript code and allows us to spend more time focusing

on the actual features of the application [32]. We will use this front-end development framework for most of our web application.

4.1.2.6 Github

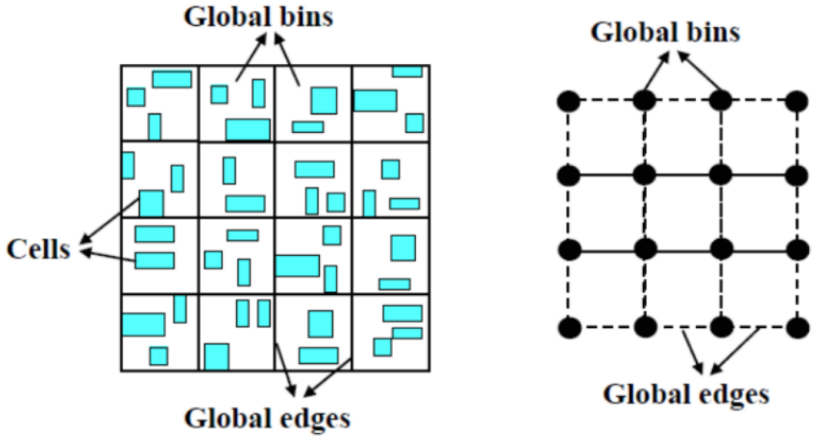
GitHub is a popular platform for hosting and managing software projects. It is widely used by developers to collaborate on software development projects, share code and other resources, and track and manage the development process. We will be using GitHub to host our source code, and the reasons are as follows:

- **It is a popular platform:** GitHub is a widely used platform, with a large and active community of users and developers. This means that there are many resources and support available if there is any specific problems or questions about how to use the platform.
- **It offers version control:** GitHub provides built-in support for version control using Git, which is a popular version control system we will use for the project's source code. This allows us to track changes to your code and resources over time, and to easily collaborate with others on the development of the project.
- **It provides tools for collaboration:** GitHub offers a variety of tools and features that make it easy to collaborate with others on a project. This includes support for code review, project management, and team communication. We will also be using Kanban based project board provided by GitHub to keep track of all progress of action items.
- **It allows us to share the project with others:** GitHub makes it easy to share our project with others, either publicly or privately.

4.1.3 Input Description

- The chip layout is partitioned into multiple rectangular grids called global bins, also known as G-cells. We can assume that a pin of a cell lies at the center of the grid if the grid contains the cell [4.3(a)].
- We can use a graph data structure to represent the partitioned layout. Each vertex represents a global bin, and each edge represents to a boundary between two adjacent global bins [4.3(b)].
- Vertical and horizontal track capacities of grid boundaries.

- A netlist, which is a set of nets to be routed on the layout, the grid graph. Each net has a set of pins to be connected.



(a) Partitioned Layout

(b) Graph Representation of the Partitioned Layout

FIGURE 4.3: Graph Representation Visual

4.1.4 Input Format

The input file must follow a specific format so the system can parse it accordingly. The format will be a simplified version of the one used in the ISPD 2007 and 2008 Global Routing Competitions. The first line of the file will specify the size of the grid($row \times column$). The second and third lines determine the vertical and horizontal capacities, respectively. The fourth line specifies the number of nets in the netlist. The information for all nets starts from the fifth line and beyond. Each net is described as follows: The first line states the net name, net ID, and the number of pins in the net, and then the following lines will be the pins' (x, y) coordinates that need to be routed and connected. Refer to 4.4 for an example of an input format.

4.1.5 Output Description

After processing and routing the netlist, the system will generate an output file. The output file holds the information on how each net is routed. The routed path of each net will be represented in the form of a list of all coordinate points connecting all pins of the net.

```
grid 3 3 # grid <numOfHorizontalGrids> <numOfVerticalGrids>
vertical capacity 2 # vertical capacity <verticalCapacity>
horizontal capacity 2 # horizontal capacity <horizontalCapacity>
num net 3 # num net <numOfNets>
net0 0 2 # <netName> <netID> <numOfPins>
  0 1 # <pin0 x-coordinate> <pin0 y-coordinate>
  1 1 # <pin1 x-coordinate> <pin1 y-coordinate>
net1 1 2
  0 2
  1 1
net2 2 2
  2 2
  1 0
```

FIGURE 4.4: Input format example

4.1.6 Output Format

The generated output file has a specific human-readable format. The format will be the same as the one used in ISPD 2007 and 2008 Global Routing Competitions. Referring to an example output format in figure 4.5, the nets are separated with "!" as the delimiter. For each net, the first line states the name of the net and its ID. The following lines will be all the edges of one unit that form a path connecting all the pins of the net. Note that since we are only routing one layer, the z-coordinate will always be 1.

```
net0 0 # <netName> <netID>
(0,1,1)-(1,1,1) # <pin x-coordinate>, <pin y-coordinate>, <pin z-coordinate>
!
net1 1
(0, 2, 1)-(1, 2, 1)
(1, 2, 1)-(1, 1, 1)
!
net2 2
(2, 2, 1)-(2, 1, 1)
(2, 1, 1)-(1, 1, 1)
(1, 1, 1)-(1, 0, 1)
!
```

FIGURE 4.5: Output format example

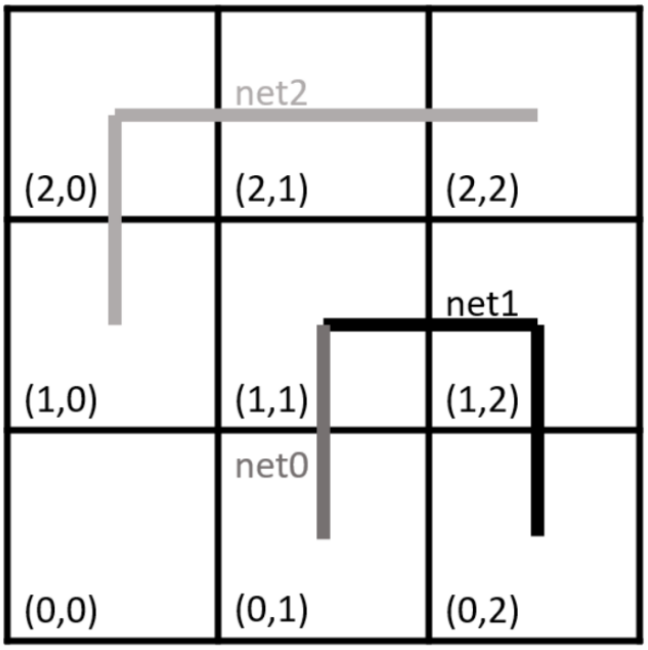


FIGURE 4.6: Output example in grid representation

4.2 Use Case Description

This section contains all the various use cases for the system. The use case descriptions and diagrams are to specify the expected behaviour of our system application.

4.2.1 Use Case: Login

Login	
Summary	User logs into the application
Pre Conditions	User has a modern browser and internet access
Primary Actors	User
Secondary Actors	System
Flow of Events	User: Enters site URL System: Displays login page User: Enters valid credentials User: Clicks login button System: Validate user credentials System: Logs user into their home dashboard
Alternative Flow	User: Enters site URL System: Displays login page User: Enter invalid credentials User: Clicks login button System: Validate user credentials System: Displays error message, remains at login page
Post Conditions	User is now logged into their dashboard

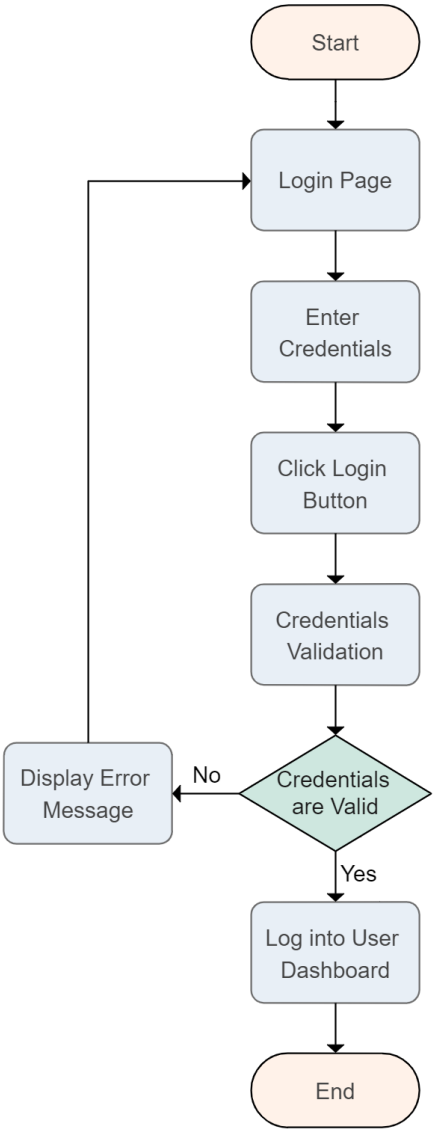


FIGURE 4.7: Login Use Case Diagram

4.2.2 Use Case: Registration

Registration	
Summary	User registers an account
Pre Conditions	User has a modern browser and internet access
Primary Actors	User
Secondary Actors	System
Flow of Events	User: Enters site URL System: Displays Account Registration page User: Enters user information, email and password User: Clicks register button System: Validate user info and credentials System: Creates a new account for the user
Alternative Flow	User: Enters site URL System: Displays Account Registration page User: Enters user information, email that already exists and password User: Clicks register button System: Validate user info and credentials System: Displays account email exists error message, remains at register page
Post Conditions	User has created a new account

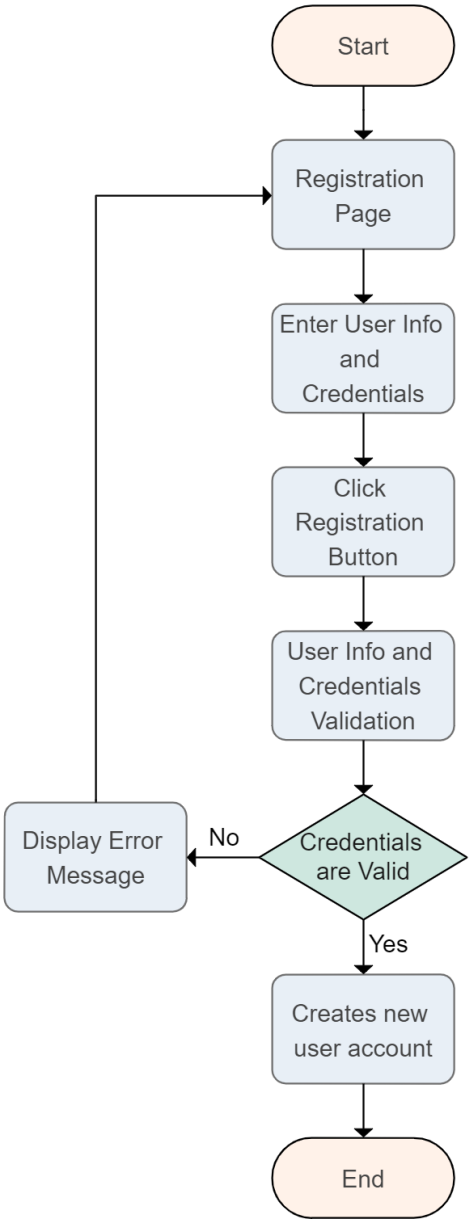


FIGURE 4.8: Register Use Case Diagram

4.2.3 Use Case: Route Netlist

Route Netlist	
Summary	User routes the input netlist
Pre Conditions	User is authenticated, the provided input netlist is following the specified input format
Primary Actors	User
Secondary Actors	System
Flow of Events	User: Logs into the home dashboard User: Uploads the netlist input file User: Selects a routing algorithm System: Parses and validates the input file System: Routes the netlist System: Generates a routed netlist output file
Alternative Flow	User: Logs into the home dashboard User: Uploads the netlist input file with an incorrect format User: Selects a routing algorithm System: Parses and Validates the input file System: Display input validation error message
Post Conditions	User's input netlist is routed, routed output file is generated

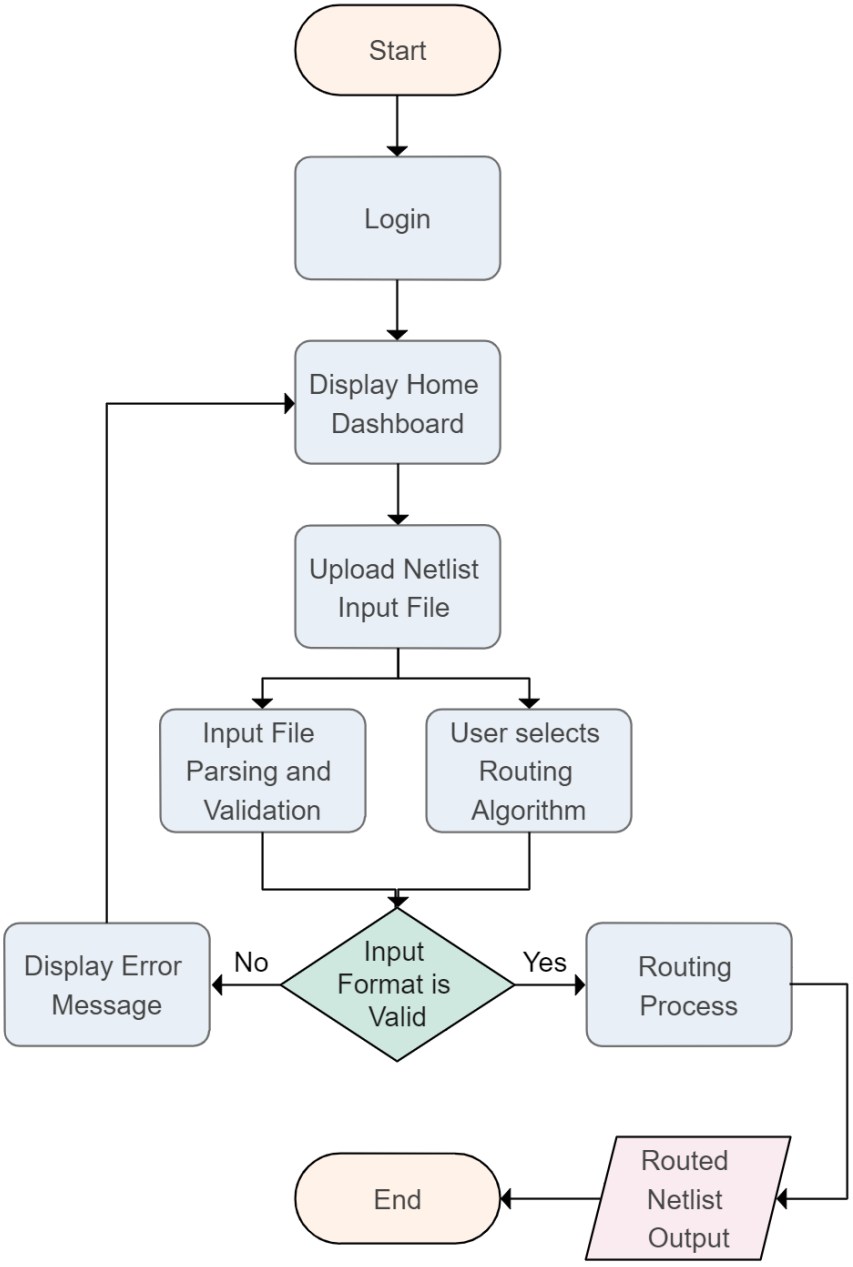


FIGURE 4.9: Route Netlist Use Case Diagram

4.2.4 Use Case: Save Output to System Storage

Save Output to System Storage	
Summary	User saves the generated output file to system storage
Pre Conditions	User just ran the routing process, and the system has just generated the routed output file
Primary Actors	User
Secondary Actors	System
Flow of Events	System: Generates a routed netlist output file User: Clicks on save to storage button System: Saves the routed netlist output file to system storage
Alternative Flow	System: Generates a routed netlist output file User: Clicks on save to storage button System: Storage connection error System: Displays error message
Post Conditions	User's routed netlist output file is securely saved into system storage

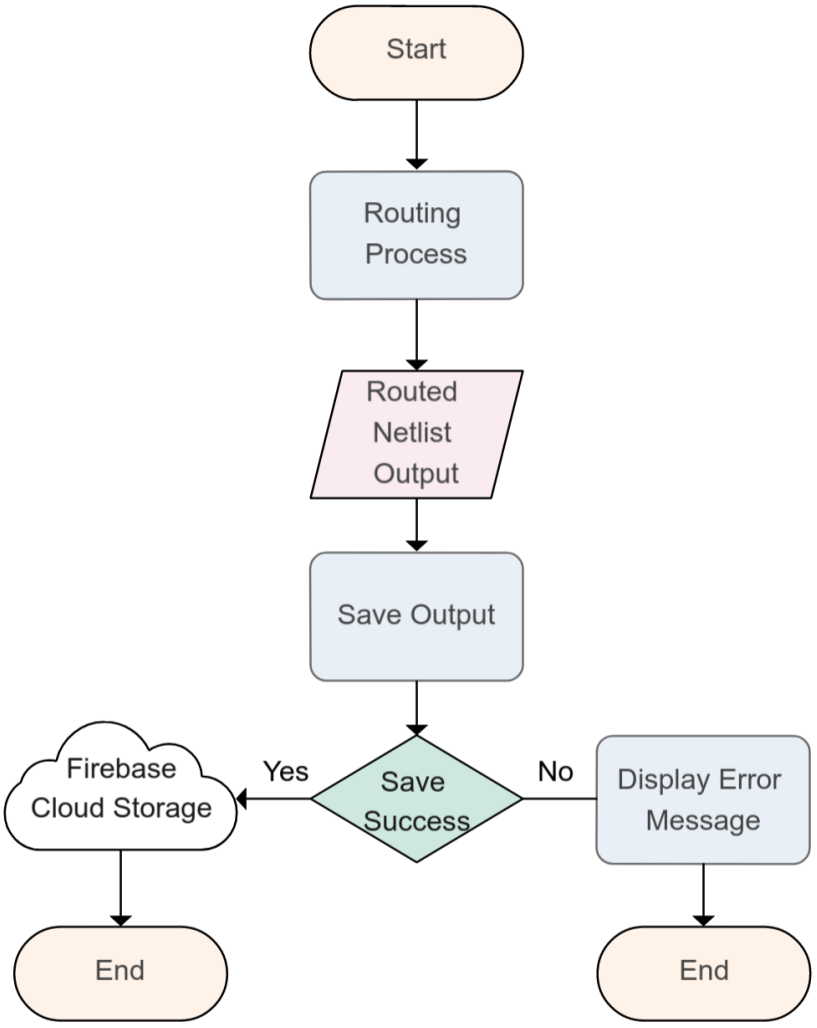


FIGURE 4.10: Saving Output Use Case Diagram

4.2.5 Use Case: View Routed Output Visualization

View Routed Output Visualization	
Summary	User views the path visualization of the routed output
Pre Conditions	User is logged into home dashboard, and a routed output file exists either selected from saved storage, or just newly generated
Primary Actors	User
Secondary Actors	System
Flow of Events	User: Selects the routed output file User: Clicks on view path visualization button System: Parses the output file System: Generates the routed output visualization System: Displays the visualization
Alternative Flow	User: Selects the routed output file User: Clicks on view congestion level map button System: Parses the output file System: Parsing error System: Displays error message
Post Conditions	User views and interacts with the visualization

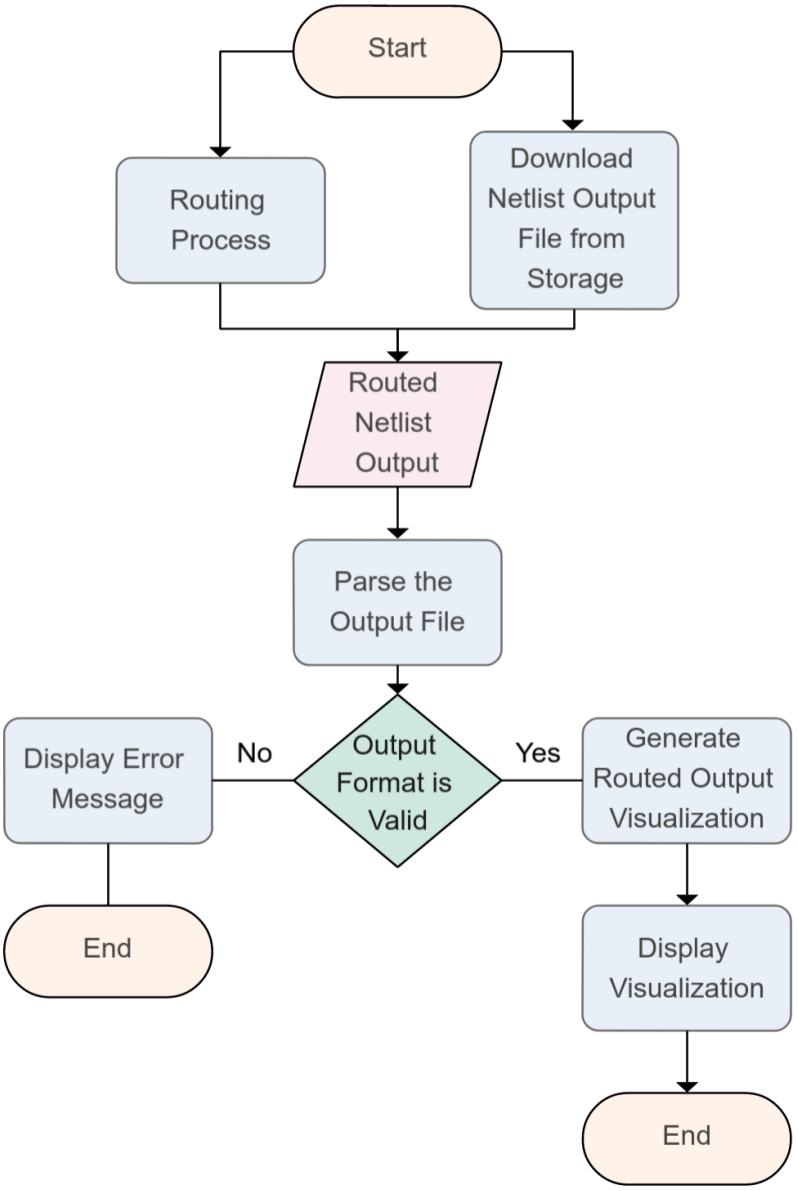


FIGURE 4.11: View Routed Output Visualization Use Case Diagram

4.2.6 Use Case: View Congestion Level Visualization

View Congestion Level Visualization	
Summary	User views the visualization of congestion level for the routed output solution
Pre Conditions	User is logged into home dashboard, and a routed output file exists either selected from saved storage, or just newly generated
Primary Actors	User
Secondary Actors	System
Flow of Events	User: Selects the routed output file User: Clicks on view congestion level map button System: Parses the output file System: Generates a congestion info file System: Generates a heat map based on the congestion info System: Displays the visualization
Alternative Flow	User: Selects the routed output file User: Clicks on view congestion level map button System: Parses the output file System: Parsing error System: Displays error message
Post Conditions	User views and interacts with the congestion level heat map visualization

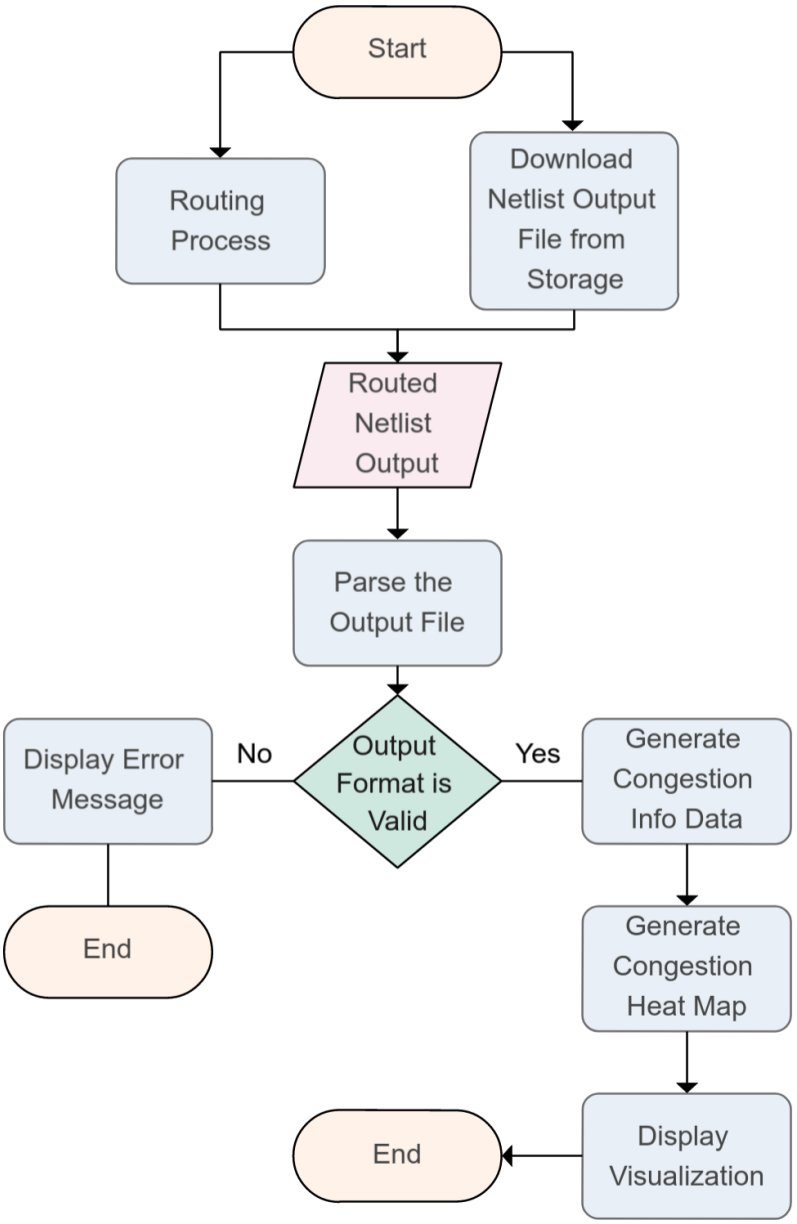


FIGURE 4.12: View Congestion Level Visualization Use Case Diagram

4.3 Risk Assessment

In this section, we will be identifying the potential risks that will preclude us from completing the project successfully. The risks will be classified in accordance to their severity and probability of arising. A mitigation plan for each risk will also be discussed. Table 4.1 outlines all the risks associated with this project and their corresponding severity and probability.

TABLE 4.1: Risk Table

Probability/ Severity	**1-Very Rare**	**2-Rare**	**3-Occasional**	**4-Probable**	**5-Very Probable**
4-Fatal		Risk 2: 4.3.2	Risk 1: 4.3.1		
3-Critical	Risk 3: 4.3.3		Risk 6: 4.3.6		
2-Major		Risk 4: 4.3.4			
1-Minor		Risk 5: 4.3.5			

4.3.1 Risk 1: Issues with Plotting Visualization Plots

- **Severity:** Fatal
- **Probability:** Occasional
- **Explanation:** The lack of experience and knowledge in plotting massive grid graphs and heat maps could slow down the development of the visualization features. It is classified as fatal because if the visualization features are not implemented, the project will not have any visuals and this will make the project very difficult to showcase later on. The assumption right now is that Pythons Matplotlib should be able to do the job, however that might not be the case.
- **Mitigation:** Allocate some time even before semester two to go through some online tutorials and resources to learn how to create these plots fit to the project's requirements. If Matplotlib is not suited for plotting, further research will be done.

4.3.2 Risk 2: Global Router Algorithms Implementation Struggle

- **Severity:** Fatal
- **Probability:** Rare

- **Explanation:** The implementation will be the most challenging part as the algorithm has quite a few constraints and optimization objectives to satisfy. The algorithm will not be just a pure path finding algorithm but rather an implementation customized and modified to fit the project's constraints and objectives. This is classified as fatal because this is the core component of the entire project, without it, the project will be deemed as a failure.
- **Mitigation:** Test cases will be written to validate the expected outcome of the algorithms. Benchmarking tests will also be carried out to gauge the performance *(congestion level, total wire length used)* of the algorithms. Further learning and research will be done throughout the implementation phase to create these global router algorithms.

4.3.3 Risk 3: Flask Learning Curve for Back-End Development

- **Severity:** Critical
- **Probability:** Very Rare
- **Explanation:** Lack of professional experience developing back-end with Python Flask could potentially slow down the development process. This is classified as fatal because this is the foundation of the web application platform for the users to interact with the routing algorithms.
- **Mitigation:** Spend some time even before semester two, to try to create a simple project with Flask, to get familiar with the stack.

4.3.4 Risk 4: Firebase Authentication Setup Issues

- **Severity:** Major
- **Probability:** Rare
- **Explanation:** Lack of professional experience developing with Firebase technologies. Firebase Authentication setup hiccups could arise and slow down the project development. It is deemed as only a major issue because ideally we want different users to have a different experience using the web application; and even without it, everyone can still access the same platform functionalities and resources.
- **Mitigation:** Spend some time before semester two reading the documentation for Firebase Authentication, and watch some tutorial videos online on how to quickly

set it up. At the meantime, a simple project with Firebase Authentication will also be created to make sure I fully understand how it works.

4.3.5 Risk 5: Firebase Cloud Storage Setup Issues

- **Severity:** Minor
- **Probability:** Rare
- **Explanation:** Lack of professional experience developing with Firebase technologies. Firebase Authentication setup hiccups could arise and slow down the project development. It is deemed as only a minor issue because ideally we want to allow users to store their generated routed output on the cloud for later access to visualize the output data, and even without this component it is okay as the user can still re-run the routing process to get the routed output they want to visualize.
- **Mitigation:** Spend some time before semester two reading the documentation for Firebase Cloud Storage, and watch some tutorial videos online on how to quickly set it up, and interact with it. At the meantime, a simple project with Firebase Cloud Storage will also be created to make sure I fully understand how it works.

4.3.6 Risk 6: Poor Time Management

- **Severity:** Critical
- **Probability:** Occasional
- **Explanation:** With poor time management, the progress of the project will be severely impacted. Promised features will not get delivered on time, or at all. The short time frame allocated for the project forces us to not procrastinate and meet the set deadlines.
- **Mitigation:** Follow the Gantt Chart in fig 4.13 as closely as possible. Define clear, concise, and small enough action items to execute.

4.4 Methodology

4.4.1 Scrum

In this project, we will be following some of the core principles of the Scrum framework. Scrum is typically used to solve problems iteratively and collaboratively by self-organizing teams that consist of multiple roles, such as the product owner, scrum master, and the scrum team. However, since I am the only contributor to the project, not all principles are feasible and applicable, so I will be only following the following principles [33]:

- **Defining Product Backlog** - The *product backlog* is a list to prioritize the work that has to be done. I will be adding, changing, and reprioritizing the backlog as I and the supervisor see fit. The items at the top of the backlog stack have more priority than the ones at the bottom.
- **Planning the Sprints** - During sprint planning phase, the supervisor and I will choose the next backlog items to work on for the sprint. The items will be chosen based on their priority and the achievability.
- **Execute the Sprints** - Once sprint planning is done, the sprint officially starts. I will be executing the sprint backlog, tackling one task at a time. I will also be setting up weekly meetings with my supervisor to keep him updated with my progress and show him some demos.
- **Sprint Review and Retrospective** - At the end of the sprint, I will be showing my supervisor what I have accomplished and a quick demo. At this stage I will also reflect on what went well and in which areas need more improvements, so that the next sprint will go smoother.

4.4.2 Kanban Board

To help us visualize the workflow of this project and help with sprint planning, a plain simple Kanban board is an excellent tool. The Kanban method is based on the principles of continuous improvement, collaboration, and transparency. Using a Kanban board, we can improve our ability to collaborate, communicate, and adapt to changing priorities and requirements. GitHub provides a Kanban board for every project repository, which is perfect as we will be using GitHub as our project repository hosting service. In the board, we will define three columns to represent the progress status of the tasks:

- **To Do** - This is the backlog column, all tasks here are yet to start.
- **In Progress** - This column holds all the tasks that are currently in progress.
- **Done** - This column holds all the tasks that have been completed.

This visual representation of the work helps us to see at a glance what work is currently being done, what needs to be done next, and where bottlenecks or potential roadblocks might occur.

4.4.3 Test Driven Development

Test-driven development (TDD) is a software development approach in which test cases are written for new code before the code is written. The idea behind this approach is that by writing tests first, we can think more critically about the functionality we are trying to build and write better code that is more likely to be correct and have fewer bugs. The benefits of using TDD include the following:

- **Improved code quality:** By writing tests before implementing the code, developers can ensure that the code they write is correct and meets the requirements.
- **Faster development:** By writing tests before implementing the code, developers can focus on writing the code necessary to pass the tests. This can help us avoid writing unnecessary code and speed up the development process.
- **Easier debugging:** If a test fails, it is easier to identify the cause of the failure and fix it. This can save time and effort that would otherwise be spent on manual debugging.
- **Better documentation:** The tests themselves serve as documentation for the code, making it easier for other readers to understand how the code works and what it is supposed to do.

Overall, TDD is a valuable approach to software development that can help us write better code and save time and effort in the development process.

4.5 Implementation Plan Schedule

Following the agile methodologies, we will break down the implementation phase into four major epics. Then, we can further break each epic into multiple smaller action

items. Breaking big tasks into multiple smaller ones will help us visualize the overall pending tasks more clearly. Tackling multiple small and feasible tasks is often more manageable than taking on a large task head-on. The four main epics, as well as their action items, are as follows:

1. Routing Algorithm Implementation
    (a) Implement netlist input parser
    (b) Implement global router based on Dijkstra's Algorithm
    (c) Implement global router based on A* Path Finding Algorithm
2. Back-end Setup
    (a) Firebase Authentication Setup
    (b) Firebase Cloud Storage Setup
    (c) Flask Server Setup
    (d) Define endpoint routes and their logic
3. Visualization Generation
    (a) Implement netlist output parser
    (b) Implement congestion data generation functionality
    (c) Generate congestion map
    (d) Generate routed output visualization
4. Front-end Setup
    (a) Jinja2 Template Setup
    (b) HTML Page Templates Setup
    (c) Bootstrap Setup

Given that there are 12 weeks for semester two, the general plan is to distribute all the action items into six sprints, each lasting for two weeks. Epics and action items with a higher priority will be tackled first. This is so that we will at least have the core functionalities of the project implemented and tested. For instance, the core of the project is to implement a global router algorithm, so we will focus on the algorithm implementation first and then move on to items with lower priority. The general plan is as follows:

4.5.1 Sprint 1

During Sprint 1, the focus will be the **(3a) implementation of a netlist input parser**. Once the system can parse and validate the input netlist, the initial **(3b) implementation of a global router based on Dijkstra's Algorithm** will begin.

4.5.2 Sprint 2

In Sprint 2, we will continue the **(3b) implementation of the global router algorithm based on Dijkstra's Algorithm** from Sprint 1. After that, we will create another **(3c) global router with A* Path Finding implementation**.

4.5.3 Sprint 3

In Sprint 3, we will focus on building the bare-bone web application. We will **1(c) set up the Flask server** and **1(d) define the endpoints** for our routing functions. Then, we will **2(a,b) create and render some basic HTML page templates with Jinja2** to let the user interact with the web application.

4.5.4 Sprint 4

During Sprint 4, the focus will be put on the **4(a) implementation of the netlist output parser**. Once the system can parse and validate the output netlist, we will be **4(d) implementing the function to generate the routed output visualization** and display it on the web application for the user. By the end of this sprint, we will have all the system's core functionalities implemented.

4.5.5 Sprint 5

During Sprint 5, we will then be **1(a) setting up Firebase Authentication** for user authentication, as well as **1(b) Firebase Cloud Storage** for saving the generated output files for later access. Next, we will implement the functionality to **4(b) create congestion data**, and the corresponding heat map and **4(c) display the heat map visualization** on the web application.

4.5.6 Sprint 6

Finally, in the final Sprint 6, we will focus on **2(c) making the look and feel of the web application better by using UI components from Bootstrap**. At this final stage, we will also be **doing some refinements** and working on any pending action items from previous sprints.

4.5.7 Implementation Plan Gantt Chart

Implementation Plan Gantt Chart

	Sprint 1	Sprint 2	Sprint 3	Sprint 4	Sprint 5	Sprint 6
Implementation of a netlist input parser						
Implementation of a global router based on Dijkstra's Algorithm						
Implementation of a global router based on A Path Finding Algorithm*						
Flask server setup						
Define endpoints and their logic						
Jinja2 template setup						
HTML page templates setup						
Implement netlist output parser						
Generate routed output visualization						
Firebase Authentication Setup						
Firebase Cloud Storage Setup						
Implement congestion data generation functionality						
Generate congestion map						
UI/UX improvement with Bootstrap						

FIGURE 4.13: Implementation Plan Gantt Chart

4.6 Evaluation

To evaluate the success of this project and measure how much of the goals are achieved, a combination of quantitative and qualitative metrics will be used.

To measure the global router algorithm's output, we will use a variety of metrics and tools. For measuring the global router algorithm's output, we will measure the congestion level, and the total wire length used. Our optimization objectives are to keep these two metrics as low as possible. A verifying tool will be developed to verify and analyze the output of the program.

In addition to measuring the algorithm outputs, it is also essential to gather and analyze user feedback to assess the satisfaction and impact of the overall project. This can be done through surveys, focus groups, user testing, and other methods that allow us to collect feedback directly from our stakeholders and users. For example, we could survey potential stakeholders to see if the visualization plots are clear and informative or if the web application is intuitive to navigate. By analyzing this feedback, we can identify areas for improvement and make necessary adjustments to our project to increase its effectiveness.

In a nutshell, this evaluation plan will allow us to measure the project's success and make informed decisions about improving and refining the project. By setting clear goals, using a variety of metrics and tools, and gathering and analyzing user feedback, we will be able to determine the impact and effectiveness of this project and make necessary adjustments to achieve all goals.

4.7 Prototype

This section covers the visual representation of the web application prototype. This prototype mock-up is how we envision the front-end look and feel of the project application once the implementation phase has been completed, and it is created using Figma [34].

Figure 4.14 shows the main landing page for the web application. On the left section, we have the title and description of the web application, as well as a *read more* button that further describes this application. On the right section, we have the login portal, which allows users to sign in with their email and password. There is a *forgot password* link below the login button if a user wants to reset the account password. On the bottom right, there are two links if a user wants to continue with Google Login or create a new account.

Figure 4.15 shows the registration page. The UI is identical to the login page, where the left section is the banner, while on the right side is the registration portal. Here, a user can also create an account with their Google account with the link on the bottom right.

Figure 4.16 and Figure 4.17 show the home dashboard page. Starting from the top, we have the top banner showing the web application title on the left; and user info on the right. A user can sign out from the application by clicking their profile icon and sign-out button in the dropdown menu.

On the dashboard section is an upload input form for a user to upload the netlist input file. The user then selects one of the routing algorithms to route the netlist and clicks on the route button. A status table below will display the status of the routing process. Status could be **pending**: Yet to be routed, **routing**: In routing process, or **complete**: Routing process completed. When a status for a routing process is complete, the user can save the output to the cloud storage for future access.

The visualization section has a dropdown on the right to select which output data to visualize. On the left is the tab pane to select *Routed Paths* tab or *Congestion Map* tab. The user can view the visualization of either Routed Paths or Congestion Map on the big viewing area below and be able to interact with it. Finally, on the bottom right, the user can download the raw output file and the plot image file.

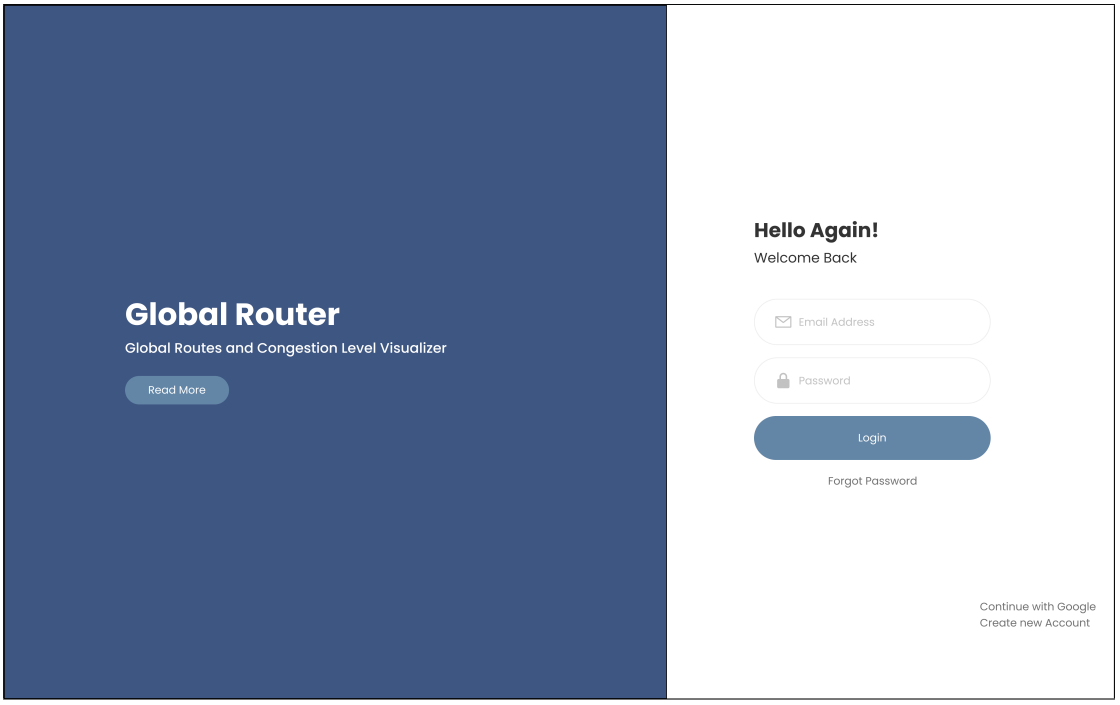


FIGURE 4.14: Prototype for login page

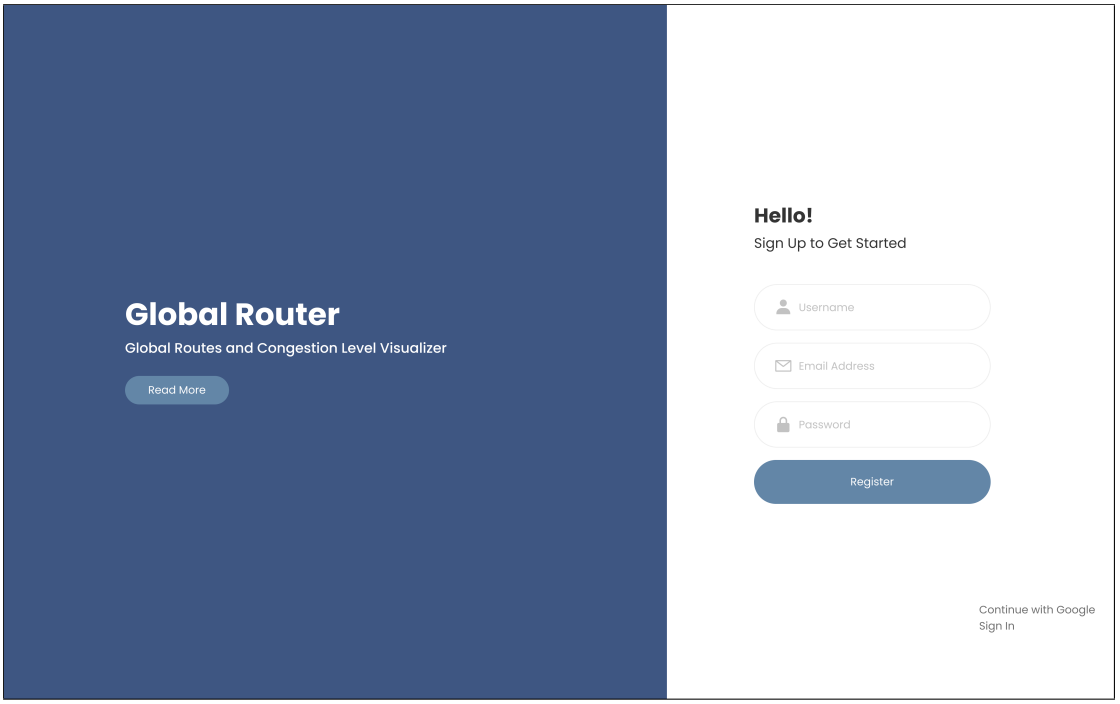


FIGURE 4.15: Prototype for registration page

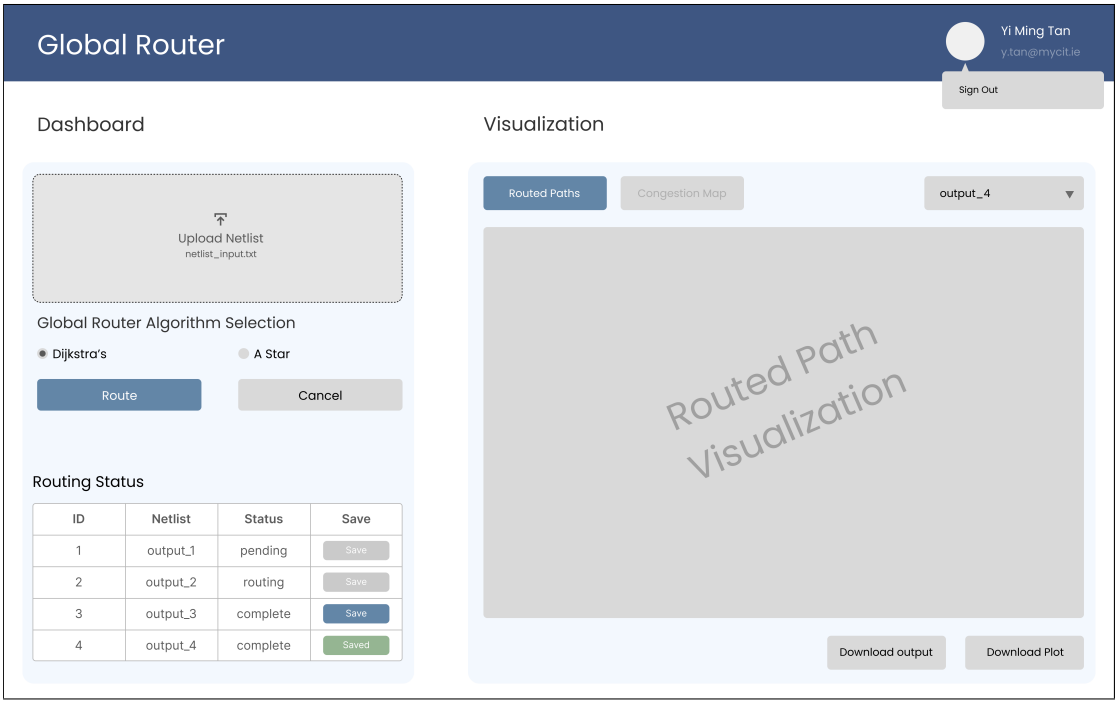


FIGURE 4.16: Prototype for dashboard page, displaying routed paths

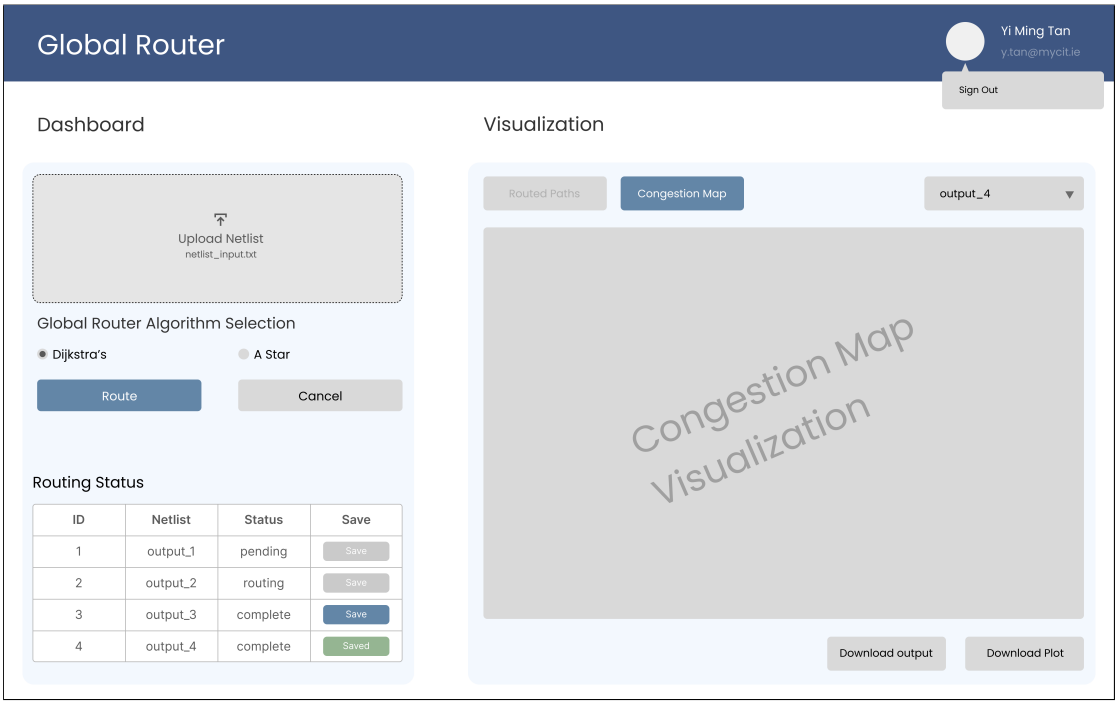


FIGURE 4.17: Prototype for dashboard page, displaying congestion map

Chapter 5

Implementation

In the upcoming chapter, we will analyze the challenges faced during the implementation of the project. These challenges will be classified into three categories: Low-level problems, Mid-level problems, and High-level problems. Low-level problems were easily resolved with minimal difficulty. Mid-level problems necessitated a more comprehensive resolution process. Finally, High-level problems were either too complex to be resolved or required an extensive resolution process.

In addition, this chapter will also discuss the solution approach adopted for the project, providing a detailed account of how the identified issues were tackled. This will offer readers valuable insights into the problem-solving process and the strategies employed to overcome the various challenges encountered.

5.1 Difficulties Encountered

5.1.1 Low-Level Problems

5.1.1.1 Firebase Realtime Database Free Tier Limitation

I encountered difficulties with Firebase's usage restriction on data downloads, which limited me to 10 gigabytes per month. This restriction posed a challenge for my project, as I used Firebase's real-time database services to store complex and data-intensive congestion plot SVG files. This resulted in my exceeding the prescribed download limit within a fortnight, which led to the suspension of my real-time database access.

The affected the architecture of my solution, as I needed to re-evaluate my approach to handling the congestion plot SVG data to prevent future issues with Firebase's usage

restriction. This difficulty represented a risk to my project, as it impeded the smooth flow of data to Firebase's real-time database services.

To manage the difficulty, I opted to initiate a new Firebase project to re-establish my access to a functional real-time database. Additionally, I re-evaluated my approach to managing the congestion plot SVG data to avoid exceeding the usage restriction in the future.

5.1.1.2 Enabling User to Download a Routed Output

I encountered difficulties identifying the most efficient approach to allowing users to download the output file resulting from the global routing process. To address this challenge, I needed to evaluate different storage solutions and determine the best fit for my purpose.

The difficulty affected the architecture of my solution, as I needed to integrate Firebase's storage product into my project's design to allow users to download the output file.

After considering various options, I opted to employ Firebase's storage solution, as it offers storage solutions for various types of files. At the end of the routing process, the resulting output file is saved directly to Firebase Storage, under each respective user's directory, to enable users to download the file efficiently.

5.1.2 Mid-Level Problems

5.1.2.1 Online Hosting

I needed to host my web application online for high availability and global accessibility. I explored two options - PythonAnywhere.com and AWS EC2. While PythonAnywhere.com was straightforward and required no additional setup, AWS EC2 required more configuration and setup. However, since I was on a free tier plan for both hosting platforms, the lightweight machines provided were not powerful enough to run my intensive algorithm. As a result, the processing time was significantly longer than on my local computer with a better CPU.

The architecture of my solution was impacted as I had to consider the limitations of the hosting platforms and their respective resource allocations. It represented a risk to my project as it could potentially affect the user experience due to slow processing times. It affected my methodology to develop my project as I had to work with limited resources and optimize my code for efficient resource utilization. It changed my implementation

schedule as I had to allocate more time for optimizing my code to work with limited resources.

To manage this difficulty, I optimized my code to be more efficient with limited resources and worked within the constraints of the hosting platform. I also explored other free hosting options and considered upgrading to a paid service plan but ultimately decided against it.

5.1.2.2 Job Monitor and Visualization Section Design

Previously, the global routing process was a blocking and sequential operation. This meant that when the user clicked on the submit netlist button, they had to wait for the global routing process to finish and display on the visualization section before moving on to another task. This was particularly problematic for complex input netlists, as the global routing process took a long time.

It affected the architecture of the solution by requiring asynchronous JavaScript calls to send global routing requests to the server and enabling the server to handle multiple global routing processes at once. It represented a risk to the project by potentially causing user frustration due to the long wait times.

To manage this difficulty, asynchronous JavaScript calls were created to send global routing requests to the server, and a job monitor was developed to display all the global routing processes running in the background. This resulted in the implementation of a non-blocking process for routing netlists asynchronously, saving users time and preventing potential frustration.

5.1.2.3 Algorithms and Data Structures Exploration

Initially, the lack of knowledge and understanding of global routing concepts made it challenging to grasp during the research phase, particularly for someone without a VLSI background. Even after understanding the problem statement and what global routing is, the implementation solution was not clear, specifically regarding the data structure to use and the pathfinding algorithm or routing algorithm to use.

This difficulty affected the project design in terms of the implementation strategy and methodology. It required extensive exploration of different data structures such as priority queues, binary heap, and Fibonacci heap, and algorithms such as breadth-first search, best-first search, and A* search algorithm. It also added more complexity to the routing process.

To overcome this difficulty, I conducted extensive research and studied different data structures and algorithms to find the most optimized solution. After thorough exploration and learning, I was able to implement the global routing process using appropriate data structures and routing algorithms. I also prioritized optimization since the code was in Python, which can be slow, and needed to ensure that the process was efficient.

5.1.2.4 Congestion Map Visualization Form

I wanted to display the congestion data of the routed output in an interactive and high-resolution format on the web application. Initially, I used Matplotlib to generate an image of the plot and embedded it on the visualization section, but it wasn't interactive and lacked high resolution.

To overcome the difficulty, I explored using the mpld3 library, which translates Matplotlib figure data and generates SVG data along with HTML code of the congestion plot. This allowed me to embed the SVG data on the visualization section of the web app and interact with it, such as zooming in and out and moving it around. This way, the congestion map was displayed interactively and in the highest resolution possible.

5.1.3 High-Level Problems

5.1.3.1 Slow Performance in Python

Python is the programming language in which I possess the highest proficiency. Nevertheless, processing the massive input netlists provided to the project using Python proves significantly slower than languages like C or C++. To mitigate this issue, I devoted a substantial amount of time to optimizing the Python code to improve its runtime. However, using C or C++ would have resolved this issue more efficiently.

To optimize the code, I explored various approaches, including parallel programming using multiprocessing and multithreading. Implementing multiprocessing proved successful, enabling me to run multiple global routing processes simultaneously. On the other hand, multithreading to split the netlist into multiple parts to process each single global routing process did not work as the routing process depends on the previous iteration's data, requiring sequential processing.

The time spent optimizing the code in Python caused a delay in the implementation schedule. In conclusion, I made the program faster by implementing multiprocessing and not multithreading.

5.1.3.2 Understanding NP-Hard Problems, Heuristic Algorithms

Through researching articles and papers, I discovered that global routing in VLSI is classified as an NP-hard problem. At first, I was unfamiliar with this concept, but I delved deeper and gained a better understanding of it. Essentially, global routing is both an optimization and heuristic problem, possessing both stochastic and deterministic properties. The stochastic aspect involves the randomization of the netlist order during routing, while the deterministic property entails sorting the netlist by the half perimeter wire length or bounding box. In the process of solving this challenging problem, I gained extensive knowledge about various algorithms, their properties, and characteristics.

I conducted extensive research to understand the nature of the problem and its different algorithmic solutions. I experimented with various algorithms and heuristics to identify the most effective approach for the given problem. This led to the development of a custom algorithm that balanced optimization and speed, resulting in an efficient global routing solution.

5.1.3.3 Challenges in Generating Actual Routed Paths Visualization

During the implementation planning phase, one of the challenges was to find a suitable library or framework that could be used to create a visualization section for the actual routed output such as shown in figure 4.6 on the web application. Despite having all the necessary data, I could not find a library or tool to convert this data into a visual output.

It affected the original project design, as the visualization section was an essential component of the web application's architecture, and without it, the project was incomplete. This difficulty represented a significant risk to the project, as the absence of a visualization component would affect the user experience and the overall value of the application. The difficulty affected the methodology to develop the project, as the original plan had to be revised to accommodate the absence of the visualization section.

I tried to find alternative visualization libraries and considered using matplotlib to plot the data. However, I could not find a suitable solution, so I ultimately decided to exclude the visualization component and focus on other areas of the project.

5.2 Actual Solution Approach

This chapter will outline the proposed approach to address the problem and devise a solution. It will provide a comprehensive description of the project's architecture and the core global routing algorithm. Additionally, we will examine all developed use cases for the final product. Furthermore, we will scrutinize the risks identified in the previous chapter to determine if they manifested and, if so, how we mitigated them. The methodology for developing the solution will also be discussed, and we shall assess the implementation schedule to determine if all components were delivered according to the plan. If not, we will analyze how to improve future projects. Finally, the final product will be presented with screenshots, and we will discuss its functionality in detail.

5.2.1 Overall Architecture and Algorithm Breakdown

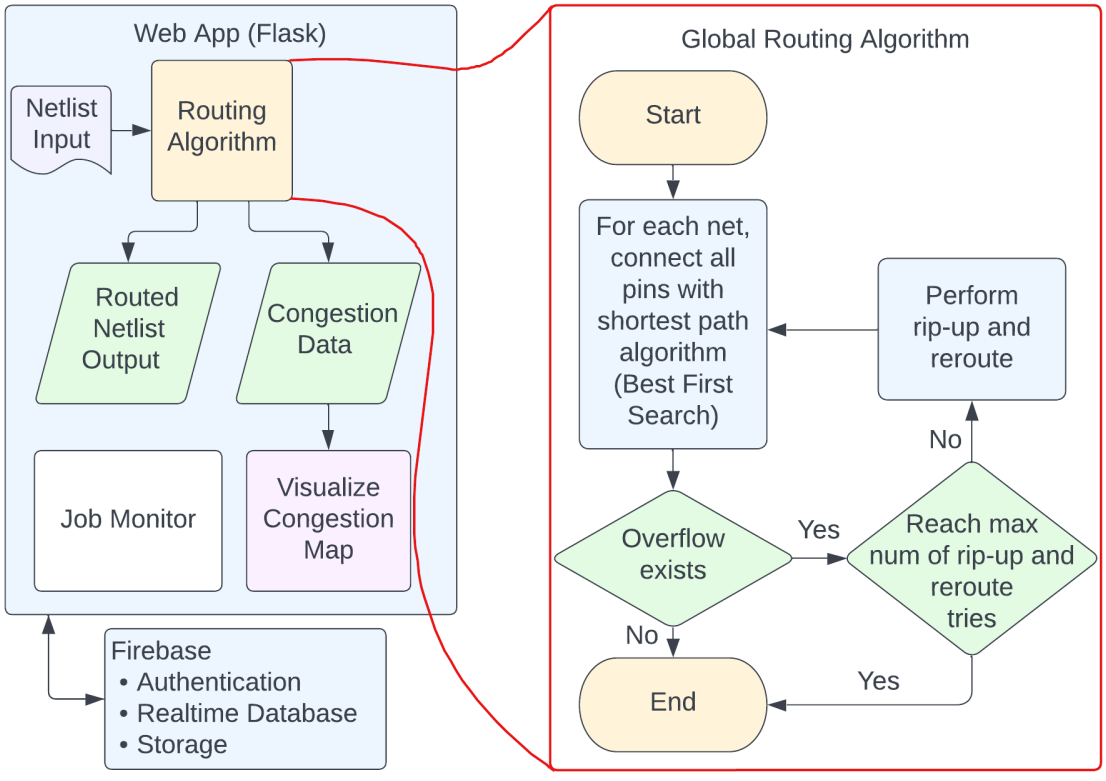


FIGURE 5.1: Final Product System Architecture

5.2.1.1 Overall System Architecture

As illustrated in the figure 5.1 above, the system architecture is characterized by a straightforward design. Specifically, the web application backend operates on a Python Flask server. The Flask server serves as the primary mechanism for handling global

routing logic and flow. Additionally, the Flask server facilitates the rendering of HTML pages through the Jinja templating system. Consequently, the end-user can interact with the web application and provide the netlist input. Once the netlist input is received, it is processed through the routing algorithm. The routing algorithm then produces the routed netlist output in addition to the congestion data associated with the layout. Subsequently, the congestion data is subjected to further processing, generating a congestion heat map visualization.

In this context, Firebase serves as a multifunctional platform for user authentication, real-time database operations, and storage management. Specifically, Firebase authentication is leveraged to manage the login and registration processes. When a user creates an account, email verification is mandatory before the initial login attempt. The Firebase real-time database is utilized to store all user data, including their netlist data and other relevant information. In addition, Firebase storage is employed to store the output files generated by the system. The output files are grouped based on the user's directory, thereby allowing users to easily access and download their respective output files.

Upon reviewing the system architecture depicted in the previous section, it is noted that it closely adheres to the initial plan. However, one notable deviation is the absence of a feature to visualize the routed netlist paths. The final product only provides the congestion output visualization. This disparity is attributed to the technical constraints discussed in the previous section 5.1.3.3

5.2.1.2 Global Routing Algorithm

This algorithm is a heuristic approach designed to address the global routing problem, an NP-hard optimization problem. Due to the complexity of the problem, exact optimal solutions are infeasible, and heuristic methods are required to obtain good solutions within a reasonable timeframe. Thus, the algorithm relies on heuristics to approximate an optimal solution that satisfies the given constraints.

For this heuristic algorithm, there exists both **stochastic property** and **deterministic property**.

- Stochastic: The algorithm employs random decision-making techniques, such as utilizing a pseudo-random number generator. As a result, it is highly probable that two separate algorithm runs will yield distinct solutions. Simulated annealing is a prominent example of a stochastic algorithm.

- Deterministic: The algorithm's decisions are deterministic, meaning that they can be repeated. Dijkstra's shortest path algorithm is an instance of a deterministic heuristic.

And as for this heuristic algorithm's structure, it is typically as follows (diagram in Figure 5.2):

1. Solve the problem with constructive algorithm and create an initial solution
2. While the termination criterion is not met, do iterative improvement on the initial solution
3. Return best-seen solution

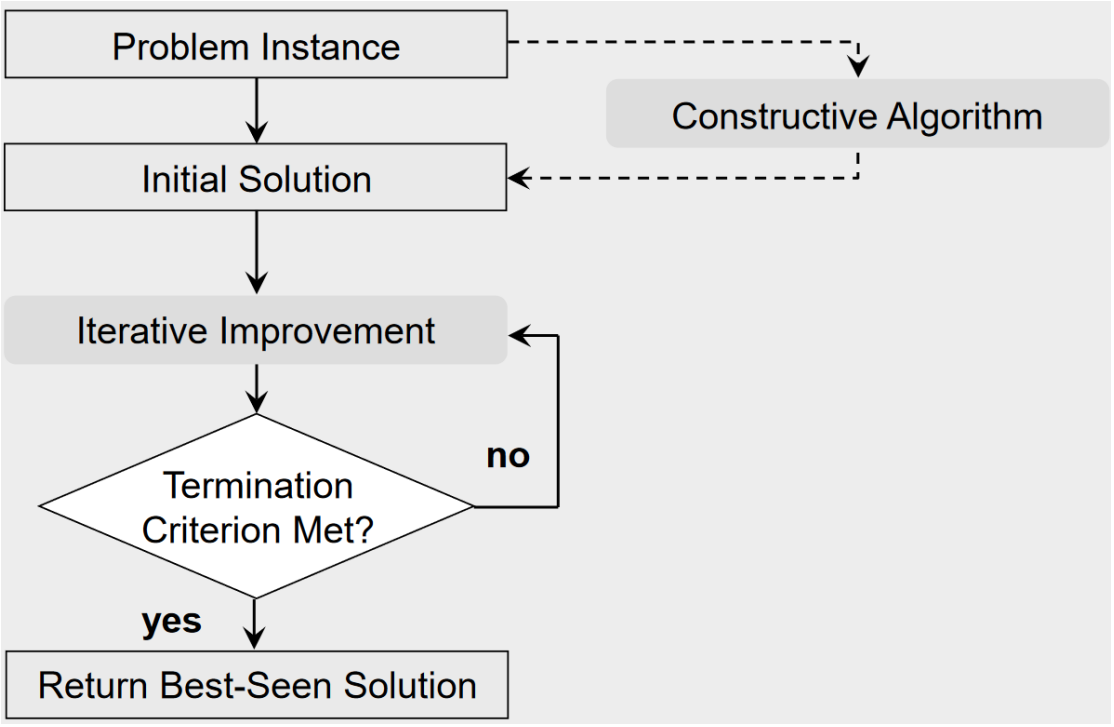


FIGURE 5.2: Heuristic Algorithm Structure

The algorithm employed in this project adheres to the structure and framework of a heuristic algorithm. In the subsequent discussion, a thorough explanation of each step and detail of the algorithm will be provided.

Firstly, upon receiving the netlist input, the global routing algorithm parses the supplied netlist file. Subsequently, it proceeds to construct and initialize essential data structures, including the grid data structure, which encompasses the grid size, horizontal and vertical capacities of the grid, among other attributes. In addition, the list of net data

structures is established, comprising the net ID, net name, and coordinates of the two pins to be connected, among other relevant features.

Next, we aim to create an initial solution by routing a path for each net in the netlist, connecting the two pins of each net in the netlist sequentially. This will be the **constructive algorithm** part in the heuristic algorithm structure. The sequence of routing the nets is critical, as the output results in different wirelength and overflow values. Nonetheless, due to the inherent stochastic and deterministic properties of this heuristic algorithm, randomization can be employed to shuffle the net routing order for the stochastic aspect, hoping to generate high-quality output. Meanwhile, for the deterministic aspect, it is imperative to sort the netlist by the half perimeter wire length (HPWL). To clarify, HPWL refers to the summation of the Manhattan distances among all pairs of pins in the net. Sorting the netlist based on HPWL can considerably enhance the output result. Specifically, larger nets can be more efficiently routed around smaller ones that have already been routed, reducing routing impediments and optimizing the utilization of routing resources. Upon implementing the constructive algorithm, an initial solution can now be generated. However, this solution may be of sub-optimal quality, which necessitates undergoing iterative improvements in the subsequent stage.

The pin routing algorithm is responsible for creating the shortest possible path to connect the two pins of a net, while also ensuring that there is no overflow. To reiterate, overflow occurs when a wire is placed on a channel track that has reached its maximum capacity. When routing two pins with the shortest wire length, the most efficient path will typically be a straight line or L-shaped path. A naive breadth-first search algorithm can achieve this result without considering congestion information in channel tracks. This method ensures that all paths routed for all nets have the shortest possible wire length. However, since congestion information is not considered, the outputs will likely have a high overflow, which can negatively impact the quality of the output. To achieve an output with lower overflow, it is necessary to compute congestion data, which is the demand and capacity ratio on the channel track. This information must be considered when deciding the path to connect the two pins.

Consider Figure 5.3, where the current node is depicted as a yellow circle. To reach the next node, any of the four directions can be considered. Assuming that the horizontal and vertical capacities are 14 and 16, respectively, the cost of reaching each adjacent node can be calculated. Moving east requires placing two wires on the channel track, while moving north incurs a cost of 12 wires. Moving west incurs a cost of eight wires, and moving south requires 16 wires. Notably, the south node has already reached its maximum vertical capacity, so traversing there would be prohibitively expensive. Therefore, the node with the lowest cost and a demand of only two is the east node.

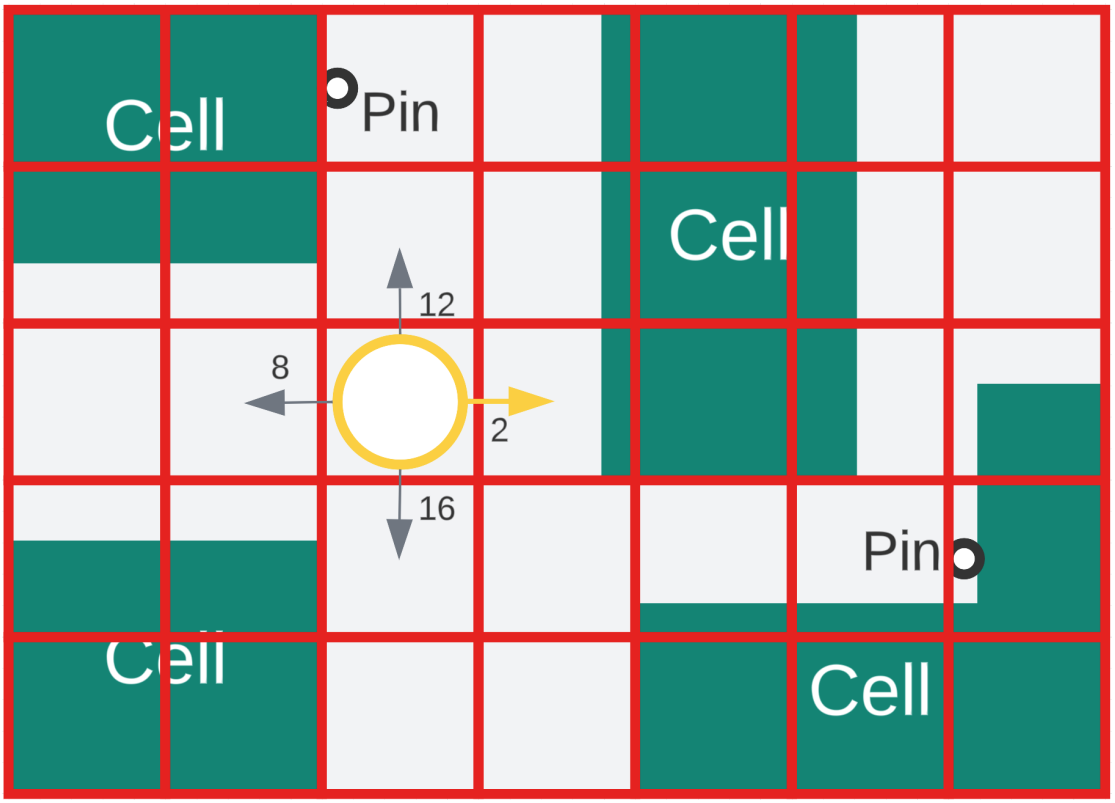


FIGURE 5.3: Path Routing with Best First Search

To implement this, the Best First Search algorithm is used. The Best First Search algorithm is similar to breadth-first search, but it maintains a priority queue to keep track of the next node to be visited. The node with the lowest cost, which in this context is congestion data, is selected from the queue and expanded. This process is repeated until the goal node is found or there are no more nodes to explore. When expanding, the next node can be in the north, south, east or west direction, and the cost will be the demand of the edge to go to the next node. The node with a low-demand edge will have a low cost and be more likely to be chosen as the next node. If the edge's demand has already reached maximum capacity, an enormous cost will be assigned to prevent it from being chosen as the next node.

The stochastic nature of the algorithm implies that the quality of the output generated by a run is uncertain. One idea is to leverage multiprocessing capabilities to execute multiple routing processes simultaneously. By generating several results with varied quality, each with a unique randomized net routing order, we can identify the most optimal solution among them. Subsequently, the best solution serves as the initial solution for the iterative improvement stage.

In this project's context, the iterative improvement process is referred to as the "rip-up and reroute" phase. This involves identifying the problematic nets that lead to overflow

and "ripping them up", followed by re-routing attempts to reduce the overflow in the layout. The termination criterion for this stage is either when the overflow reaches zero or when the maximum number of rip-up and reroute attempts are reached. Upon satisfying the termination criterion, the resulting output is a complete set of paths for all nets with the best-seen quality yet.

Essentially, the implementation of this global routing algorithm closely resembles the one suggested in 2.12

5.2.2 Risk Assessment

In this section, we will evaluate the risks identified in Chapter 4 Section 4.3 to determine if they materialized and how I handled them.

5.2.2.1 Risk 1: Issues with Plotting Visualization Plots

Despite dedicating significant time to researching libraries and frameworks that could effectively represent the routed path output, I could not find a suitable solution. This issue arose from the complexity of the grid graph and heat map visualizations, which required extensive knowledge and experience in plotting massive data. Despite the mitigation plan of allocating time to go through online tutorials and resources to learn how to create these plots to fit the project's requirements, it was still difficult to find a suitable library. While Python's Matplotlib was initially considered as a potential solution, further research revealed that it might not be sufficient for this project's specific requirements. As a result, the search for a more suitable solution is still ongoing.

5.2.2.2 Risk 2: Global Router Algorithms Implementation Struggle

Despite the challenges of implementing a solution to the global routing problem with a heuristic algorithm, I successfully created a customized implementation that satisfied the project's constraints and objectives. This challenging task required extensive research and learning throughout the implementation phase. I conducted benchmarking tests to evaluate the algorithm's performance and the quality of the outputs, such as if all routed paths are valid and not floating, measuring the congestion level and the total wire length used. While the implementation process was demanding, I was able to create a solution that met the project's requirements and achieved the desired results.

5.2.2.3 Risk 3: Flask Learning Curve for Back-End Development

Contrary to the initial risk assessment, the learning curve for Flask's back-end development was not a significant issue. Despite having limited professional experience with Flask, I was able to accomplish everything I wanted with the stack. The Flask documentation was thorough and easy to understand, and there was a large and supportive community available to provide assistance when needed. Additionally, Flask's user-friendly interface made it straightforward to use and develop the necessary features for the web application platform. Ultimately, the concern regarding the Flask learning curve proved to be unfounded, and the platform was a suitable choice for the project's requirements.

5.2.2.4 Risk 4: Firebase Authentication Setup Issues

Fortunately, the risk of Firebase Authentication setup issues did not materialize during the project development. I was able to set up Firebase Authentication with ease, thanks in part to Firebase's user-friendly interface and the completeness of their documentation. Additionally, I utilized the Pyrebase library to manage user authentication in Firebase, which simplified the process even further. In summary, the initial concern regarding Firebase Authentication setup proved to be unfounded, and the platform was straightforward to use and implement in the project.

5.2.2.5 Risk 5: Firebase Cloud Storage Setup Issues

Fortunately, the setup for Firebase Cloud Storage was not an issue at all. With the help of Firebase's detailed documentation and tutorial videos online, I was able to quickly and easily set it up. I was able to save all the output netlist files grouped by user, just as I had planned. Everything worked smoothly for me, and I'm happy to say that this risk did not pose a challenge during the project development.

5.2.2.6 Risk 6: Poor Time Management Towards The End

Time management is a crucial aspect of any project, especially when juggling multiple responsibilities at once. In the initial stages of the project, time management was well-planned and executed efficiently. However, as the final few weeks approached, allocating enough time to each subject and project became increasingly challenging. The amount of assignments and deadlines from other subjects began to pile up, making it challenging to devote as much time as necessary to the research project. Despite these obstacles, a successful balance was still maintained, and the necessary work was completed. The

ability to manage time effectively allowed for the project to remain a priority while still attending to other commitments. In the end, despite some setbacks, the project was able to progress, and some new ideas were even implemented.

5.2.3 Solution Development Methodology

As discussed in Chapter 4 Section 4.4, I decided to use the Scrum methodology for managing the development process for this project. I began by defining the product backlog, which included all the features that I wanted to implement in the web application and the algorithm. Then, I divided the development process into sprints and conducted sprint planning for each sprint to decide on the features to be implemented. After completing each sprint, I conducted a sprint review to evaluate the progress made during the sprint and to identify any areas for improvement.

To track all my tasks, I used GitHub's project board, which is similar to a Kanban board. I created different columns on the board to represent different stages of the development process, namely "Backlog", "Ready", In Progress, "In Review", and "Done". Each task was created as a separate card on the board, and I would move the cards across the different columns as I progressed with the development process. Using this board helped me to keep track of all the tasks that needed to be completed and ensured that nothing fell through the cracks. Overall, using the Scrum methodology and the project board on GitHub helped me to manage the project effectively and ensured that I stayed on track to meet my deadlines.

In addition to using Scrum methodology and GitHub's project board, I proposed to use test-driven development (TDD) in Chapter 4 Section 4.4. However, for this project, I did not implement TDD. The reason being, I did not find it necessary for this particular project. I believe TDD is more suitable for code refactoring, where changes can potentially break the codebase. Since I was building the project from scratch, I felt that writing tests before the code was unnecessary. That being said, I still included unit tests for my project, but they were written after the code was developed.

Figure 5.4 displays a comprehensive list of all unit tests developed for the model package. These tests examine all modules within the package, including the global_router, grid, net, and path modules, ensuring that everything is functioning as expected.

Ensuring comprehensive unit testing is a critical aspect of software development. It is recommended that all code be thoroughly tested and covered by unit tests. A reliable method of measuring test coverage is by utilizing Python's coverage tool. This tool

FIGURE 5.4: List of Unit Tests

provides a metric for determining the extent to which tests exercise the code. In short, Python coverage is a powerful tool for evaluating the effectiveness of unit testing efforts.

In Figure 5.5, we showcase how coverage is executed. We use pytest to invoke the unittests and pipe that command to coverage, which generates a detailed report about the tests. We specify the source as the models directory since we are only testing modules within the models package. Figure B also confirms that all 15 test cases pass with no issues, and a coverage file is then generated right after.

Figure 5.6 presents the coverage report for our tests, with an average total of 81%. While a high coverage score does not necessarily equate to quality tests, we are satisfied with our current coverage score.

```
ming@ubunga:~/Documents/MTU/global-router$ coverage run -m --source=models --omit=__init__.py pytest tests/
========================================= test session starts ==========================================
platform linux -- Python 3.11.2, pytest-7.3.1, pluggy-1.0.0
rootdir: /home/ming/Documents/MTU/global-router
collected 15 items

tests/global_router_test.py .......                                                          [ 46%]
tests/grid_test.py .....                                                                     [ 80%]
tests/net_test.py ..                                                                         [ 93%]
tests/path_test.py .                                                                         [100%]

========================================== 15 passed in 0.14s ==========================================
```

FIGURE 5.5: Coverage Run with Pytest

Coverage report: 81%

filter...

coverage.py v7.2.5, created at 2023-05-01 10:48 +0100

Module	*statements*	*missing*	*excluded*	*coverage*
models/global_router.py	195	48	0	75%
models/grid.py	42	5	0	88%
models/net.py	22	0	0	100%
models/node.py	10	0	0	100%
models/path.py	14	0	0	100%
Total	**283**	**53**	**0**	**81%**

coverage.py v7.2.5, created at 2023-05-01 10:48 +0100

FIGURE 5.6: Coverage Report in HTML

Lastly, implementing a logging functionality in a Python application is crucial for several reasons. Firstly, it provides valuable insights into the application's behaviour, performance, and errors. The logs generated by the application can be analyzed to identify issues, debug problems, and optimize performance. Secondly, logging can help with auditing and compliance by recording user activities, system events, and security-related activities. Additionally, logs can be used to generate reports, track usage patterns, and monitor application usage. Finally, logging is an essential tool for collaboration, as it allows multiple developers to work on the same codebase while maintaining a clear understanding of the application's behaviour. In this project, logging functionality is implemented to capture various aspects of the application's execution. This includes tracking the time taken by each function to run, recording output statistics, and detecting any errors that occur during runtime. By logging these details, valuable insights into the behavior of the application are gained, issues are diagnosed quickly, and monitor its performance over time. An example log is shown in 5.7.

Overall, using Scrum methodology and GitHub's project board, along with having unit tests and logging, were sufficient for managing the project and ensuring its quality.

5.2.4 Implementation Schedule Review

This project's implementation plan schedule has been a crucial element in the journey towards success. A burndown chart is developed to compare the ideal timeline with the actual timeline and track our progress. This chart includes Figure 5.8, which displays all the tasks and their sizes, and their distribution across the timeline, and Figure 5.9, which illustrates the comparison between the ideal and actual timelines. Despite poor time management towards the final weeks due to more assignments coming up, I was delighted to discover that the actual timeline closely resembles our estimated ideal timeline. This is a significant accomplishment as it demonstrates that I managed to complete most of the tasks within the allocated timeframe. Overall, I am thrilled with the progress we've made and are confident in our ability to continue working towards meeting our project goals.

5.2.5 Final Product Showcase

This section offers a detailed summary of the final product, encompassing all of its components and features. Furthermore, we have included a guide to its user interface for ease of use. In addition, we will be contrasting the implemented features with the prototype presented in chapter 4 Section 4.7.

Figure 5.10 shows the main landing page for the web app. The user logs into their account by simply entering the email and password. Users can also access the documentation by clicking on the *Read Documentation* link.

Figure 5.11 shows the registration page. To create an account, the user must provide their information and complete the registration process. Afterward, they will receive a prompt to verify their account using the email address they used to sign up.

The login and registration page implemented is very similar to what was planned initially.

Upon successful login, the user will be directed to the dashboard page. At the top section of the page, the user will find options to select a netlist input, an algorithm selector, and a random seed input. The user has the flexibility to either upload their own netlist file or choose a sample netlist file from the dropdown menu. The random seed input is used to shuffle and arrange the netlist in a specific order for reproducibility purposes.

Once the selection is made, the user can initiate the routing process by simply clicking on the "submit netlist" button.

After submitting a routing job, users can monitor the progress of their request through the job monitor. This feature displays the job's name, status, completion timestamp, and provides a convenient button for viewing the output plot.

The congestion visualization section contains a convenient tabbed pane that enables users to switch between plots of various routing jobs. Within each plot, users can easily zoom in and out, as well as move the plot around, using intuitive controls. Additionally, there's a handy button to toggle a modal that displays the plot legend. Below the plot, users can view the netlist details, as well as access a button to download the output file containing the routed paths.

The dashboard page and its functionalities closely align with our initial plan, with the exception of the routed paths visualizer, which was not implemented as discussed in Section 5.1.3.3

Lastly, it's worth noting that users also have the option to route the netlist and generate output using the command line interface. For more detailed instructions, please refer to the document available at the following Documentation Link (https://global-router.readthedocs.io/en/latest/cli.html).

logs > gr.log

```
2023-05-01 15:57:47,796 [INFO]: Running: parse_input
2023-05-01 15:57:47,904 [INFO]: benchmarks/ibm01.modified.txt parsed successfully, data structures created
2023-05-01 15:57:47,905 [INFO]:
Layout and Netlist Details
==========================
Grid: 64 x 64
Vertical Capacity: 12
Horizontal Capacity: 14
Number of nets: 13357

2023-05-01 15:57:48,990 [INFO]: Number of Global Routers in parallel: 5
2023-05-01 15:57:49,404 [INFO]: Running: route
2023-05-01 15:57:49,404 [INFO]: Generating God Seed...
2023-05-01 15:57:49,404 [INFO]: Using God Seed: 2661196334825363483
2023-05-01 15:57:49,474 [INFO]: Running: route
2023-05-01 15:57:49,475 [INFO]: Generating God Seed...
2023-05-01 15:57:49,475 [INFO]: Using God Seed: 908420623344048847
2023-05-01 15:57:49,557 [INFO]: Running: route
2023-05-01 15:57:49,558 [INFO]: Generating God Seed...
2023-05-01 15:57:49,558 [INFO]: Using God Seed: 6975555856833548978
2023-05-01 15:57:49,603 [INFO]: Running: route
2023-05-01 15:57:49,604 [INFO]: Generating God Seed...
2023-05-01 15:57:49,605 [INFO]: Using God Seed: 5629749344996364220
2023-05-01 15:57:49,635 [INFO]: Running: route
2023-05-01 15:57:49,636 [INFO]: Generating God Seed...
2023-05-01 15:57:49,637 [INFO]: Using God Seed: 4649242936849001993
2023-05-01 15:58:06,806 [INFO]: Total Overflow: 2
2023-05-01 15:58:06,807 [INFO]: Total Wirelength: 60229
2023-05-01 15:58:06,820 [INFO]: Function route executed in 17.2171s
2023-05-01 15:58:07,035 [INFO]: Total Overflow: 1
2023-05-01 15:58:07,035 [INFO]: Total Wirelength: 60549
2023-05-01 15:58:07,047 [INFO]: Function route executed in 17.4121s
2023-05-01 15:58:07,220 [INFO]: Total Overflow: 3
2023-05-01 15:58:07,222 [INFO]: Total Wirelength: 60445
2023-05-01 15:58:07,238 [INFO]: Function route executed in 17.8348s
2023-05-01 15:58:08,686 [INFO]: Total Overflow: 1
2023-05-01 15:58:08,686 [INFO]: Total Wirelength: 60399
2023-05-01 15:58:08,696 [INFO]: Function route executed in 19.1393s
2023-05-01 15:58:09,257 [INFO]: Total Overflow: 3
2023-05-01 15:58:09,257 [INFO]: Total Wirelength: 60509
2023-05-01 15:58:09,266 [INFO]: Function route executed in 19.7917s
2023-05-01 15:58:11,904 [INFO]:
============================
    Initial Routing Result
============================
Best Router Index: 2
Best Router Seed: 6975555856833548978
Best Overflow: 1
Best Wirelength: 60399
============================

2023-05-01 15:58:11,904 [INFO]: Rip-up and reroute #1:
2023-05-01 15:58:11,905 [INFO]: Running: rip_up_and_reroute
2023-05-01 15:58:20,041 [INFO]: Function rip_up_and_reroute executed in 8.1362s
2023-05-01 15:58:20,041 [INFO]: New Min Overflow: 0
2023-05-01 15:58:20,041 [INFO]: New Min Wirelength: 60259
2023-05-01 15:58:20,042 [INFO]:
============================
     Final Routing Result
============================
Best Router Index: 2
Best Router Seed: 6975555856833548978
Best Overflow: 0
Best Wirelength: 60259
============================

2023-05-01 15:58:20,042 [INFO]: Running: dump_result
2023-05-01 15:58:20,179 [INFO]: Output successfully dumped into output/ibm01.modified.out
2023-05-01 15:58:20,197 [INFO]: Congestion data generated into output/ibm01.modified.out.fig
```

FIGURE 5.7: Example log file for routing an IBM01 input

Global Router Project Burdown Chart														
Tasks	Initial Estimate	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13
Implement netlist input parser	1	1												
Firebase Authentication Setup	1	1												
Boostrap Setup	1	1												
Flask Server Setup	1		1											
Set up automated CI/CD for the project	1		1											
Set up Sphinx Documentation	1		1											
Host Sphinx Document Online	1		1											
Add Modules and CLI documentation	1			1										
Extract Firebase and Project Secrets to .env file	1			1										
Implement Global Router algo with simple BFS	5		1	1	1	1					1			
Add readthedocs config yaml	1				1									
Add GUI command to CLI	2			1	1									
HTML Page Templates Setup	1			1										
Define endpoint routes and their logic	4					2	1	1						
Jinja2 Template Setup	1			1										
Generate Congestion Map	4					2		2						
Implement Congestion Data Generation Functionality	4				3	1								
Netlist Sorting for More Optimal Solution	2					2								
Implement Multiprocessing to have multiple workers in parallel	4						3	1						
Implement Congestion Visualization with SVG	3							3						
Integrate Output Verifier	1										1			
Implement Ripup and Reroute	3								2		1			
Set up Firebase Cloud Storage to store output files	1										1			
Implement download output file feature	1										1			
Add timestamp to each routing process	1										1			
Explore best first vs breadth first	1											1		
Add legend modal	1											1		
Add random seed input to recreate simulation	2											1	1	
Explore fibonacci heap vs binary heap for priority queue	3											2	1	
Add algorithm selection best first and breadth first	3												2	1
Code Optimization	2											2		
Host project on PythonAnywhere.com	3							2		1				
Host project on AWS	3												3	
Remaining Effort	65	62	57	51	45	37	33	24	22	21	15	8	1	0
Ideal Line	65	60	55	50	45	40	35	30	25	20	15	10	5	0

FIGURE 5.8: Tasks Table

FIGURE 5.9: Burndown Chart

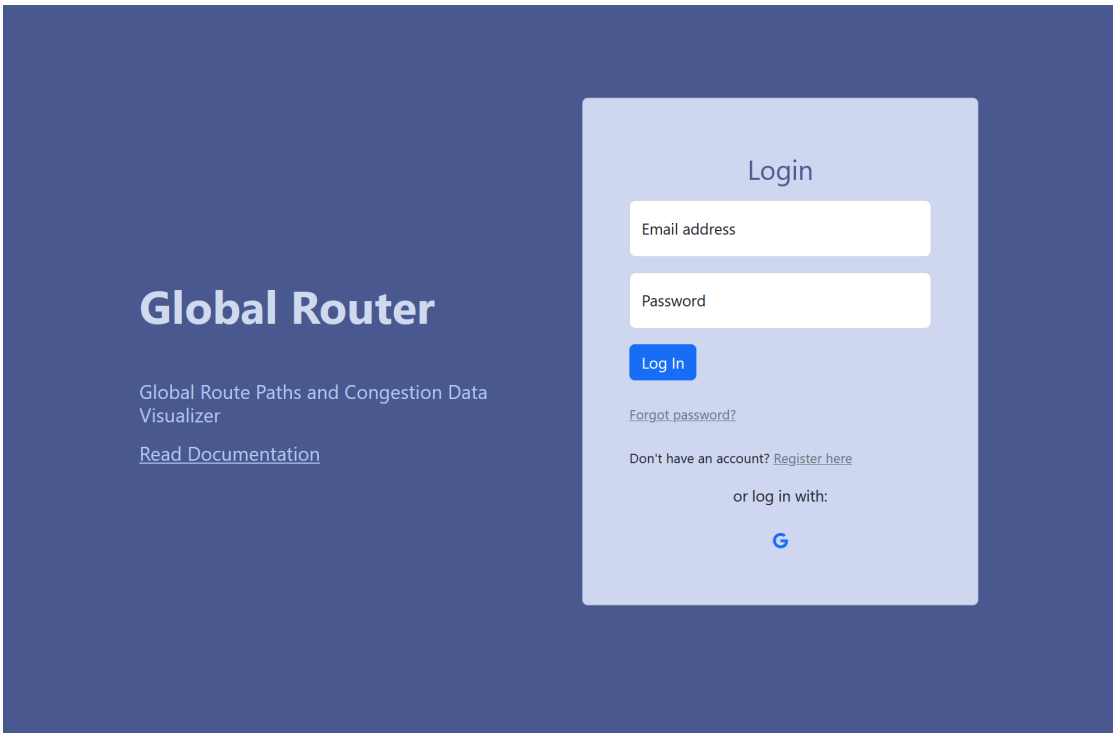


FIGURE 5.10: Login Page

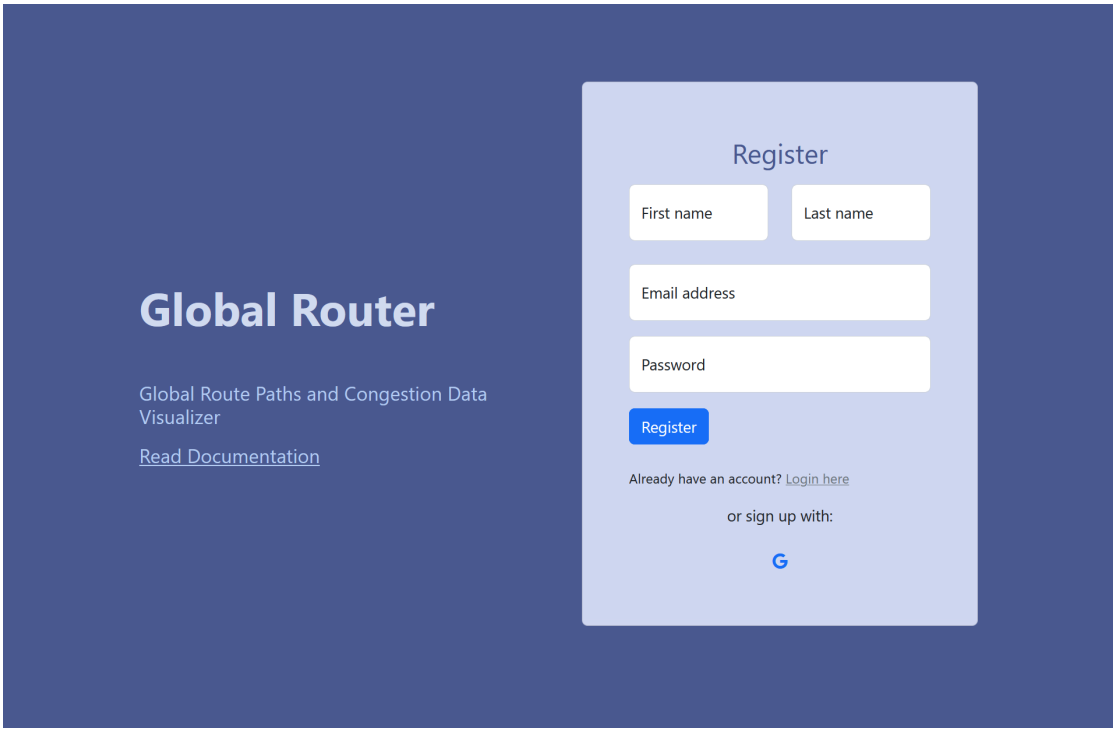


FIGURE 5.11: Register Page

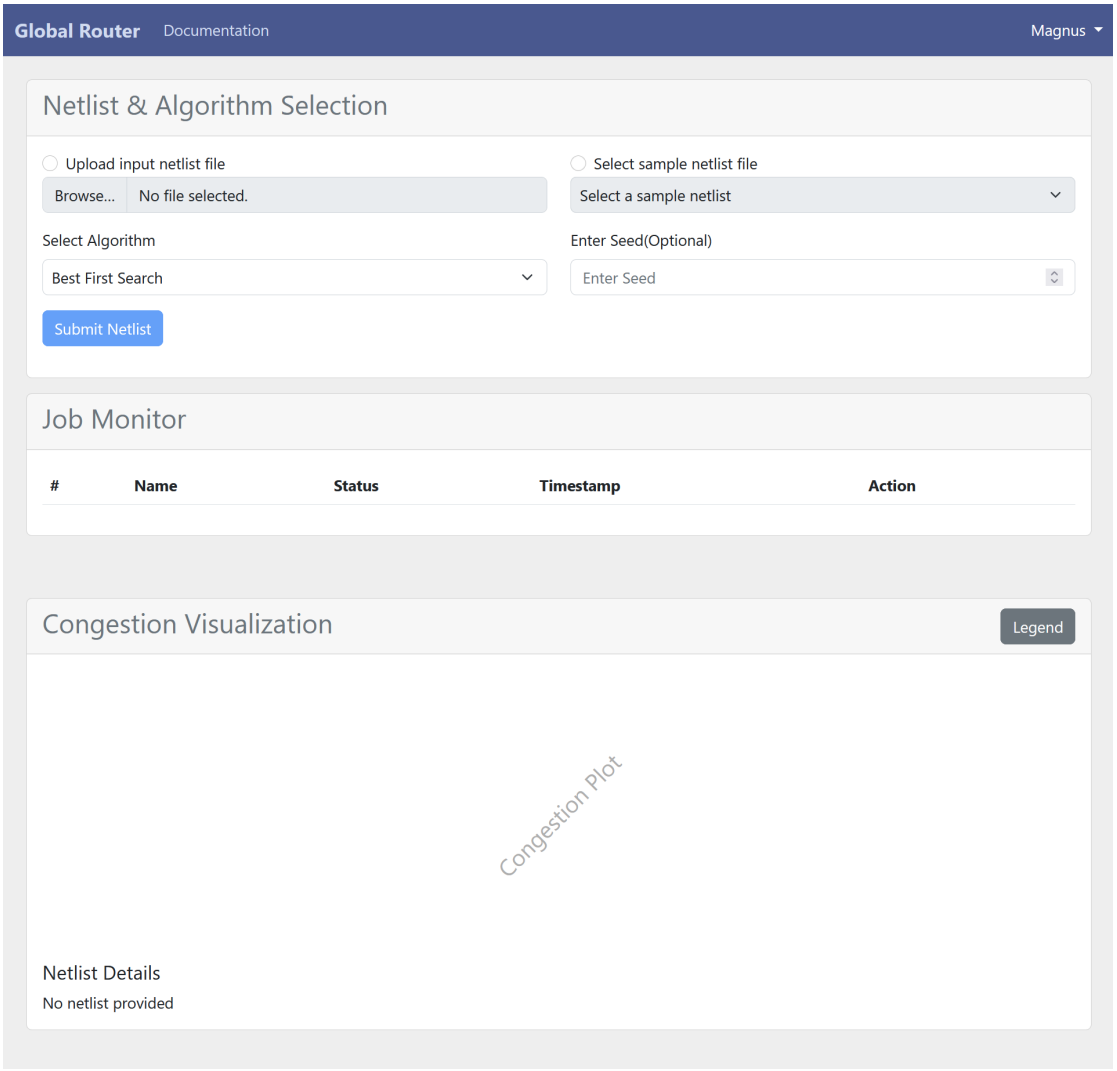


FIGURE 5.12: User Dashboard

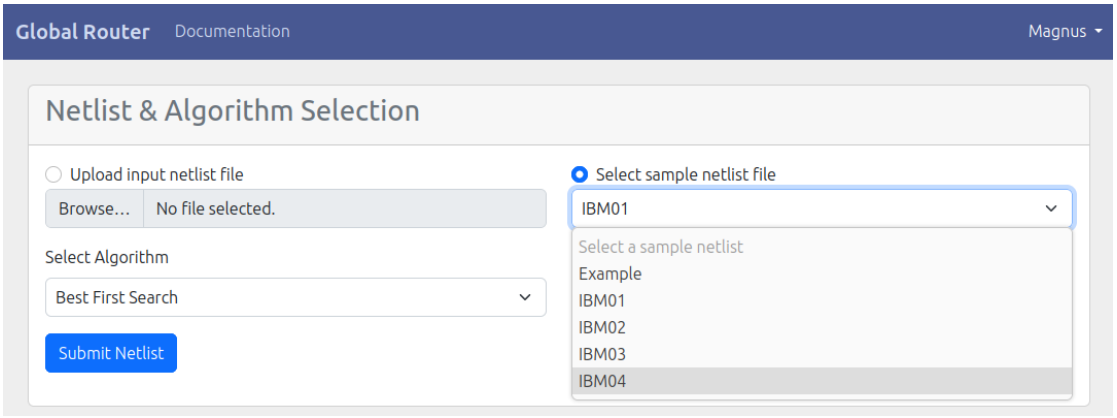


FIGURE 5.13: Sample Netlist Select

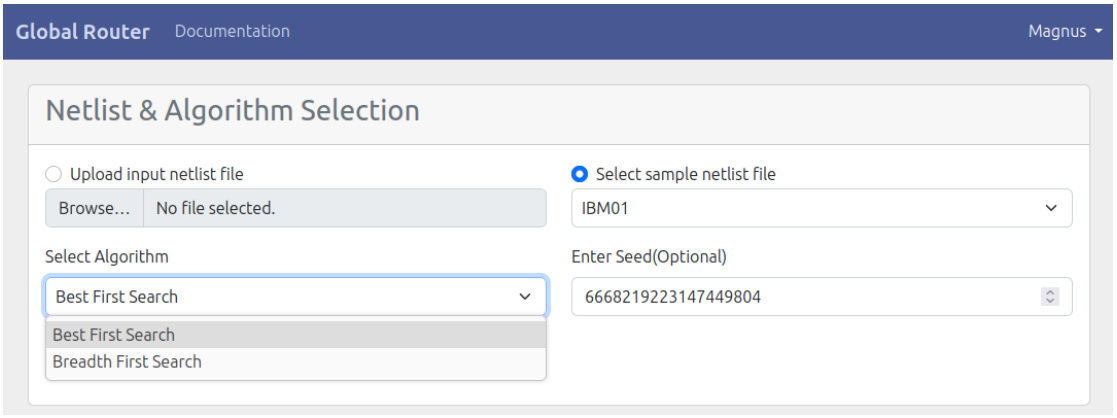


FIGURE 5.14: Algorithm Select

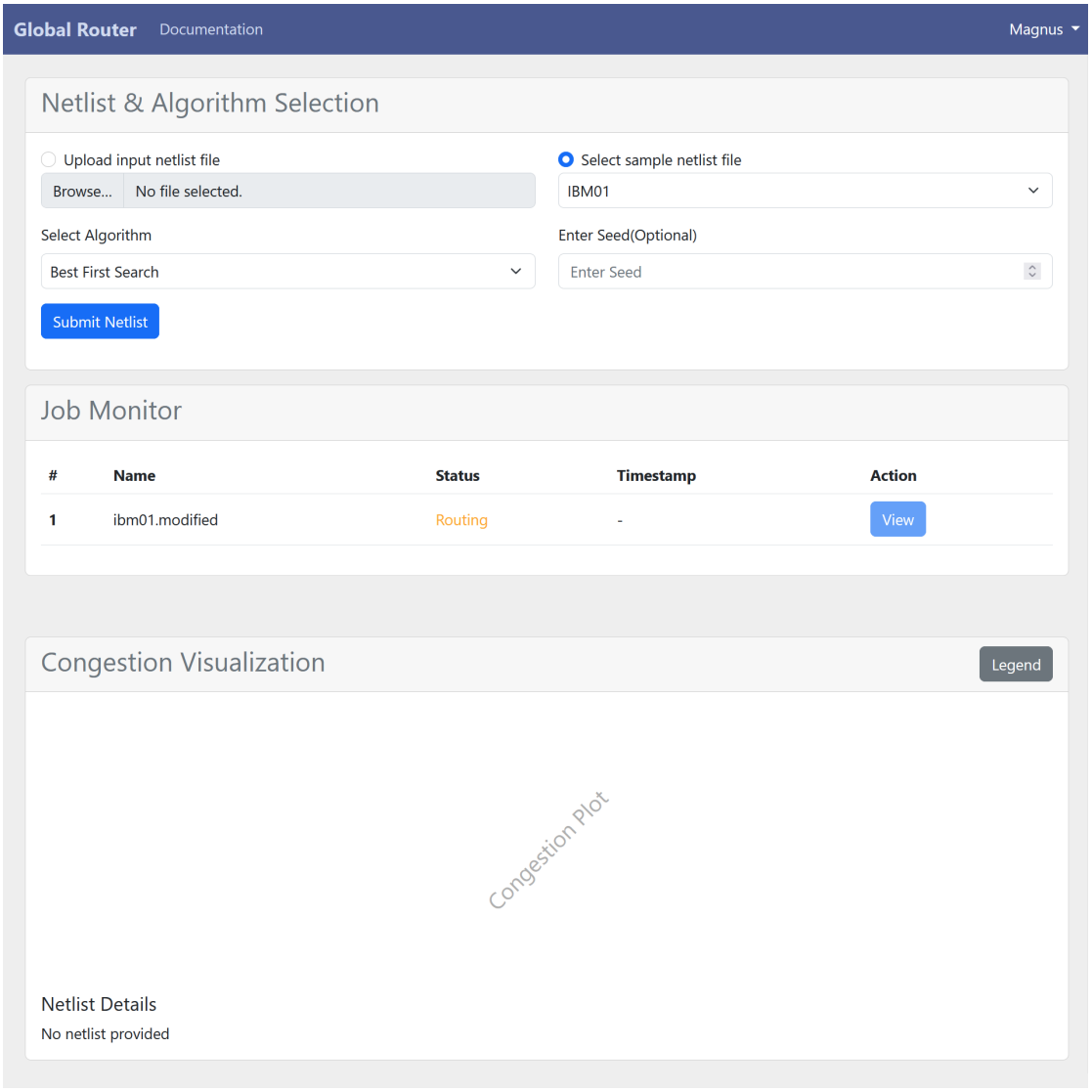


FIGURE 5.15: Select a netlist file and an algorithm then submit it, the routing process will appear in the job monitor

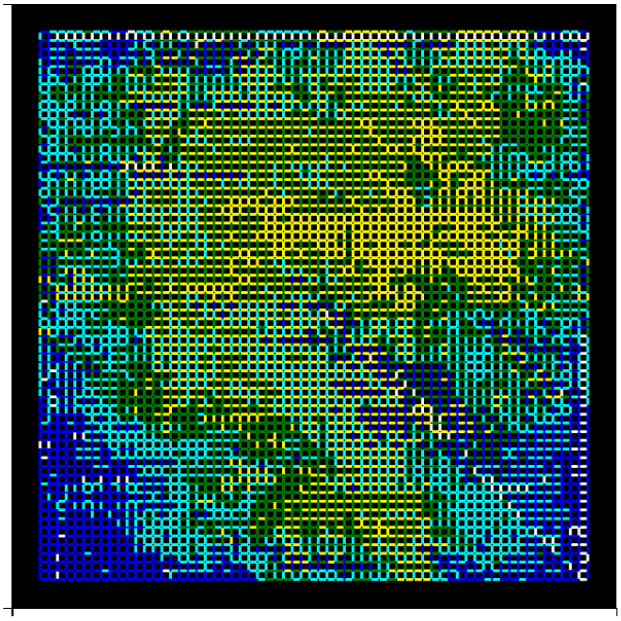
Job Monitor

#	Name	Status	Timestamp	Action
1	ibm01.modified	Ready	2023-04-29 12:50:22	Opened
2	ibm01.modified	Ready	2023-04-29 16:43:44	Opened

Congestion Visualization

Legend

Plot-1 Plot-2



Netlist Details

- Name: ibm01.modified
- Grid: 64 x 64
- Horizontal Capacity: 14
- Vertical Capacity: 12
- Netlist Size: 13357
- Overflow: 0
- Wirelength: 60277
- Timestamp: 2023-04-29 16:43:44
- Algorithm: Best First Search
- Seed: 6668219223147449804

Download Output

FIGURE 5.16: Routing is ready, open the plot and view

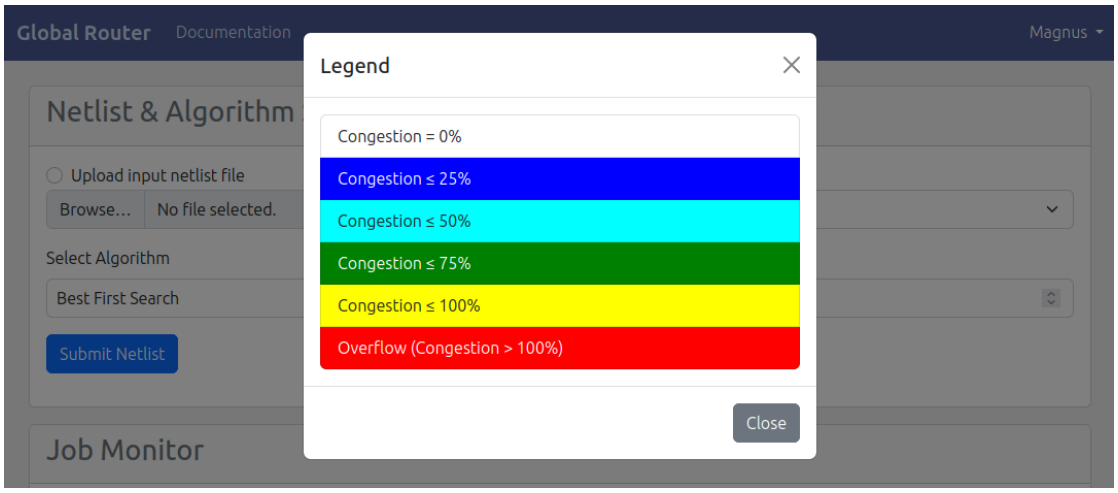


FIGURE 5.17: Plot Legend

Chapter 6

Testing and Evaluation

6.1 Metrics

In evaluating the quality of the output result generated by the global router algorithm, we have identified two key metrics: **overflow** and **wirelength**. The lower the wirelength, the better, and the lower the overflow, the better. Ideally, the overflow should be 0, making it our primary objective. Our secondary objective is wirelength, and we prioritize low overflow and longer wirelength over more overflow with shorter wirelength. In addition to these objectives, we also consider the runtime of the algorithm in seconds as our tertiary objective. By tracking these metrics, we can ensure that our global router algorithm generates the best possible output results for our project.

6.2 Benchmarks

Several benchmark files have been obtained as part of the NTHU CS6135 HW5 Global Routing assignment , namely example, IBM01, IBM02, IBM03, and IBM04. The following bullet points provide a general idea of the benchmarks, including the size of the grid and the number of nets for the inputs:

- **Example**
  - Grid Size: 3×3
  - Netlist Size: 3
- **IBM01**
  - Grid Size: 64×64

  - Netlist Size: 13357

- **IBM02**
  - Grid Size: 80×64
  - Netlist Size: 22465
- **IBM03**
  - Grid Size: 80×64
  - Netlist Size: 21690
- **IBM04**
  - Grid Size: 96×64
  - Netlist Size: 27781

6.3 Results

To streamline the benchmark runs, a shell script was developed to automate the entire process. Figure 6.1 provides an overview of all benchmark run results, which are obtained by utilizing the verifier tool. This tool, which will be further elaborated on later, generates results for overflow, wirelength, runtime, and the status of the run. Automating the benchmark runs and using the verifier tool help ensure accuracy, consistency, and efficiency in our project's evaluation process.

The tabulated results presented in Table 6.1 below indicate the routed outputs for the aforementioned sample netlists. Notably, the first three sample netlist inputs demonstrate zero overflows and therefore do not necessitate any rip-up and reroute operations. In contrast, the IBM04 netlist exhibits an overflow of 91 even after performing ten iterations of rip-ups and reroutes. Figures 6.3 to 6.7 show the congestion data for all the outputs while Figure 6.2 illustrates the corresponding legend for the congestion maps.

6.4 Output Validation

In order to ensure the correctness of the output, a verifier tool obtained from the NTHU CS6135 HW5 Global Routing assignment is integrated into the project. This tool is capable of validating the output produced by my algorithm and detecting anomalies such as floating wires (i.e., wires that are not connected to the pins of the net) and

```
|-------------------------------------------------------------|
|                                                             |
|                    Final Assessment Results                 |
|                                                             |
|-------------------------------------------------------------|
  testcase |   overflow | wirelength |    runtime | status
     ibm01 |          0 |      59931 |      12.98 | success
     ibm02 |          0 |     157688 |      23.00 | success
     ibm03 |          0 |     142580 |      23.10 | success
     ibm04 |         99 |     160884 |     221.69 | success
```

FIGURE 6.1: All Benchmark Runs Results

	IBM01	IBM02	IBM03	IBM04
Overflow	0	0	0	91
Wirelength	59931	157688	142580	160884
Initial Routing Time (s)	12.98	23	23.1	20.59
Num of Rip-up and Reroutes	0	0	0	10
Rip-up and Reroute Time (s)	N/A	N/A	N/A	201.1

TABLE 6.1: Output Table for all Test Cases

duplicate wires. Additionally, it provides final result statistics and the cost of the routed netlist. Examples of verification runs are illustrated in Figures 6.8 and 6.9.

It is worth noting that the algorithm's deterministic property involves arranging the netlist in ascending order of its half perimeter wirelength (HPWL). This arrangement enables the global router to generate much better quality output and perform significantly better. A detailed explanation for this can be found in Chapter 4, Section 5.2.1.2. In Figures 6.10 and 6.11, we shall compare the differences in total overflow and wirelength, in the context of all four benchmarks, between sorting the netlist in HPWL order and without.

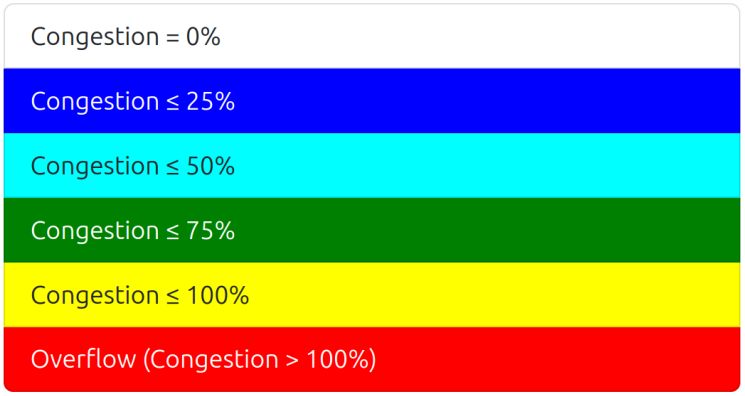


FIGURE 6.2: Congestion Map Legend

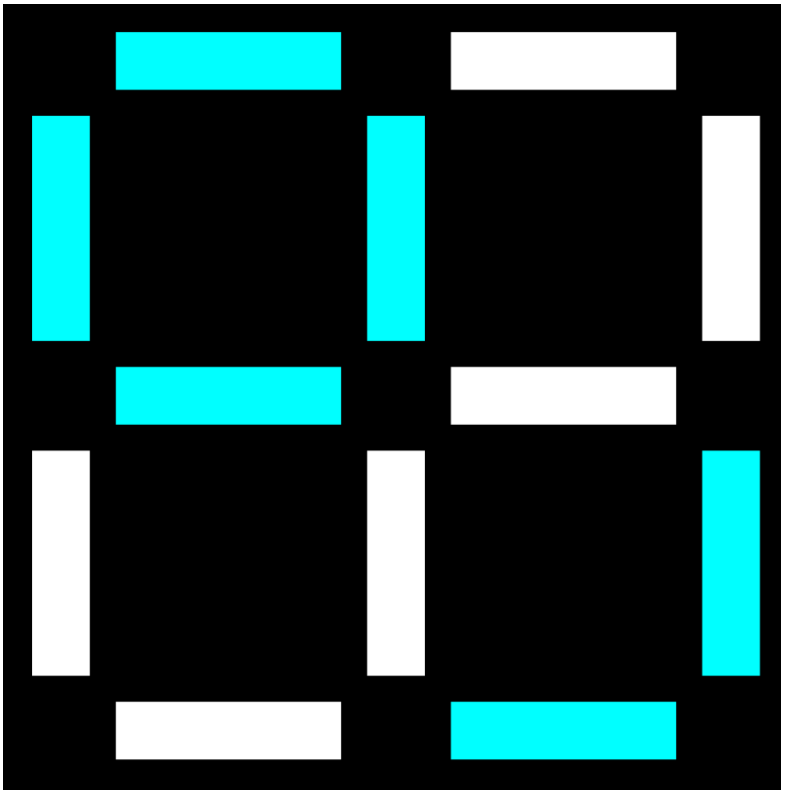


FIGURE 6.3: Simple Example Congestion Map

6.5 Comparison on Performance of the Algorithm Against Competitors

Figure 6.12 provides an in-depth comparison of my algorithm's performance against that of other competitors, who were students that completed the same assignment in NTHU over the years. In terms of overflow and wirelength, my algorithm performed on par with the competitors and even produced the best result for the IBM01 benchmark. However, the primary difference between my solution and the others is that my algorithm is built

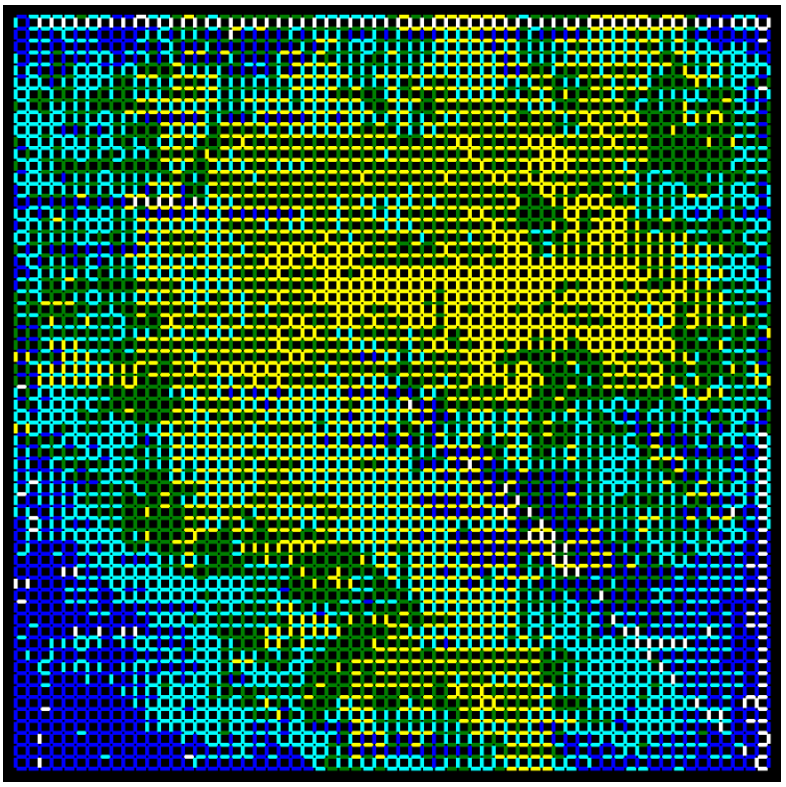


FIGURE 6.4: IBM01 Congestion Map

in Python, while the competitors are built in C++. This means that my algorithm may face a disadvantage in terms of runtime, as Python is known to run slower than C++.

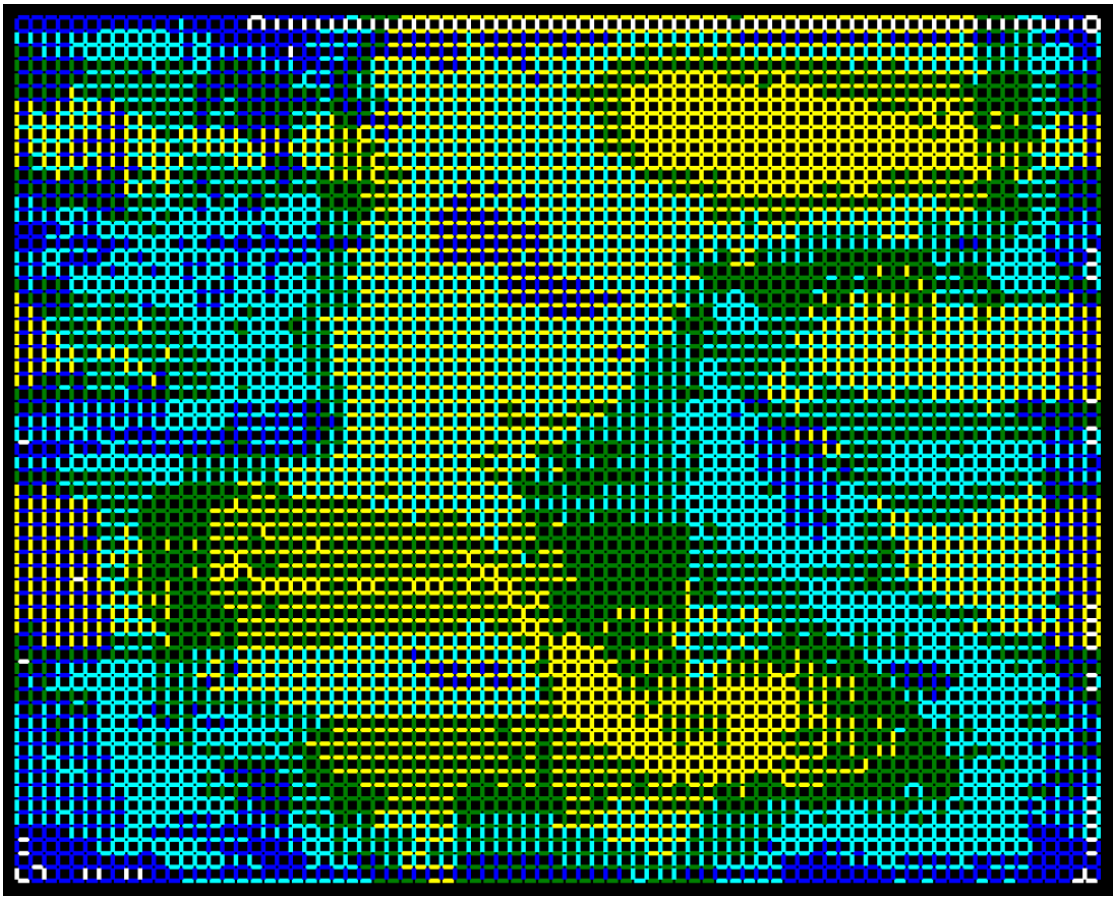


FIGURE 6.5: IBM02 Congestion Map

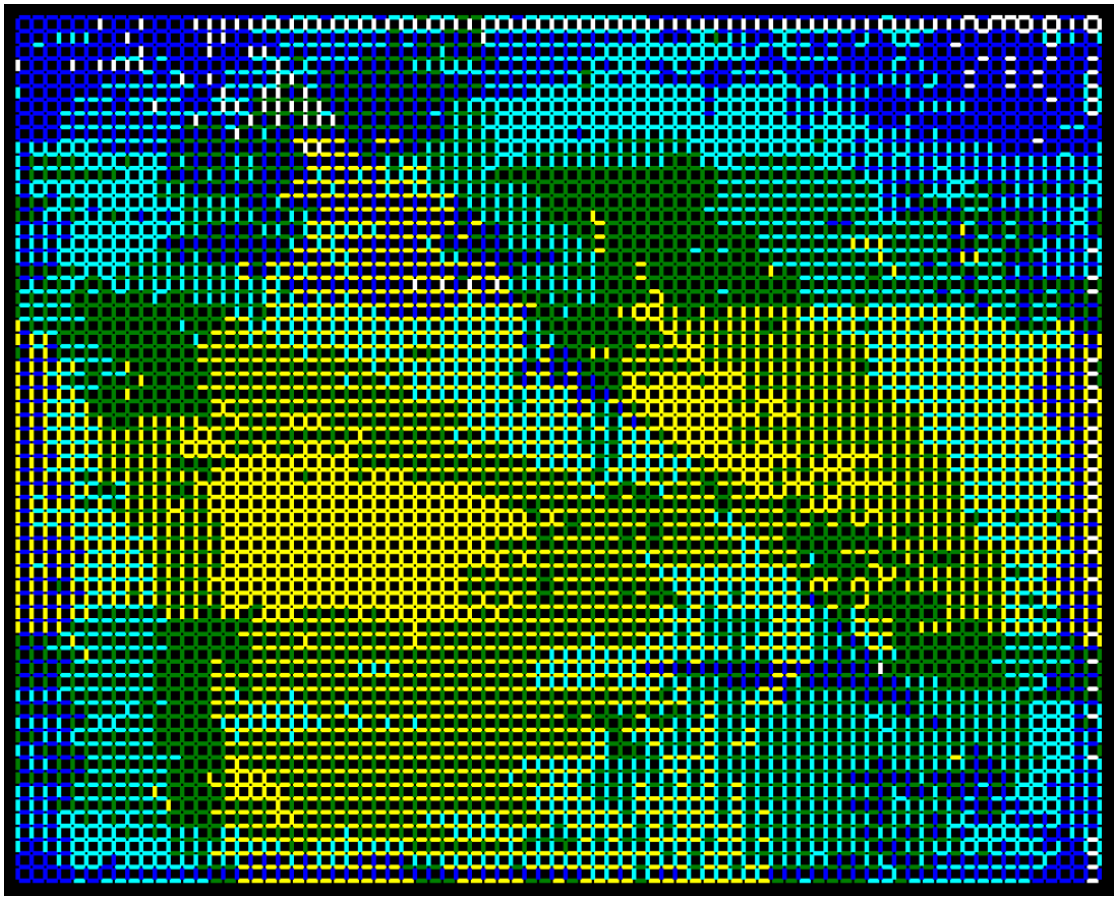


FIGURE 6.6: IBM03 Congestion Map

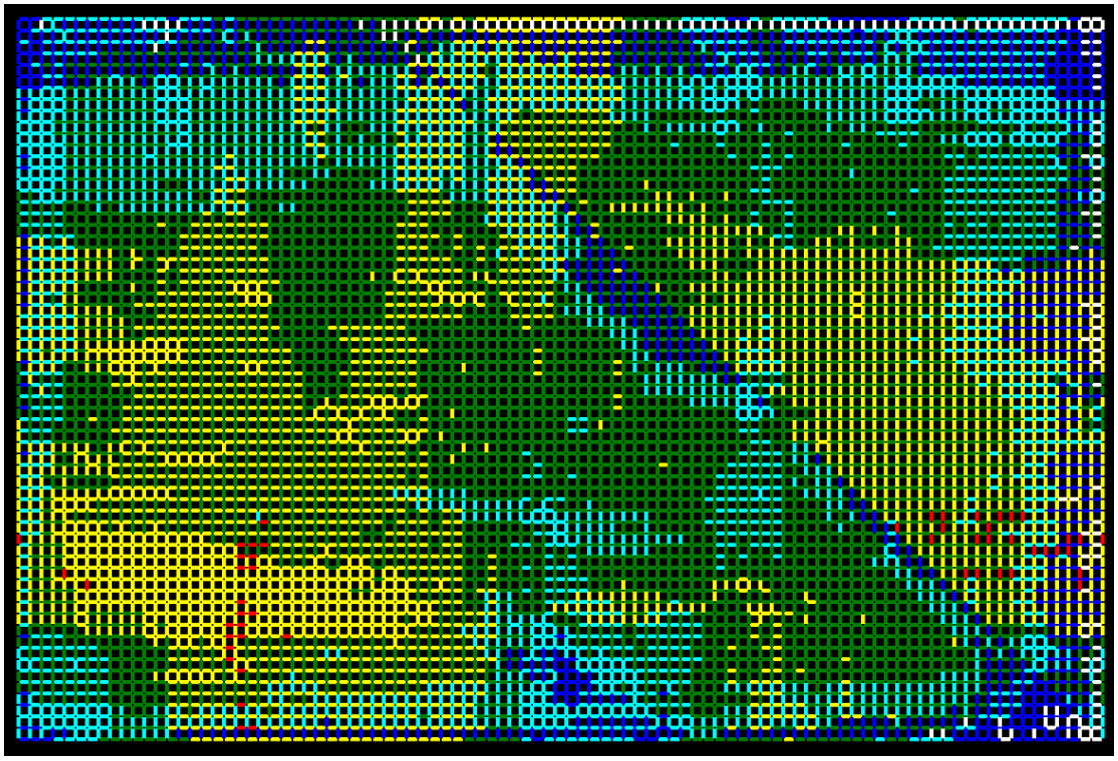


FIGURE 6.7: IBM04 Congestion Map

```
Begin to verify routing result.
Parsing input file..
        Test case:      testcase/ibm03.modified.txt
        Result file:    ibm03.modified_1682845633
Total nets to route=21609
Verifing...
Starting Dumping Congestion Map in Graphical File: testcase/ibm03.modified.txt.png
Drawing Congestion Map in Graphical File: testcase/ibm03.modified.txt.png
End Drawing: testcase/ibm03.modified.txt.png
########################## RESULT ################################
# of unconnected net because of pin: 0
# of unconnected net because of wire: 0
# of net has duplicate wires: 0
# of overflow: 0
max overflow: 0
# of net without overflow: 21609 / 21609
 Wire bend count: 0
  XY  WireLength: 142534
 Via  WireLength: 0
Total WireLength: 142534
Total    Cost:    142534
```

FIGURE 6.8: IBM03 Output Verification

```
Begin to verify routing result.
Parsing input file..
        Test case:      testcase/ibm04.modified.txt
        Result file:    ibm04.modified_1682846961
Total nets to route=27781
Verifing...
Starting Dumping Congestion Map in Graphical File: testcase/ibm04.modified.txt.png
Drawing Congestion Map in Graphical File: testcase/ibm04.modified.txt.png
End Drawing: testcase/ibm04.modified.txt.png
########################## RESULT ################################
# of unconnected net because of pin: 0
# of unconnected net because of wire: 0
# of net has duplicate wires: 0
# of overflow: 86
max overflow: 5
# of net without overflow: 26926 / 27781
 Wire bend count: 0
  XY  WireLength: 160676
 Via  WireLength: 0
Total WireLength: 160676
Total    Cost:    160676
```

FIGURE 6.9: IBM04 Output Verification

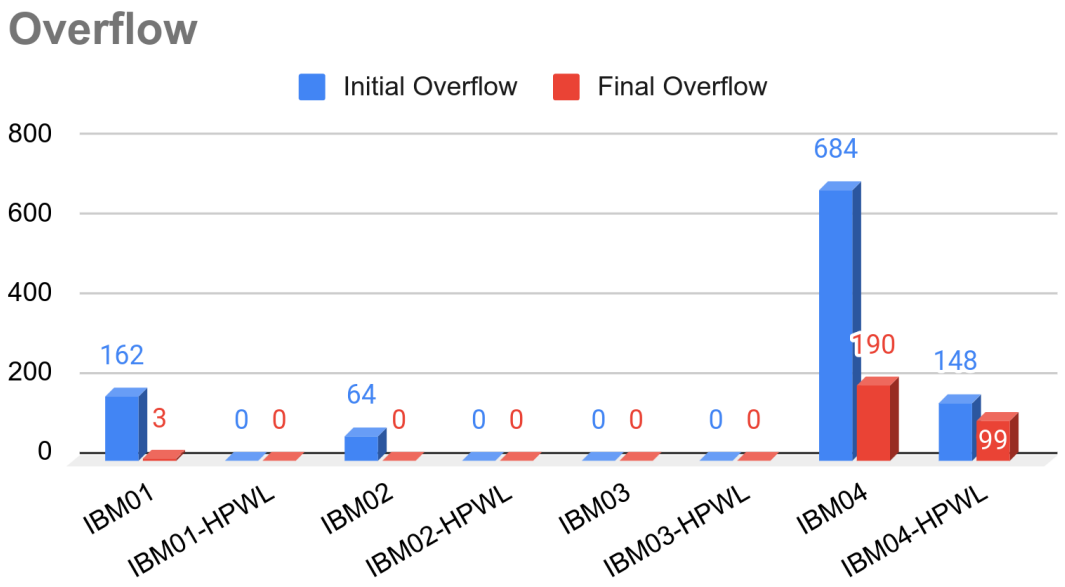


FIGURE 6.10: Overflow Result HPWL Comparison

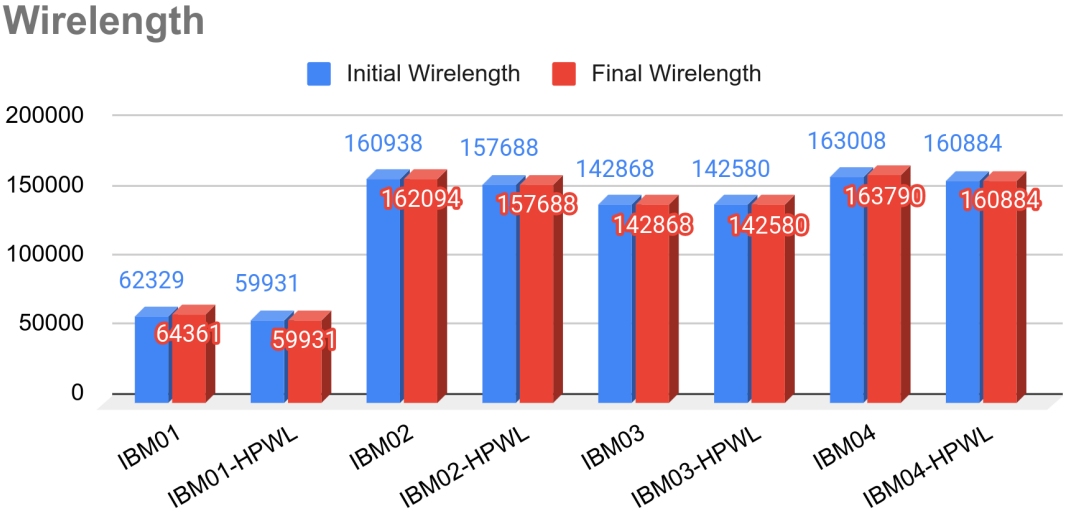


FIGURE 6.11: Wirelength Result HPWL Comparison

Benchmark	IBM01			IBM02			IBM03			IBM04		
Solution	OvF	WL	Time	OvF	WL	Time	OvF	WL	Time	OvF	WL	Time
1st	0	59371	8.89	0	156556	14.41	0	142680	3.57	71	160342	158.51
2nd	0	59519	3.91	0	157394	6.4	0	143000	2.91	63	159136	55.57
3rd	0	63495	6.57	0	166148	8.83	0	143894	13.21	101	162460	122.5
Lee Yellow	0	61271	33	0	161245	113	0	147086	91	285	161728	148
Romulus	0	62529	107.24	0	161486	350.2	0	144522	87.82	222	159408	590
Lapaceyc	0	60391	4.39	0	160066	16.58	0	143594	3.39	119	160888	55
Lei Hsiung	0	60025	1.03	0	157550	3.04	0	142502	3.4	91	160660	12.73
Yi Ming Tan	0	59931	12.98	0	157688	23	0	142580	23.1	99	160884	221.69

FIGURE 6.12: Comparison on Performance of the Algorithm Against Competitors

Chapter 7

Discussion and Conclusions

This chapter provides a comprehensive review of the solution and overall project. A general conclusion will be discussed, along with potential future work that could be undertaken should additional time be available.

7.1 Solution Review

Overall, the implemented solution has been found to be satisfactory, particularly when compared to competing solutions. The project's solution is built using Python, in contrast to the competitors' use of C++. This distinction results in a performance deficit, as C++ generally outperforms Python in terms of speed. Nevertheless, this deficiency is relatively insignificant since runtime is only a tertiary objective. Regarding the two primary objectives, namely total overflow and wirelength, this project's solution has proven to be highly effective. In fact, it even outperformed all competitors for the IBM01 benchmark by achieving the shortest wirelength of them all (shown in Figure 6.12).

Here are several essential techniques that significantly improve the quality and performance of the solution, including:

- Parallelism: Parallelism: By enabling the global router to run multiple instances of the initial routing process, parallelism increases efficiency. If necessary, the router with the best outcome will be chosen for the rip-up and reroute stage (iterative improvement stage).
- Randomization: The routing order of the nets is critical, and different orders can produce different results. Randomizing the net routing order introduces stochasticity, which may generate better-quality output.

- Sorting netlist in order of HPWL: Sorting the netlist in order of HPWL (Half-Perimeter Wirelength) allows larger nets to be routed more effectively around smaller ones that have already been routed. This reduces routing impediments and optimizes the utilization of routing resources. The difference in total overflow and wirelength, in the context of all four benchmarks, between sorting the netlist in HPWL order is compared in the Figure 6.10 and Figure 6.11. and without.

Upon closer examination of the algorithm, we focused on the function responsible for routing two pins. In particular, we investigated two methods: naive breadth-first search and best-first search.

Breadth-first search represents a naive approach since it does not consider the congestion level of the edges between nodes. Instead, it relies on a simple queue data structure to store nodes to be explored next. The algorithm works by expanding outward like a wave, exploring all nodes in four directions (north, east, south, and west) until the destination pin is found. Subsequently, the algorithm backtracks from the destination pin to the starting pin to return the shortest path possible. While this approach ensures the shortest possible connection between two pins, it often results in significant overflow due to its disregard for congestion levels. Running breadth-first search on IBM01 will give you an overflow of 3394 and a wirelength of 56773.

In contrast, the best-first search approach is a more intelligent method for routing two-pin nets. This approach employs a priority queue data structure to store nodes to be visited next, with the priority of each node calculated based on its congestion level. The lower the congestion level, the higher the priority to be selected as the next node. During our investigation, we explored two different priority queue implementations: binary heap and Fibonacci heap. We found that a simple binary heap implementation performs better than the Fibonacci heap implementation in this context. With the binary heap implemented priority queue to store nodes based on their congestion level in place, the algorithm continuously explores the next node until it reaches the destination pin. Subsequently, a straightforward backtracking step is performed, and the path that connects two pins with minimal overflow is returned. Running best-first search on IBM01 will give you 0 overflow and a wirelength of around 60000.

7.2 Project Review

7.2.1 Key to Success for this Project

Despite the challenges posed by the complexity and unfamiliarity of VLSI global routing, the project progressed smoothly overall. This success can be largely attributed to smart decision-making early in the project timeline, specifically front-loading the core elements such as the algorithm and command-line interface in January and February. Additionally, it is worth noting the extensive preparation performed beforehand, including acquiring the necessary skills to tackle a topic of this nature. Consistent self-learning was required to develop an effective global routing algorithm, and this groundwork proved invaluable in enabling a smooth and successful project outcome.

7.2.2 Retrospection

In retrospect, if given a chance to restart this project, I would opt to develop it using C++ instead of Python, especially given its performance-critical nature. Although Python may offer greater ease of use and readability, it falls short of C++ in terms of performance. This is attributed to the fact that C++ is a compiled language that compiles directly into machine code, which is executed directly by the CPU, whereas Python is an interpreted language that requires an interpreter to execute, adding an extra layer of processing that can slow down execution. Python's sluggishness necessitated considerable optimization efforts to speed up its runtime. I would relish the challenge of developing the project with C++ if the opportunity arose, as it would require me to master its fundamentals and concepts, resulting in my becoming an expert in C++ by the end of the project rather than merely being comfortable developing apps in Python.

Regarding the web app, a simple Flask server can still be utilized alongside a GRPC channel to facilitate communication between the C++ routing API and the Flask server. GRPC is an open-source, high-performance remote procedure call (RPC) framework that is platform-agnostic and enables efficient communication between distributed systems. Since it supports multiple programming languages, including C++ and Python, it is an ideal choice for multi-language projects. Incorporating gRPC would provide several benefits, such as faster communication, reduced overhead, and simplified service development and deployment. Furthermore, while it may be beyond the scope of this project, developing the frontend with React could be an advantageous choice. React is a widely used JavaScript library for building user interfaces, which enables the creation of reusable UI components, efficient rendering, and a declarative programming approach,

resulting in faster and more maintainable code. It is a more contemporary alternative to Ajax, which may be considered outdated in 2023, although it remains a viable option.

7.2.3 Problems that arose and how they were handled

Developing this project in Python can be a relatively smooth-sailing process; however, due to its interpreted nature and an additional layer of computation, it may experience slower performance. Consequently, a significant amount of time was dedicated to scrutinizing the codebase and identifying the most resource-intensive sections. In optimizing the code, it was necessary to consider both time and space complexity of the program.

In the first example, the function responsible for routing two-pin nets employs a best-first search algorithm. In this implementation, the algorithm maintains a record of visited nodes, and if the current node has been visited before, the program simply continues processing the next node. The data structure utilized for tracking visited nodes is crucial. Initially, a list data structure was employed for this purpose, with the program checking at every iteration whether the current node already existed in the list. However, the time complexity of searching for an item in a list data structure is $O(n)$, which can become increasingly inefficient as the number of visited nodes grows. To address this issue, a set data structure was adopted instead. Compared to a list, the time complexity of searching for an item in a set is a constant $O(1)$. Consequently, using a set for storing visited nodes represents a superior approach, as its efficiency remains unaffected even as the number of nodes grows.

Another example pertains to the path module, where a section of code within the constructor of the Path class populates the coordinates_list attribute by traversing the nodes in reverse and obtaining the coordinates of each node. The initial approach was to append each node's coordinates to a list data structure, and once the coordinates list was fully populated, to reverse the list. However, reversing a list in Python has a time complexity of $O(n)$, which can become inefficient as the size of the list grows. To address this issue, it was realized that the extra computation and reversing step were unnecessary if the correct data structure was employed to store the coordinates. As such, a deque (double-ended queue) data structure was adopted instead. This data structure allows for appending items to both the left and right of the queue and popping items from either end, with all operations having a time complexity of $O(1)$. By appending each node's coordinates to the left of the deque at every iteration, a reversed order coordinates list is generated without the need for a separate reversing step, thereby avoiding additional computation.

7.2.4 Newly Developed Skills and Applications

7.2.4.1 Unit Testing and Coverage

Writing comprehensive unit tests with high coverage is crucial in any software projects. Writing unit tests is crucial in ensuring the fundamental units of a program are functioning as intended, and to detect any code changes that may impact other parts of the program. Utilizing coverage tools can help achieve sufficient unit test coverage and generate comprehensive reports. This skill is invaluable regardless of the programming language used.

7.2.4.2 Logging

Implementing a logging module for log generation and management is important in software projects. Having a well-structured logging system in place benefits both end-users and developers, as it provides a clear understanding of system activities and events over time. Without logs, the system's inner workings can become a mystery, leading to confusion and ambiguity when issues arise.

7.2.4.3 Documentation

Setting up and writing documentation is an often overlooked yet vital skill in software engineering. Documenting code and writing user guides not only aids others in understanding the project but also serves as a reference for developers who may forget specific details. Neglecting this aspect of the project can lead to confusion and hinder progress in the long run.

7.2.4.4 Algorithms and Data Structures

Heuristic algorithms have been widely used to solve NP-hard problems across various domains. These algorithms allow us to navigate large search spaces while balancing completeness and optimality efficiently. Despite being computationally expensive in the worst-case scenario, heuristic algorithms can significantly reduce search time in practice. Furthermore, with more sophisticated heuristics, these algorithms can become even more powerful in finding near-optimal solutions to complex problems. However, their efficiency and effectiveness are heavily dependent on the chosen data structures, as well as the consideration of time and space complexities during the implementation process. By carefully selecting the appropriate data structures and optimizing the code, significant

improvements in the performance of heuristic algorithms can be achieved. Gaining a clear understanding of data structure algorithmic concepts is crucial, as it enables one to apply the same algorithmic structure and flow when faced with problems of similar structure. This allows for a more efficient problem-solving process, as one can build upon their existing knowledge and experience to tackle new challenges.

7.2.4.5 Multiprocessing and Multithreading

The utilization of multiprocessing and multithreading are essential concepts in software engineering. Multiprocessing is useful when dealing with heavy computational tasks that can be split into smaller, independent processes. It allows multiple processes to run concurrently on different cores, significantly reducing computation time. On the other hand, multithreading is better suited for input/output-bound tasks that require waiting for external resources such as user input or file access. Multithreading enables multiple threads to run concurrently, allowing for better resource utilization and faster response times. Understanding when to use multiprocessing versus multithreading is crucial as a software engineering skill. It is also important to consider the limitations and potential bottlenecks of each method when implementing them into a project. Proper implementation of multiprocessing and multithreading can lead to significant performance gains and more efficient resource utilization. Conversely, a poorly implemented multiprocessing or multithreading solution can lead to unexpected errors and decreased performance. Therefore, knowing these concepts and their applications is essential for effective software engineering.

7.3 Conclusion

7.3.1 Primary Conclusions

7.3.1.1 VLSI Global Routing is Complex

VLSI global routing is a complex topic that involves routing signals across an integrated circuit that contains millions or even billions of transistors, all packed into an incredibly small area. These components are minuscule, often measured in nanometers, and as chipmakers continue to push for smaller and smaller transistors, the challenges of VLSI global routing only become more daunting. In fact, in 2023, chipmakers are now working with transistors as small as 3nm, allowing more transistors to be packed into a single chip, making it more powerful. As a result, VLSI global routing requires a deep understanding of computer science, electrical engineering, and mathematics, as well as a variety of

specialized tools and techniques to tackle the unique challenges posed by these incredibly small components.

7.3.1.2 Heuristic Algorithms are used to find Near-Optimal Solutions

In conclusion, heuristic algorithms are a practical approach for solving NP-hard problems where finding an exact solution is impractical. These algorithms do not guarantee optimal solutions but instead, aim to find near-optimal solutions in a reasonable amount of time. By utilizing techniques such as greedy algorithms, local search, and metaheuristics, heuristic algorithms can efficiently search large solution spaces and find solutions close to the optimal one. It is essential to balance the exploration and exploitation phases of the algorithm to avoid getting stuck in local optima. With careful consideration of the problem's constraints, objectives, and available computational resources, heuristic algorithms can be a powerful tool for solving complex optimization problems.

7.3.1.3 Sorting Netlist in Ascending Order of HPWL Yields Better Results

Sorting the netlist in the order of its half-perimeter wirelength (HPWL) has proven to be an effective technique in improving the overall performance of global routing algorithms. This is because the HPWL metric gives a measure of the wirelength required to connect two pins of a net, and arranging the netlist in increasing order of HPWL allows for the smaller nets to be routed first, potentially reducing congestion and improving routing efficiency.

7.3.2 Secondary Conclusions

7.3.2.1 Time Management is #1 Key to Success

Time management is crucial for success in a challenging global routing project, especially for someone without prior knowledge of VLSI. With the complexity of the project and the steep learning curve, it is essential to set clear goals and timelines and prioritize tasks to make the most efficient use of the available time. Poor time management can lead to missed deadlines, a poorly executed project, and added stress. On the other hand, good time management skills can lead to a better understanding of the project, improved productivity, and a successful outcome. Managing time effectively is a valuable skill in any project and is particularly important when facing challenges like those encountered in global routing.

7.3.2.2 Python is Slow

While Python is a versatile language and offers many advantages for software development, it may not be the best choice for computationally intensive tasks such as this global routing project. While Python can be used for global routing, it may require additional optimization and careful consideration of the algorithms and data structures used to ensure the program runs efficiently. As an interpreted language, Python requires an additional layer of computation, which can significantly slow down the program's execution time. As a result, a lot of time and effort may need to be invested in analyzing the codebase and identifying the most expensive parts of the program. To optimize the program's performance, developers must keep in mind the time and space complexity of the algorithms and use efficient data structures to store and manipulate the data.

7.4 Future Work

7.4.1 Load Balancing

Currently, all requests are being handled by a single Flask server in the implemented system. Although a user can submit several routing jobs in parallel, it does not necessarily mean they will run without overhead or performance issues. The server may become overloaded with even a single job, let alone multiple jobs. Consequently, sending several job requests to the server simultaneously will overload it. To resolve this issue, a load-balancing architecture is required. By creating a pool of server instances that are idle and awaiting job assignments, the load balancer can distribute job requests evenly among the instances. This way, when multiple users send multiple job requests simultaneously, the load balancer can manage all the loads and prevent any single instance from being overloaded. Figure 7.1 depicts an example load-balancing architecture.

7.4.2 3D Inputs and Multi-Pin Nets Support

The current implementation of the algorithm only supports a single-layer 2D layout and is limited to 2-pin nets. However, this falls short of the requirements in the chip-making industry, where layouts typically consist of multiple layers in a 3D environment and involve multi-pin nets. The primary objective is to connect all pins of a net with minimal wirelength and overflow, subject to various constraints that were not considered in this project. To address these challenges, one approach is to use the rectilinear Steiner minimum tree (RSMT), as described in Section 2.2.5. Given more time, the abovementioned limitations can be overcome and fully supported.

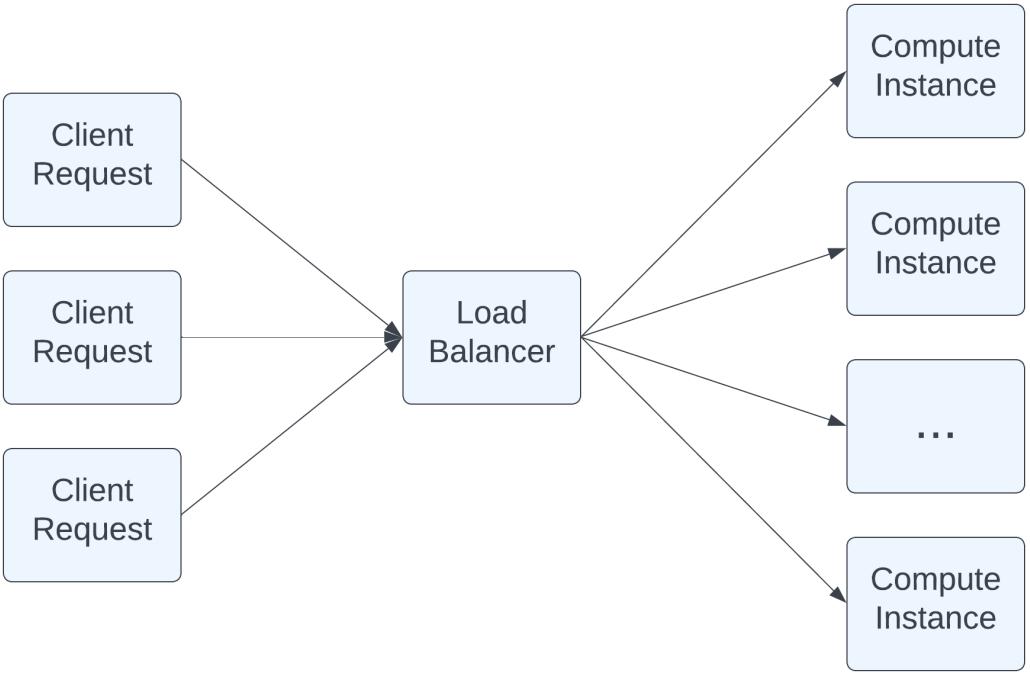


FIGURE 7.1: Example Load-Balancing Architecture

7.4.3 Improved Congestion Data Visualizer

The current implementation of the congestion data visualizer can only display the congestion level of the circuit elements through a static visualization. Although it can be moved, zoomed in and out, it lacks interactive features. Given more time, a side panel could be developed to show all nets in the netlist. The individual nets can then be clickable links, which, when selected, should enable the visualizer to display whether the net is causing overflow and provide the length of the net. Additionally, the net should be highlighted within the congestion visualization plot, allowing for a more interactive and informative experience.

Bibliography

[1] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *1.8 Common EDA Terminology.* Springer, 2022, p. 26–29.

[2] C. Patel, "Advanced vlsi design." [Online]. Available: https://redirect.cs.umbc.edu/courses/graduate/CMPE641/Fall08/cpatel2/slides/lect04_LEF.pdf

[3] Cadence, 2017. [Online]. Available: http://coriolis.lip6.fr/doc/lefdef/lefdefref/lefdefref.pdf

[4] "Vlsi design cycle," Apr 2022. [Online]. Available: https://www.geeksforgeeks.org/vlsi-design-cycle/

[5] vlsi4freshers, "Vlsi design flow," Jan 2020. [Online]. Available: https://www.vlsi4freshers.com/2020/01/vlsi-design-flow.html

[6] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *1.6 Algorithms and Complexity.* Springer, 2022, p. 20–23.

[7] G. L. McDowell, *Big O.* CareerCup, LLC, 2021, p. 38–59.

[8] W. T. Tutte and W. T. Tutte, *Graph theory.* Cambridge university press, 2001, vol. 21.

[9] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *1.7 Graph Theory Terminology.* Springer, 2022, p. 24–25.

[10] B. Y. Wu and K.-M. Chao, *Spanning trees and optimization problems.* Chapman and Hall/CRC, 2004.

[11] R. Venkateswaran and P. Mazumder, "Routing algorithms for vlsi design," Ph.D. dissertation, University of Michigan.

[12] I. Shirakawa and S. Futagami, "A rerouting scheme for single-layer printed wiring boards," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 2, pp. 267–271, 1983.

[13] A. S. OLIVEIRA, "Initial detailed routing algorithms," Ph.D. dissertation, 2021.

[14] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 346–365, 1961.

[15] A. Teman, "Lecture 9: Routing," Jan 2019.

[16] E. Moore, *The Shortest Path Through a Maze*, ser. Bell Telephone System. Technical publications. monograph. Bell Telephone System., 1959. [Online]. Available: https://books.google.ie/books?id=IVZBHAAACAAJ

[17] T.-C. Wang, "Routing," 2003.

[18] J. Soukup, "Fast maze router," *15th Design Automation Conference*, pp. 100–102, 1978.

[19] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[20] A. Javaid, "Understanding dijkstra algorithm," *SSRN Electronic Journal*, 01 2013.

[21] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, pp. 100–107, 1968.

[22] G. Clow, "A global routing algorithm for general cells," in *21st Design Automation Conference Proceedings*, 1984, pp. 45–51.

[23] M. Cho, K. Lu, K. Yuan, and D. Z. Pan, "Boxrouter 2.0: architecture and implementation of a hybrid and robust global router," in *2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 503–508.

[24] M. D. Moffitt, "Maizerouter: Engineering an effective global router," in *2008 Asia and South Pacific Design Automation Conference*, 2008, pp. 226–231.

[25] Y.-J. Chang, Y.-T. Lee, and T.-C. Wang, "Nthu-route 2.0: A fast and stable global router," in *2008 IEEE/ACM International Conference on Computer-Aided Design*, 2008, pp. 338–343.

[26] L. Hsiung, "Cs6135 hw5 global routing report," Jan 2021. [Online]. Available: https://github.com/twweeb/VLSI-Physical-Design-Automation/blob/master/HW5%20-%20Global%20Routing/submit/CS6135_HW5_109062509_report.pdf

[27] "Snapdragon 8 gen 2 mobile platform." [Online]. Available: https://www.qualcomm.com/products/application/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-8-gen-2-mobile-platform

[28] "Welcome to flask." [Online]. Available: https://flask.palletsprojects.com/en/2.2.x/

[29] "Visualization with python." [Online]. Available: https://matplotlib.org/

[30] "Firebase authentication." [Online]. Available: https://firebase.google.com/docs/auth

[31] "Cloud storage for firebase — firebase storage." [Online]. Available: https://firebase.google.com/docs/storage

[32] J. T. Mark Otto, "Get started with bootstrap." [Online]. Available: https://getbootstrap.com/docs/5.2/getting-started/introduction/

[33] Mijacobs, "What is scrum? - azure devops." [Online]. Available: https://learn.microsoft.com/en-us/devops/plan/what-is-scrum

[34] "The collaborative interface design tool." [Online]. Available: https://www.figma.com/