ESE 556 - VLSI Physical and Logic Design Automation

Timberwolf Algorithm

Submitted by

Thomas Kral (111558962)

Felix Chimbo Cungachi (113470260)

Karan Rajendra (115375145)

1. Abstract

Within the realm of Very Large Scale Integration (VLSI) design, the pursuit of enhancing integrated circuit yield through layout modification has become imperative, especially as routing options become increasingly constrained due to their designs. While numerous algorithms already exist for optimizing various aspects of VLSI floorplanning such as area, wirelength, power, and temperature, there remains a pressing need to improve the computation time it takes to acquire the best routes. This report introduces our implementation and experimentation with the Timberwolf Algorithm tailored specifically to mitigate floorplanning issues in VLSI layout design with the use of creations of various classes and structures in C++ that will aid in hopefully achieving this goal.

2. Introduction

When it comes to the design of VLSI circuits, the process is complex where it requires several interconnected stages, one of which requires having the circuit minimize its area and wirelength. The layout design process is an important part of the design where this minimization occurs as it assigns a specific location for the elements that are to be used on the circuit along with a specific wire connection that is needed among these elements Due to this complexity of the process, it can be described in two phases, where the first is called placement and the second is routing. For the placement phase, it is essentially an initial placement of all the circuit elements that are needed; this can either place an element in the most ideal location or it can do the opposite. The routing stage is all the connections needed for the elements where electric wire connections are made.

The Timberwolf algorithm serves as a powerful heuristic optimization tool tailored to address the intricate challenges of circuit design. Within VLSI implementations, where the arrangement of components and interconnections plays a pivotal role in circuit performance, TimberWolf excels in two crucial phases: placement and routing. During the placement phase, it facilitates the strategic positioning of circuit elements on the chip, optimizing the layout to meet specified design criteria. Subsequently, in the routing stage, Timberwolf orchestrates the establishment of wiring paths, ensuring efficient interconnections while minimizing signal delay and power dissipation.

The following report will tackle our approach to the Timberwolf algorithm where it will involve the creation of several classes and structures in C++ that will be used to simulate the process of the pseudo code that has been given to us for reference. We will tackle the various methods to be able to test the output of the given input files where it will then create a floorplan that is most optimal for all the electrical components followed by the ideal wire length for a faster connection speed between the components. These results will then be presented towards the end of the report to verify if our implementation of the algorithm is functional.

3. Problem Statement

Using a set of given benchmarks, the objective is to take the set of nodes and map it in an FPGA environment so that it is a feasible design. Feasibility is determined by the ability to route the entire circuit.

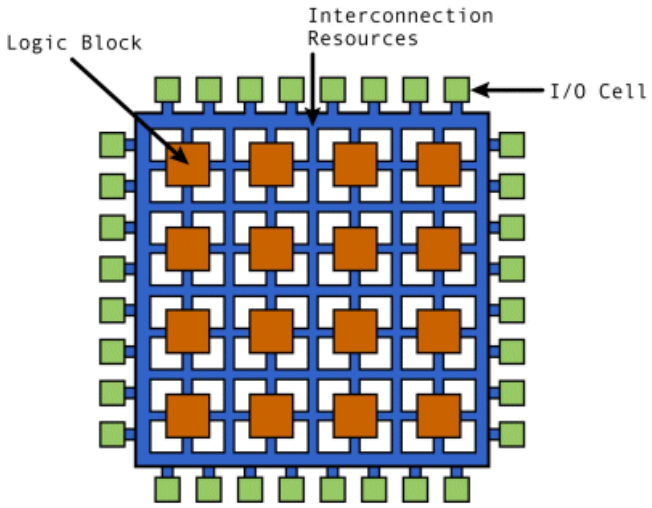


Figure 1. Diagram of FPGA. [1]

Figure 1 shows the setup of FPGA circuitry. There are pre-placed route paths that the circuit has the ability to take, however these paths have a constrained number of wires within that can be used. With this knowledge we'll need to map the gates and pins from the benchmarks to appropriate places, then determine whether it's feasible without actually routing the design.

The methodology to be used will be the Timberwolf algorithm, which comes with obstacles within itself to find parameters appropriate for simulated annealing, cost function weights, and probabilities of our perturb function, which will be further explained in section 5.

The main goals of our program are to minimize the size of the FPGA, minimize interconnect length, decrease the delay of critical nets, and most importantly find a routable solution.

4. Related Work

As previously mentioned there are many existing algorithms that are for optimizing various aspects of VLSI floorplanning, one of which is the most well developed placement methods available which is the simulated annealing (SA)[9]. This technique is used in many of the phases of the VLSI physical design as it is used in placement as an interactive improvement algorithm. However there is a long run time resulting in the results calculations not being ideal. AS with other algorithms these are then revisited to have a better improvement in their calculations such as K. Shahookar and P. Mazumder as they are able to automate the entire layout process via the use of standard cell design styles for efficient software packages for automated placement and routing.

Another in particular involves the derivation of bounds of the routing for a given placement by Arunshankar Venkataraman and Israel Koren[10]. This is to improve product yields as it has been a problem facing the semiconductor manufacturing industry for years as it takes a considerable amount of runtime and effort to be able to get the following calculations. Based on their results they were able to calculate/ predict the product yield during the placement phase in a short amount of time resulting in there being a faster calculation of the process. This is further expanded via Rajnish K. Prasad and Israel Koren

[11] where they discuss the effect of placement on yield for standard cell design routing as it has been developed. They further discuss the improvement on how it is able to be enhanced if they manage to bypass the predetermined placement as that is what takes a huge chunk of run time.

Although the majority is software implementations of the algorithm there have also been hardware implementations that have looked towards data and structures that are to be implemented in large grids of executional elements on FPGAs[12]. As discussed by A. Khan and S.M. Sait, they have proposed two fuzzy integration functions that have been applied to the VLSI cell placement problems resulting in them receiving better performance from their simulations.

5. Proposed Solution

The overview of our solution will be to use a simulated annealing with our own perturb function very consistent with the pseudo code provided in the lecture slides [4]. Figure 2. shows the pseudo code presented in class.

Simulated Annealing Algorithm

```
Algorithm SIMULATED-ANNEALING
begin
    temp = INIT-TEMP;
    place = INIT-PLACEMENT;
    while (temp > FINAL-TEMP) do
        while (inner_loop_criterion = FALSE) do
            new_place = PERTURB(place);
            ΔC = COST(new_place) - COST(place);
            if (ΔC < 0) then
                place = new_place;
            else if (RANDOM(0, 1) > e^(ΔC/T)) then
                place = new_place;
        temp = SCHEDULE(temp);
end.
```

Figure 2. [5]

This still leaves the issues of what to use for the parameters such as temperature, cooling factor, cost function weights, and perturb probabilities. The structure of our program can be seen below in the flowchart in Figure 3.

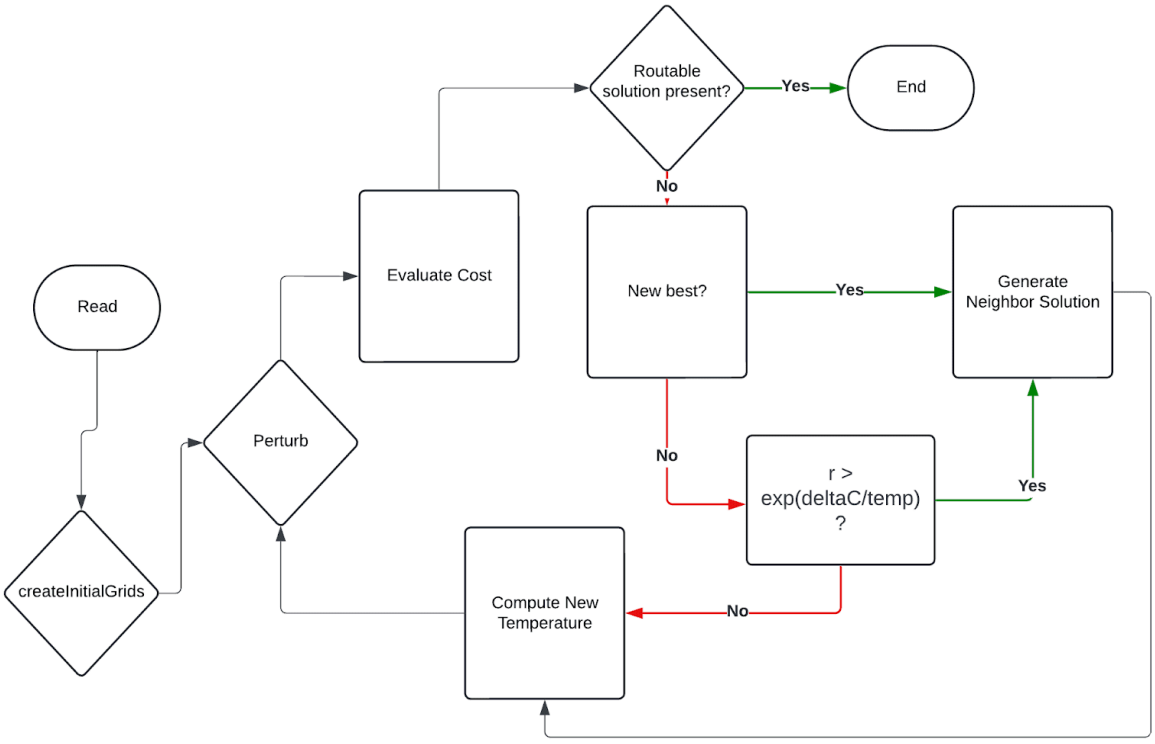


Figure 3. Program Flowchart

5.1 Temperature

An issue faced before even beginning to write the simulated annealing function is what temperature value to begin with. There are several ways to choose a temperature, which will be compared for effectiveness. The first way is described by Ameur [2], which involves mathematically testing various temperatures on a set, then finding an acceptability probability and using the temperature with an exceptional acceptability probability. Another way is suggested by Johnson et. al. [3]. The final way we planned to try was to find an optimal choice by trial and error.

The first way mentioned by Ameur [2] is to iteratively change the initial temperature value until an acceptance probability is above a predetermined threshold, the example used in the paper is 0.8. The formula for acceptance probability is given by:

$$\chi(T) = \frac{\sum_{t \text{ positive}} \pi_{\min_t} \frac{1}{|N(\min_t)|} \exp\left(-\frac{\delta_t}{T}\right)}{\sum_{t \text{ positive}} \pi_{\min_t} \frac{1}{|N(\min_t)|}}. \quad (1)$$

Ameur writes an approximate estimation as follows:

$$\hat{\chi}(T) = \frac{\sum_{t \in S} \pi_{\min_t} \frac{1}{|N(\min_t)|} \exp\left(-\frac{\delta_t}{T}\right)}{\sum_{t \in S} \pi_{\min_t} \frac{1}{|N(\min_t)|}}$$
$$= \frac{\sum_{t \in S} \exp\left(-\frac{E_{\max_t}}{T}\right)}{\sum_{t \in S} \exp\left(-\frac{E_{\min_t}}{T}\right)}. \quad (2)$$

To rephrase what is written in Ameur's paper, the summation of t contained in a set S refers to the summation of positive transitions available between an element and its neighbors. My method to implement this into our program was to treat the initial population as a starting element, and sample some changes to it, such as crossover and mutation, since selection would not represent any transition. Then the summation of all cost improvements made in our one sample iteration would provide us with our summation variable. It should be noted that the transitions were respected if they were less than zero. For some extra context to the presented equations, "πmint * (1/|N(mint)|) represents the probability to generate a transition t when the
energy states are distributed in conformance with the stationary distribution." To summarize this methodology produces an initial temperature depending on the cost function and desired acceptability probability.

After these step we can represent how the value of the temperature will be iterated with the following equation:

$$T_{n+1} = T_n \left(\frac{\ln(\hat{\chi}(T_n))}{\ln(\chi_0)}\right)^{\frac{1}{p}} \quad (3)$$

The estimation of the acceptability probability of our previous temperature is compared to our desired acceptability probability and the ratio is multiplied by the previous T. This process iterates until a T with a desired acceptance probability is found.

```
double generateInitialTemp(vector<Result> init, double prob, float const w1, float const w2,
map<string, Net> const nets, bool& routable, int wireConstraint) {
        double emax = 0., emin = 0.;
        double esum = 0;
        vector<double> es;
        for (Result r : init) {
                double e = 0.;
                while (e <= 0.) { //get an positive transition
                        int ra = rand() % 3;
```

```
                        int ri = rand() % init.size();
                        if (ra <= 1) { //select
                                e = r.cost - init.at(ri).cost;
                        }
                        else if (ra == 2) {
                                //crossover
                        }
                        else {
                                Grid copy = r.g;
                                int rx = rand() % copy.getGridX(); //create initial grid x param;
                                int ry = rand() % copy.getGridY();//create initial grid y param;
                                copy.mutation(rx, ry);
                                bool route = false;
                                vector<Bounds> b;
                                float cost = copy.calcCost(w1, w2, nets, routable, wireConstraint, b);
                                Result n(copy, cost, route, b);
                                e = n.cost - r.cost;
                        }
                }
                es.push_back(e);
                esum += e;
        }
        auto emaxit = max_element(es.begin(), es.end());
        auto eminit = min_element(es.begin(), es.end());
        emax = *emaxit;
        emin = *eminit;
        double t = emax;
        double xo = 0.8;
        double x = 0.;
        while (x < xo) {
                x = exp(-(emax / t)) / exp(-(emin / t));
                t = t * pow((log(x) / log(xo)), (1. / prob));
        }
        return t;
}

double schedule(double temp, double initialTemp) {
        double percentComplete = (initialTemp - temp) / initialTemp;
        if (percentComplete < 0.1 || percentComplete > 0.92) {
                return 0.95 * temp;
        }
        else return 0.8 * temp;
}
```

Figure 4. generateIntitialTemp() and schedule()

Here is our attempt at an implementation of the proposed method by Ameur [2]. The goal is to take a Result vector as an input, iterate through each grid, and apply selection to find the most fit members of the population, then randomly apply either crossover, swap, or move. This get repeated for each grid until a positive change in cost is found, and is then done k-1 more times depending on what the desired population size is. The positive transitions are all recorded to the vector 'es.' This function has a time complexity of O(nxk) where n is the number of Results objects being evaluated, x is the variable amount of times the perturb sub function has to run until a positive transition is found, and k is the constant of executing the perturb sub function.This gives enough information to then apply it in equation (2) and (3). The actually execution of the formula is found here:

```
auto emaxit = max_element(es.begin(), es.end());
        auto eminit = min_element(es.begin(), es.end());
        emax = *emaxit;
        emin = *eminit;
        double t = emax;
        double xo = 0.8;
        double x = 0.;
        while (x < xo) {
                x = exp(-(emax / t)) / exp(-(emin / t));
                t = t * pow((log(x) / log(xo)), (1. / prob));
        }
```

Figure 5. Snippet from Figure 4. showing calculations.

From our vector we can find min and max energy value, then setting a desired acceptance probability, 'xo,' to 0.8, we iteratively apply the equation until temperature is considered acceptable. Figure 4. also shows the cooling schedule method used to iterate through the simulated annealing function.

When looking at the code in the appendix, it may be noticed that our generateInitialTemp method is commented out, this is because the method takes a long time to run, and we didn't find it necessary to conduct with every time we ran the code since the calculation is very similar every time it's run. Only when we made a change to the cost function, or started a new benchmark.

5.2 Cooling Factor and Cost Weights

The cooling factor and cost weights are to be found entirely experimentally. We predict that weighing the overlap cost higher than the net length cost will allow our algorithm to converge to a solution that is routable quicker since routability directly depends on this. We also found it important to rerun the generateInitialTemp method to find a good starting point with any change to how the cost function is computed

After implementing an initial temperature calculation, we noticed that the temperature drops much faster than the population improves, so we tweaked the cooling schedule so temperature takes 0.95 * t at the

beginning and end of the simulated annealing, and 0.9 * t otherwise as opposed to what shown in figure 4.

After some time of experimentation we saw in our results for ibm02 that netlength would not decrease as much as desired, and this was due to the fact that the overlap cost had the highest impact on cost when all weights were equal to 1. We needed to not only make net length cost equal to the overlap cost, but make the net length cost significantly greater than the overlap cost, as in theory decreasing net length should result in less overlap.

Another factor is that with an initial random configuration, there is so much overlap that trying to optimize on overlap is nearly impossible, since the overlap will be unavoidable if net length is barely taken into consideration.

5.3 Classes

Before the simulated annealing can be applied, we needed to implement classes and structs so that our input data could be organized. The structs used were Net, Bounds, Coords, and Result. The classes used were Node, square, utilGrid, and Grid.

First we'll explain the Nodes and Nets, and then Bounds and Coords since they're supplemental. First is the Node class what's code can be seen below:

```
class Node {
private:
    std::string name;
    std::vector<Net*> nets; //requires forward declaration
    int xcoord, ycoord;
    int weight;

public:
    Node();
    Node(std::string name, std::vector<Net*> nets, int xcoord = 0, int ycoord = 0);
    ~Node();
    int getX() const { return xcoord; }
    int getY() const { return ycoord; }
    std::vector<Net*> getNets() const;
    void addNet(Net* net);
    void removeNet(Net* net);
    std::string getName() const;
    void setCoords(int x, int y);
    const bool isTerminal();
    int getWeight();
    void setWeight(int w);
};
```

Figure 6.

As can be seen, each node has its name, its x and y coordinates for placement, its weight, and a list of Net objects that it is a part of. The public methods are all standard set and get methods, besides isTerminal(). isTerminal() returns whether the name of the node starts with a 'p.' Since every node has coordinate parameters, you may ask why is there a need for a Coords struct? The point of the Coords struct is that later will be able to keep a list of coordinates in our grid representation that are empty. Below is the Coords struct:

```
struct Coords {
    int x, y;
    Coords(int x, int y) : x(x), y(y) {}
    bool operator==(const Coords& other) const {//ChatGPT
        return x == other.x && y == other.y;
    }
```

Figure 7.

The struct is pretty straight forward as it's only meant to be placed into a vecotor, so all empty nodes can be kept track of. As seen by the comment, the operation overload is ChatGPT generated. The way this was generated was by asking the bot: 'make an overloaded == operator for …' where the struct was pasted afterwards.

The Net struct is shown below:

```
struct Net {
    Net() = default;
    std::string name;
    std::vector<Node*> Nodes;
    int weight;
    bool isCritical;
    Net(const std::string& name) : name(name) {}

};
```

Figure 8.

The net struct contains a name, a list of nodes it contains, a weight, and whether it's critical. The supplemental struct to the Net is 'Bounds,' as shown below:

```
struct Bounds {
    int x1, x2, y1, y2;
    const Net* net;
};
```

Figure 9.

The purpose of this is to keep track of the bounding box of each Net in the grid representation. The reason this is its own struct and not part of the Net struct is that there will be several populations derived from the same node and net list, so if a change is made to a nets bounds then it will be saved to all populations. Putting these bounds into a small struct reserves memory opposed to saving a separated net list to each population.

This brings us to how the populations will be saved. The Result struct:

```
struct Result {
    Grid g;
    float cost;
    bool routable;
    vector<Bounds> bounds;
    Result(Grid g, float cost, bool routable, vector<Bounds> bounds) : g(g), cost(cost),
routable(routable), bounds(bounds) {}
    Result() : cost(0), routable(false) {}
};
```

A vector<Result> will be a representation of a population. This way we'll have k amount of saved grids, with their cost, whether they're routable, and the bounds of each net.

This leads to the need to explain the Grid, utilGrid, and square classes. The first class to be explained is the square class, since is the basic piece of the grid:

```
enum class squareType {
    Terminal, //pin e.g. p123 ***in this case the square can hold 2 pins e.g. [p1, p2]
    Node,//gate e.g. a123
    Routing // space for wires
};

class square {
private:
    squareType type;
    const Node* node;
```

```
    int wires; //count of wires
public:
    square();
    square(squareType type, const Node* n = nullptr, int wires = 0);
    void setType(squareType st);
    squareType getType();
    void incWires(); //increment wire count if wire type
    void decWires(); //dec if wire type
    void setNode(const Node* n); //set node if terminal or gate
    const Node* getNode();
    bool isEmpty();

    friend class Node;
};
```

Figure 10. Square class, and squareType class.

Every square in a grid consists of a type. The type can either be Terminal, Node, or Routing for what they'll be containing. If they are of Terminal or Node type, then the Node* member will be utilized to point to what node is placed there. For possibly the next problem, if it is of type Routing, then there will be a count of how many wires are run through that space already, but will have to be expanded to account for switch boxes. Next we can take a look at the container of the square class objects, the Grid and utilGrid:

```
class Grid {
private:
    vector<vector<square>> grid;
    utilGrid ug;
    vector<Coords> enodes; //bounds of empty nodes in grid
    vector<Coords> eterms;
public:
    Grid();
    Grid(const std::map<std::string, Node>& nodes); // Updated constructor
    void write(int x, int y, square s);
    void swap(int x1, int y1, int x2, int y2);
    void move(int x1, int y1, int x2, int y2);
    void mutation(int x1, int y1);
    void initialPlacement(const std::map<std::string, Node>& nodes);
    square getSquare(int x, int y); //get square with coordinates
    float calcCost(float const w1, float const w2, map<string, Net> const nets, bool& routable, int
wireConstraint, vector<Bounds>& bounded) const;
    int getGridX();
```

```
    int getGridY();
    // New methods for crossover support
    void placeNode(int x, int y, const Node* node);
    bool isNodePlaced(const Node* node) const;

    friend class square;
    friend class utilGrid;
};
```

Figure 11.

We won't explain all of the public methods yet, and will be covered in their own section. The private members of the class are the grid itself, which is a 2D array made with vectors, a utilGrid, and a list of empty gates and terminals. To explain further what these grids are, we've made visualizations using pyplot:

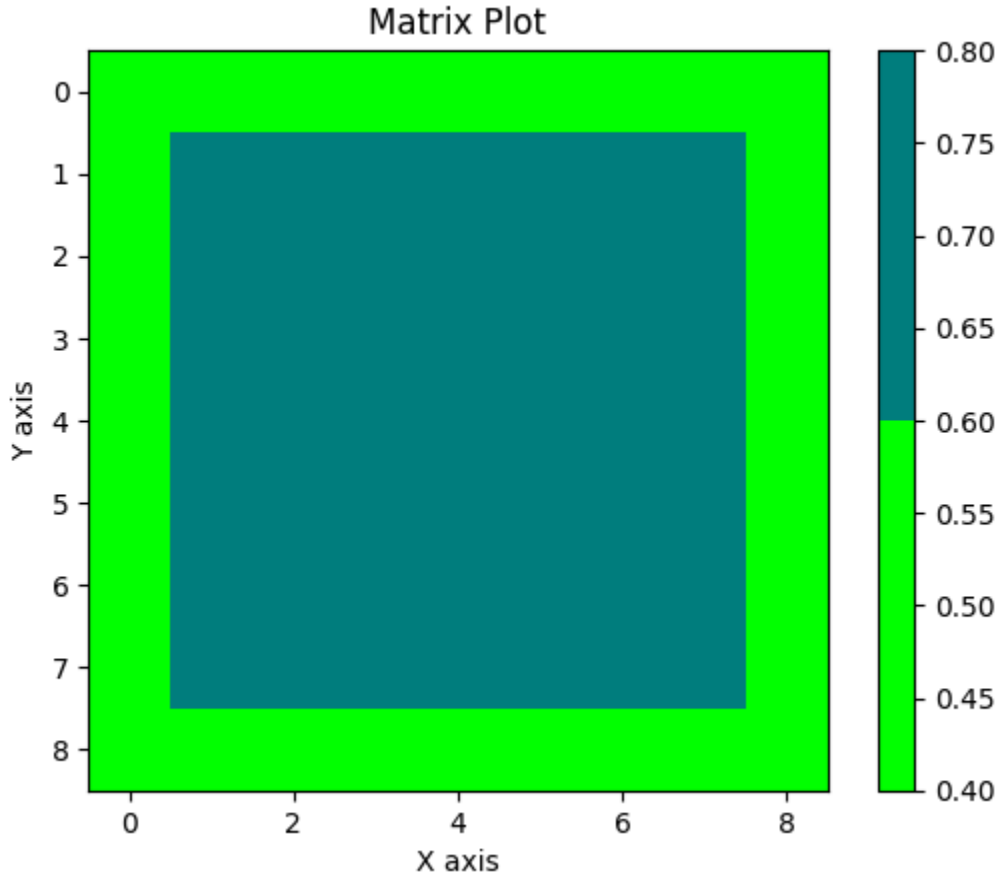


Figure 12. Grid representation.

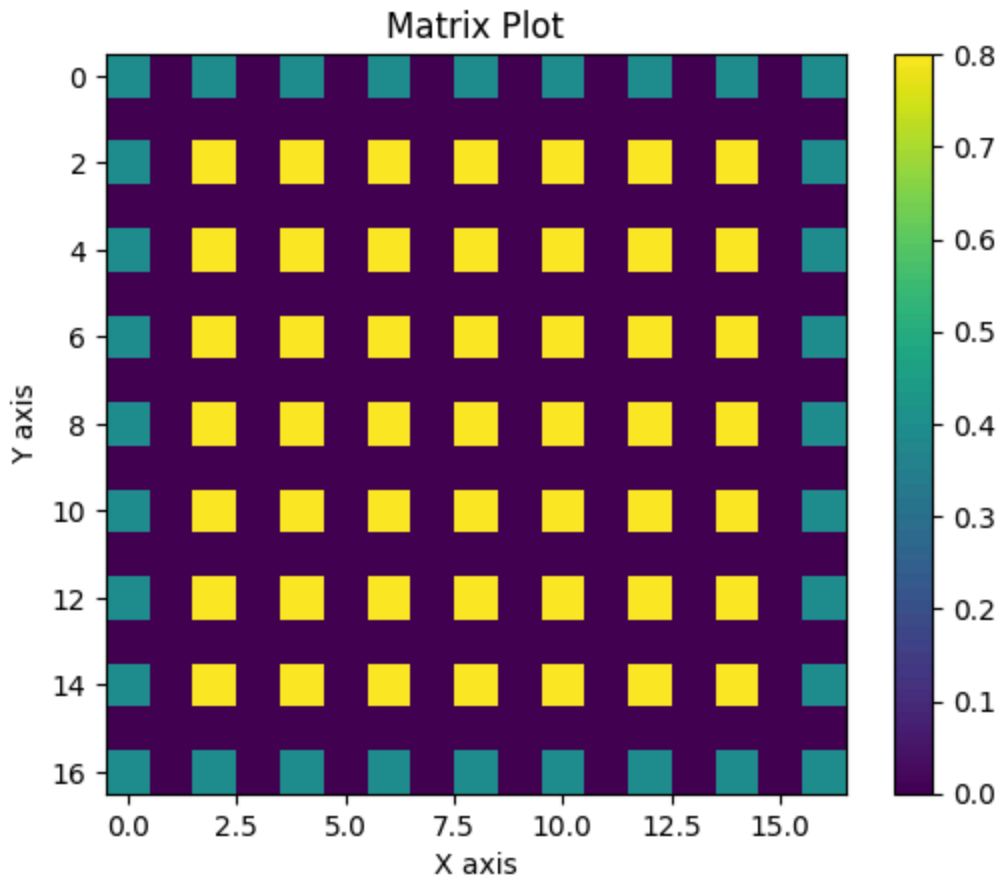


Figure 13. utilGrid representation

Figure 12 shows our Grid, where blue is only for gates, and the green border is only for the placement of terminals, however this is not an accurate representation of an FPGA layout. The second grid is our utilGrid, which more closely resembles the physical layout of an FPGA. Here the yellow is gates, the blue is terminals, and purple is space for routing. This class was made when we began writing our program with anticipation of use, however is not utilized in any of our placement methodology, however we do anticipate to use it for our Project 3. The utilGrid is made by mapping the grid of the Grid to it's respective spaces in the utilGrid with its constructor method:

```
utilGrid::utilGrid(vector<vector<square>> ogrid) {//ChatGPT aided
    int n = ogrid.size();
    std::vector<vector<square>> spacedMatrix(2 * n - 1, vector<square>(2 * n - 1));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            spacedMatrix[2 * i][2 * j] = ogrid[i][j];
        }
    }
    for (int i = 0; i < (n * 2) - 1; ++i) {
```

```
        for (int j = 0; j < (n * 2) - 1; ++j) {
            if (!(spacedMatrix[i][j].getType() == squareType::Node || spacedMatrix[i][j].getType() ==
squareType::Terminal)) {
                square s = square(squareType::Routing, 0);
                spacedMatrix[i][j] = s;
            }
        }
    }
    grid = spacedMatrix;
}
```

Figure 14.

This was written with the aid of chatGPT. We referenced some code produced by ChatGPT using the phrase “write a cpp program that will take an nxn matrix and will create a new matrix that spaces each square out by one space.” and produced the following:

```
#include <iostream>
#include <vector>

// Function to print a matrix
void printMatrix(const std::vector<std::vector<int>>& matrix) {
    for (const auto& row : matrix) {
        for (int elem : row) {
            std::cout << elem << " ";
        }
        std::cout << std::endl;
    }
}

// Function to create a spaced-out matrix
std::vector<std::vector<int>> spaceOutMatrix(const std::vector<std::vector<int>>& matrix) {
    int n = matrix.size();
    std::vector<std::vector<int>> spacedMatrix(2 * n - 1, std::vector<int>(2 * n - 1, 0));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            spacedMatrix[2 * i][2 * j] = matrix[i][j];
        }
    }

    return spacedMatrix;
}
```

```
int main() {
    // Input matrix dimensions
    int n;
    std::cout << "Enter the size of the matrix (n): ";
    std::cin >> n;

    // Input matrix elements
    std::vector<std::vector<int>> matrix(n, std::vector<int>(n));
    std::cout << "Enter the elements of the matrix:" << std::endl;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            std::cin >> matrix[i][j];
        }
    }

    // Create spaced-out matrix
    std::vector<std::vector<int>> spacedMatrix = spaceOutMatrix(matrix);

    // Print the spaced-out matrix
    std::cout << "Spaced-out matrix:" << std::endl;
    printMatrix(spacedMatrix);

    return 0;
}
```

Figure 15. ChatGPT produced spaced out matrix creation.

To explain how the function works, it takes our original grid which only consists of terminal and non-terminal nodes, and spaces every element out by a single square. These new spacing squares are of type ‘Routing.’ The program takes every coordinate from the original grid, some i and j, and maps it to the new utilGrid with coordinates 2*i and 2*j.

The last important part of the Grid class is it’s constructor as follows:

```
Grid::Grid(const std::map<std::string, Node>& nodes) {
    int totalNodes = nodes.size();
    // Estimate grid size: square root of total nodes times a factor (>1) for spacing, rounded up
    int gridSize = ceil(sqrt(totalNodes) * 1.1); // Adjust the 1.1 factor as needed

    // Initialize an empty grid
    grid = vector<vector<square>>(gridSize, vector<square>(gridSize, square()));
```

```
    // Perform initial placement
    initialPlacement(nodes);
}
```

Figure 16. Grid constructor.

Here the only input is the map of nodes read in by the scanner. The chosen size of the 1.1 times the size of the total amount of nodes, this way there are some extra empty spaces left over to move nodes into while executing move(), but also so the size of the simulated FPGA isn't significantly larger than what's needed so the 'cost' of FPGA isn't too great. Then a 2D vector of the square class is created, which is just an empty 2D vector to be filled with with the default value of a square. The most important part is the initialPlacement function, which has the purpose of iterating through the nodes map, and placing them in their appropriate places in the grid.

```
void Grid::initialPlacement(const std::map<std::string, Node>& nodes) {
    // Adjust the coordinate system to start from 0,0 if minimum is -33.
    int coordinateShift = 33; // Assuming -33 is the minimum coordinate.
    int maxX = 0, maxY = 0, minX = 0, minY = 0;
    // Containers for edge terminals.
    std::vector<const Node*> topEdge, bottomEdge, leftEdge, rightEdge, isTerminal;
    // For random placement
    std::mt19937 rng{ std::random_device{}() };
    std::set<std::pair<int, int>> occupiedPositions;

    for (auto pair : nodes) {
        if (pair.second.isTerminal()) {
            isTerminal.push_back(&nodes.at(pair.first));
        }
    }

    auto maxElementX = max_element(isTerminal.begin(), isTerminal.end(), [](const Node* a, const
Node* b) {//ChatGPT "how to find a max struct in a vector by its int value"
        return a->getX() < b->getX();
        });
    maxX = (*maxElementX)->getX();
    auto maxElementY = max_element(isTerminal.begin(), isTerminal.end(), [](const Node* a, const
Node* b) {
        return a->getY() < b->getY();
        });
    maxY = (*maxElementY)->getY();
    auto minElementX = min_element(isTerminal.begin(), isTerminal.end(), [](const Node* a, const
Node* b) {
        return a->getX() < b->getX();
```

```
    });
  minX = (*minElementX)->getX();
  auto minElementY = min_element(isTerminal.begin(), isTerminal.end(), [](const Node* a, const
Node* b) {
    return a->getY() < b->getY();
    });
  minY = (*minElementY)->getY();

  for (auto pair : nodes) {
    Node& node = pair.second;
    if (node.isTerminal()) { // Corrected to use function call syntax
      // Determine the edge for each terminal using getters
      if (node.getY() == maxY) topEdge.push_back(&nodes.at(pair.first)); // Top edge
      else if (node.getY() == minY) bottomEdge.push_back(&nodes.at(pair.first)); // Bottom edge
      else if (node.getX() == minX) leftEdge.push_back(&nodes.at(pair.first)); // Left edge
      else if (node.getX() == maxX) rightEdge.push_back(&nodes.at(pair.first)); // Right edge
    }
  }

  auto distributeTerminals = [&](const std::vector<const Node*>& edgeTerminals, char edge) {
    int numTerminals = edgeTerminals.size();
    int spacing = (edge == 't' || edge == 'b') ? grid[0].size() / (numTerminals + 1)
      : grid.size() / (numTerminals + 1);
    for (int i = 0; i < numTerminals; ++i) {
      int pos = 0;
      if (spacing == 1)  pos = ((i + 1) * spacing);
      else  pos = ((i + 1) * spacing) - 1; //want to start at index 1 no terminals at corners
      int x = 0, y = 0;
      if (edge == 't') { x = pos; y = 0; }
      else if (edge == 'b') { x = pos; y = grid.size() - 1;}
      else if (edge == 'l') { x = 0; y = pos; }
      else if (edge == 'r') { x = grid[0].size() - 1; y = pos; }
      write(y, x, square(squareType::Terminal, edgeTerminals[i]));
    }
  };

  distributeTerminals(topEdge, 't');
  distributeTerminals(bottomEdge, 'b');
  distributeTerminals(leftEdge, 'l');
  distributeTerminals(rightEdge, 'r');

  // Place non-terminal nodes randomly
  std::uniform_int_distribution<int> distX(1, grid.size() - 2), distY(1, grid[0].size() - 2);
```

```
    for (auto pair : nodes) {
        auto& node = pair.second;
        if (!node.isTerminal()) {
            bool placed = false;
            while (!placed) {
                int randomX = distX(rng);
                int randomY = distY(rng);
                if (occupiedPositions.find({ randomX, randomY }) == occupiedPositions.end()) {
                    // If position is not occupied, place the node
                    write(randomX, randomY, square(squareType::Node, &nodes.at(node.getName())));
                    occupiedPositions.insert({ randomX, randomY });
                    placed = true;
                }
            }
        }
    }

    for (int i = 1; i < grid.size()-1; i++) {
        for (int j = 1; j < grid[0].size()-1; j++) {
            if (grid[i][j].getNode() == nullptr) {
                Coords c(i, j);
                enodes.push_back(c);
            }
        }
    }
}
```

Figure 17. intialPlacement method.

How this function works is that there are given coordinates for the pin placements in the .pl file. initialPlacement considers these coordinates, keeps track of which pins are one which edge, indicated by the TopEdge, BottomEdge, RightEdge, and LeftEdge vectors. Then depending on how many are in each vector and how large the grid has been initialized, the pins will be evenly mapped throughout their respective edges. Once the pins have been placed along the perimeter, the rest of the grid is iterated through, and nodes are placed randomly.

5.4 Perturb

```
vector<Result> perturb(std::vector<Result>& population, map<std::string, Net>& nets, float w1, float
w2, float w3, int wireConstraint, map<string, Node> nodes, float selectP, float crossP, float mutP) {
        std::vector<Result> nextGeneration;
        std::random_device rd;
        std::mt19937 gen(rd());
```

```
	int count = 0; //to keep track of how many offspring created already

	//Selection:
	nextGeneration = tournamentSelection(population, nets, selectP);
	count += nextGeneration.size();
	//Crossover
	int cCount = population.size() * crossP;
	vector<Result> crossoverOffspring;
	//multithreadedCrossover(crossoverOffspring, nextGeneration, nets, cCount, nodes, w1, w2,
w3, wireConstraint);

	for (int i = 0; i < cCount; ++i) {
		int i1 = rand() % count;
		int i2 = 0;
		while (i2 != i1) {
			i2 = rand() % count;
		}

		Grid* parent1 = &nextGeneration[i1].g;
		Grid* parent2 = &nextGeneration[i2].g;
		Grid child = crossover(parent1, parent2, nets, nodes);
		bool rout = false;
		vector<Bounds> b;
		float cost = child.calcCost(w1, w2, w3, nets, rout, wireConstraint, b);
		Result r(child, cost, rout, b);
		nextGeneration.push_back(r);

	}

	count += cCount;
	//Mutation

	for (int i = count; i < population.size(); i++) {
		int in = rand() % count;
		Grid* copy = &nextGeneration[in].g;
		for (int i = 0; i < 1000; i++) { //1 mutation has a miniscule effect on cost, changed to
avoid early convergence
			std::uniform_int_distribution<> distx(1, copy->getGridX() - 2);
			std::uniform_int_distribution<> disty(1, copy->getGridY() - 2);
			int rx = distx(gen);
			int ry = disty(gen);
			copy->smartMutation(rx, ry, nextGeneration[in].bounds, nets);
```

```
                    //copy->mutation(rx, ry);
                }
                //copy->mutation(rx, ry);
                bool rout = false;
                vector<Bounds> b;
                float cost = copy->calcCost(w1, w2, w3, nets, rout, wireConstraint, b);
                Result r(*copy, cost, rout, b);
                nextGeneration.push_back(r);
        }
        return nextGeneration;
}
```

Figure 18. Perturb method, with significant methods to be covered highlighted.

The perturb will take in a population (vector<Result>) of any size, and then generate a new population from applying the three sub functions to the input. The method also takes in floats that define how much of the new population should come from each sub function. These percentages are highlighted blue to accentuate how they're utilized. It can be noticed that mutP goes unused since the left over spaces in the new population have to be filled. As shown in the highlighting, the next functions that will be covered in this section are tournamentSelection, crossover, mutation, and then calcCost will be the next section. To summarize how these functions are called here, a percentage of the new population that should be made up of selection and crossover will be taken as input. The selection portion will simply choose the most 'fit' candidate from the original population (taken as input), the crossover portion will iteratively take two randomly selected parent from the selection portion and perform crossover, and the rest of the empty space in the new population will come from mutated candidates from both the selection and crossover portion. This function has a complex time complexity which for now we shall call the summation of selection, crossover, and mutation.

5.4.1 Selection

```
bool compareByFloat(const Result& a, const Result& b) { //ChatGPT
        return a.cost > b.cost; // Change to < for ascending order
}

vector<Result> tournamentSelection(std::vector<Result> population, const std::map<std::string,
Net>& nets, float percentage) { //ChatGPT

        int topKNum = population.size() * percentage;
        vector<Result> topFive = population;
        sort(topFive.begin(), topFive.end(), compareByFloat);
        auto h = topFive.begin();
        advance(h, topKNum);
        vector<Result> newTopFive(topFive.begin(), h);
        return newTopFive;
```

```
}
```

Figure 19. tournamentSelection, and supplemental function compareByFloat.

To clarify the “ChatGPT” comments, we wrote the function on our own originally, then pasted our function into ChatGPT to remove any possible errors, then we had to edit it once again to ensure it functioned properly. When we originally wrote the function, we wrote for the selection of specifically five members of a population, which explains why ‘topFive’ appears here. The function begins by taking the ‘percentage’ input and using it to select a size of the resulting vector, e.g. if we input a population of size 10 with a percentage input of 0.7, the resulting selection output vector will be of size 7. A copy of the input vector is made, which the sort(function) from the <algorithms> library. Here is where the supplemental function compareByFloat is required, since the sort() function will have no way to compare our custom struct object to each other. For this part for the sake of speed we replied to our previously mentioned chatGPT generation “how to apply this to a struct with a float member.” where it produced pseudo code for comparByFloat(). This function has a time complexity of O(n*log(k) + c) where n is the number of members we’re selecting multiplied by log(k) which represents the sorting function on the member of size k, and summed with c which is the constant of all other commands in tangent with the sort.

5.4.2 Crossover

```
Grid* crossover(Grid* parent1, Grid* parent2, const std::map<std::string, Net>& nets, map<string,
Node> nodes) {
        // Assume Grid has a constructor that takes the size and nets to initialize an empty grid.
        auto child = new Grid(nodes);

        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<> dis(0, parent1->getGridSize() - 1);

        int crossoverPoint = dis(gen);

        // Copy up to the crossover point from parent1
        for (int i = 0; i <= crossoverPoint; ++i) {
                for (int j = 0; j < parent1->getGridSize(); ++j) {
                        auto node = parent1->getSquare(i, j).getNode();
                        if (node && !child->isNodePlaced(node)) {
                                child->placeNode(i, j, node);
                        }
                }
        }

        // Fill in the rest from parent2, avoiding duplicates
        for (int i = crossoverPoint + 1; i < parent2->getGridSize(); ++i) {
                for (int j = 0; j < parent2->getGridSize(); ++j) {
                        auto node = parent2->getSquare(i, j).getNode();
                        if (node && !child->isNodePlaced(node)) {
                                child->placeNode(i, j, node);
                        }
                }
        }
        return child;
}
```

Figure 20. Crossover Function

Above is the crossover function. This method was created originally by us, then pasted into chatGPT for the correction of any errors, and addition of comments to provide explanation. As input, this function will take in two parent grids. When the concept was presented in class, it was taught to produce two children by splitting both parents in half, then swapping the bottom halves of the parents, as seen in the figure below. At this point the function would have a time complexity of approximately O((2n/2)^2) since half of each parent is iterated through, however this finds a drastic change after we worked out bugs.

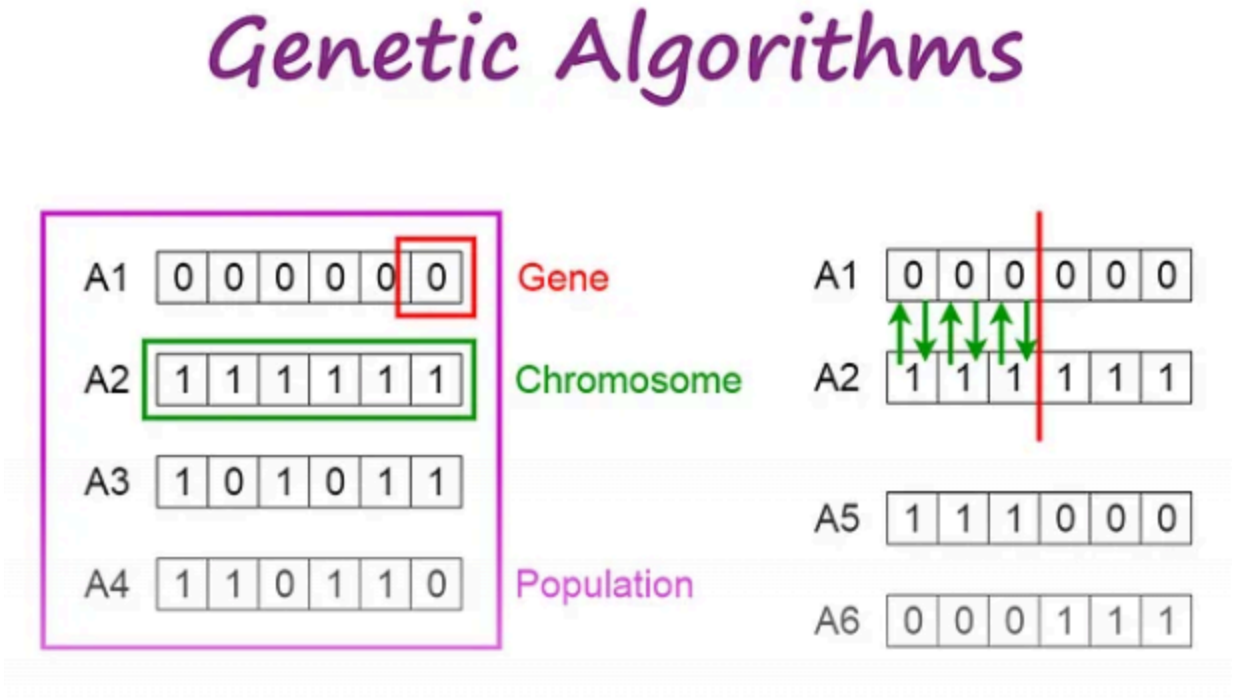


Figure 21. Genetic Algorithm crossover. [6]

Our implementation is different in that it picks a random point in the grid to place the split, and only produces one child per random set of parents. Thinking of this sampling in the terms of a hyperspace mentioned by Whitley [7], our method will still explore the hyperspace in a similar fashion, however in our we will explore points in the hyperspace that would take the original method much longer to get. Whether this is exactly better or worse is tough to predict on our own, and will be determined if a difference is made at all during experimentation.

To explain further how our method works, once the splitting point, 'crossoverPoint' is created, the new child grid will iteratively take the placements from the first parent until the crossover point, then continue with the placement from the second grid in the latter half. Figure 20 knows not to place a node if it has already been placed, however it will leave those nodes empty, making for missing gates in our child grid. Our renovation will be shown in section 6. This code will also be further explained then.

5.4.3 Mutation

```
void Grid::move(int x1, int y1, int x2, int y2) {
    if (grid[x1][y1].getType() == squareType::Terminal || grid[x2][y2].getType() ==
squareType::Terminal) {
        cout << "Cannot cast move on terminal nodes." << endl;
    }
    else if (grid[x1][y1].getType() == squareType::Node && grid[x2][y2].getType() ==
squareType::Node) {
```

```
    if (x1 > grid.size() || y1 > grid[0].size() || x2 > grid.size() || y2 > grid[0].size()) cout << "Trying to
Move out of bounds" << endl;
    else  if (grid[x2][y2].getNode() == nullptr) {
        if (grid[x1][y1].getNode() != nullptr) {
            nodeCoords[grid[x1][y1].getNode()->getName()].x = x2;
            nodeCoords[grid[x1][y1].getNode()->getName()].y = y2;
        }
        grid[x2][y2] = grid[x1][y1];
        grid[x1][y1].setNode(nullptr);
        updateEmpties(x1, y1, x2, y2, grid[x1][y1].getType() == squareType::Terminal);
    }
    else if (grid[x1][y1].getNode() == nullptr) {
        nodeCoords[grid[x2][y2].getNode()->getName()].x = x1;
        nodeCoords[grid[x2][y2].getNode()->getName()].y = y1;
        grid[x1][y1] = grid[x2][y2];
        grid[x2][y2].setNode(nullptr);

        updateEmpties(x2, y2, x1, y1, grid[x1][y1].getType() == squareType::Terminal);
    }
    else cout << "Trying to move to none empty Node square" << endl;
  }
}
```

Figure 22. Move() function.

First we'll explain the move function as well as provide approximate pseudo code in Figure 23. since the code is difficult to read. As input two sets of coordinates are entered, the coordinates of the node we'd like to move (x1, y1), and coordinates of the empty node we'd like to move to (x2, y2). The first condition checked is that both squares are of Node type, so that no terminal squares can be moved, and no Node can be moved to an empty terminal square. A secondary condition is to check whether the coordinates are even within the grid. The tertiary condition is whether the space we'd like to move the node to is actually empty, by checking if the square's Node* is nullptr. The time complexity of this method is O(n^2 + k), where n is the count of elements in the grid, and k is a constant referring to the other commands in tangent with updateEmpties(). updateEmpties has a time complexity of O(n^2) since it iterates through the grid with a nested for loop as seen in Figure 24. The purpose of updateEmpties is to update the Grid class' vector of coordinates where nodes are empty.

```
Move(two sets of coordinates){
        If coordinates point to terminal -> exit //cant move terminals
        Else if both coordinates points to Node type square {
                If coordinates within grid{
                        If first set is null{
                        second set of coords get first's value;
```

```
                    Update record of Grid Class;
                }
                Else if second set is null{
                    First set of coords gets seconds value;
                    Update record;
                }
            }
        }
}
```

Figure 23. Move() Pseudo Code.

```
void Grid::updateEmpties(int x1, int y1, int x2, int y2, bool isTerminal) {
   Coords a(x2, y2);
   Coords b(x1, y1);
   enodes.push_back(b);
   grid[x2][y2].setNode(nullptr);
   for (auto it = enodes.begin(); it != enodes.end(); ++it) {
      if ((*it).x == a.x && (*it).y == a.y) {
         enodes.erase(it);
         break; // Optional, if you know there is only one element with the value 3
      }
   } //somtimes causes errors
}
```

Figure 24. updateEmpties function.

```
void Grid::swap(int x1, int y1, int x2, int y2) {
   if (grid[x1][y1].getType() == squareType::Terminal || grid[x2][y2].getType() ==
squareType::Terminal) {
      cerr << "Cannot cast move on terminal nodes." << endl;
   }
   else if (grid[x1][y1].getType() == squareType::Node && grid[x2][y2].getType() ==
squareType::Node) {
      if (x1 > grid.size() || y1 > grid[0].size() || x2 > grid.size() || y2 > grid[0].size()) cerr << "Trying to
Swap out of bounds" << endl;
      else {
         square temp = grid[x1][y1];
         grid[x1][y1] = grid[x2][y2];
```

```
            grid[x2][y2] = temp;
        }
    }
}
```

Figure 25. Swap Code.

The other sub function used in Mutation() is the swap function, where two non-terminal nodes are swapped with each other. This method uses the same primary and secondary conditions as the move() function, however doesn't utilize the tertiary function since it doesn't matter whether the second set of coordinates are empty or not.

```
void Grid::mutation(int x1, int y1) {
    int ran = rand() % 2;
    int rand_x = 0;
    int rand_y = 0;
    int ran_i = 0;

    if (ran % 2 == 0) { //Even will use the swap function
        rand_x = rand() % grid.size();
        rand_y = rand() % grid[0].size();
        swap(x1, y1, rand_x, rand_y);
    }
    else {  //Odd will use the move function
        ran_i = rand() % enodes.size();
        rand_x = enodes.at(ran_i).x;
        rand_y = enodes.at(ran_i).y;
        move(x1, y1, rand_x, rand_y);
    }
}
```

Figure 26. Mutation Code.

This function uses a random number generator to decide whether to call the swap function or the move function with a 50% chance for each. If swap is called, then two random sets of coordinates with the squareType Node will be swapped. If move is called then we need to utilize the empty nodes vector<Coords> to randomly find one of the empty nodes, and add empty node coordinates after the move is executed. It should also be noted that all of these functions are class functions, so when they're called in our simulated annealing function we will need to make a copy of the original grid as to not change the original object.

5.5 Cost Function

```
float Grid::calcCost(float const w1, float const w2, map<string, Net> const nets, bool& routable, int wireConstraint, vector<Bounds>& bounded) const {
    float totalCost = 0, totalLength = 0, overlapCount = 0, critCost = 0;

    //vector<Bounds> bounded;
    bounded.clear();//incase bounded already populated
    for (const auto& netPair : nets) { // Assuming 'nets' is accessible and stores the Net objects
        const Net* net = &netPair.second;
        int xmin = net->Nodes.at(0)->getX(), xmax = xmin, ymin = net->Nodes.at(0)->getX(), ymax = ymin;
        Bounds newBounds;
        // Calculate the wirelength for this net by finding the x and y bounds (half-param measure)

        for (size_t i = 0; i < net->Nodes.size(); ++i) { //iterate through nodes and find min/max x/y
            int x = net->Nodes.at(i)->getX();
            int y = net->Nodes.at(i)->getY();
            if (x < xmin) xmin = x;
            if (x > xmax) xmax = x;
            if (y < ymin) ymin = y;
            if (y > ymax) ymax = y;

            totalLength = abs(xmax - xmin) + abs(ymax - ymin);
            if (net->isCritical) critCost += totalLength / 2; //if the net is critical then add additional cost equivlent to 1/2 net length

            newBounds.x1 = xmin, newBounds.x2 = xmax, newBounds.y1 = ymin, newBounds.y2 = ymax, newBounds.net = net;
        }
        bounded.push_back(newBounds);

        //calculated overlap of nets
        int olcount = 0;
        for (Bounds const bounds : bounded) {
            if (bounds.net->name != netPair.first) {
                //overlap cost (total nets overlap)
                float x_overlap = max(0, min(newBounds.x2, bounds.x2) - max(newBounds.x1, bounds.x1)); //x overlap
                float y_overlap = max(0, min(newBounds.y2, bounds.y2) - max(newBounds.y1, bounds.y1));//y overlap
                float interArea = x_overlap * y_overlap; //total intersection area
                float area1 = abs(newBounds.x2 - newBounds.x1) * abs(newBounds.y2 - newBounds.y1);
```

```
//calculating area of current net box
            float area2 = abs(bounds.x2 - bounds.x1) * abs(bounds.y2 - bounds.y1); //net j box area
            if ((interArea / min(area1, area2)) <= 0.25) { //if overlap is greater than 25%
               olcount++;
               overlapCount++;
               if (olcount > wireConstraint) routable = false; //if overlapping net boxes > constraint then
the design is not routable
            }
         }
      }
   }
   delete& bounded;
   float ocnorm = overlapCount / nets.size(); //normalized overlap count cost => total count of nets is
max, min is zero
   float den = (ug.grid.size() * ug.grid[0].size());
   float tlnorm = totalLength / den; //normallized total length cost => total grid area * net count is max,
min is ~ 1
   totalCost = (w1 * tlnorm) + (w2 * ocnorm);
   return totalCost;
}
```

Figure 27. calcCost() function.

The cost function takes in weights w1 and w2 as inputs, as well as the nets and nodes maps, a bool telling whether the object is routable, a wireConstraint used to determine whether the object is routable, and a vector<Bounds> telling the bounding box of each net. There are then three factors that determine the value of cost, being total net length, total amount of overlapping net bound boxes, and length of critical nets.

To simplify the function, we can split it into two sections based on the inner for loops. The first for loop is for iterating through each Net, where an inner for loop is called and the min and max bounds of the nets bounding box is found, effectively giving us a half-parameter length, or manhattan length as well as parameters for our Bounds objects. The min and max bounds of the bounding box are found by iterating through each node in a net and recording the minimum and maximum x and y values. The net length is then calculated by subtracting the minimum x and y position from their respective maximum. After this length is found and Bounds object created with a pointer to the current net, it is checked whether the net is considered critical. If it is flagged as critical then a critCost will be added equal to 0.5 of the nets length. This means the critCost will be an additional conditional cost proportional to the net length cost. Figure 28. shows the function that determines whether a net is critical:

```
bool sortByValueDescending(const std::pair<string, Net>& a, const std::pair<string, Net>& b)
{//ChatGPT
        return a.second.nodesSize > b.second.nodesSize;
```

```
}

void findTop10Percent(const std::map<string, Net>& inputMap) {//ChatGPT
	// Calculate the number of elements that constitute the top 10%
	int top10PercentSize = inputMap.size() * 0.1;

	// Convert the map to a vector of pairs for sorting
	std::vector<std::pair<string, Net>> vec(inputMap.begin(), inputMap.end());

	// Sort the vector by value in descending order
	std::sort(vec.begin(), vec.end(), sortByValueDescending);

	// Output the top 10% elements
	std::cout << "Top 10% elements:" << std::endl;
	for (int i = 0; i < top10PercentSize; ++i) {
		vec[i].second.isCritical = true;
	}
}
```

Figure 28. Functions to determine whether a net is critical

Since the dataset does not explicitly flag what nets are to be considered critical, and node size is not a factor in FPGA design, we decided to sort the net list by the amount of nodes they contain and mark the top 10% as critical. 10% is a large amount, and this may be changed depending on performance. Getting into explaining the method's functionality, it was generated with the help of ChatGPT as indicated in the comment by the declaring line. The phrase used was “write code to ten percent in a map of ints,” which was used as a reference to create the method in Figure 28. First the function creates an output vector with a size equal to ten percent of the inputs. The sort() method is used again from the <algorithm> library, and a similar sorting method is used to that in Figure 19. Then the first ‘topTenPercentSize’ elements are then flagged as critical.

After the Bounds information is recorded in the first for loop, the second for loop begins by iterating through all existing Bounds objects in the vector, and checking if there is any overlap. The method to finding the overlap is to first find the intersecting area between the two bounds, record the area of each Bounds, then use a conditional to check if there is at least a 25% overlap between the two. We chose to define true overlap as an overlap of at least 25% since if there is an overlap of a few grid coordinates, this would not indicate an unroutable situation, and it is likely there is room to route. For each Bounds we are checking, we are also keeping a count ‘olcount’ to determine if there is more overlap than a given wireConstraint. This means that if there are more than ‘wireConstraint’ Bounds overlapping, then the grid is unroutable. There is another variable ‘overLapCount’ that keeps track of the total amount of overlap present in the whole grid, which gives an overlap cost.

Then after both we have to normalize the outputs of the net length cost and the overlap cost so one doesn't drown the other. These normalization factors are found with the equations at the variables 'ocnorm,' 'crnorm,' and 'tlnorm.'

5.6 Simulated Annealing

```
Result simulatedAnnealing(vector<Result> initialGrids, float const w1, float const w2, float const w3,
map<string, Net> nets, int wireConstraint, map<string, Node> nodes) {
	bool routable = false; // Ensure it's declared
	//double t = generateInitialTemp(initialGrids, 5., w1, w2, w3, nets, routable, wireConstraint,
nodes);
	double t = 92.52;
	double initT = t;
	vector<Result> population = initialGrids;
	vector<Result> new_pop;
	vector<Result> best_pop = population;
	double deltaC = 0;
	cout << "Initial Cost: " << bestCost(population).cost << endl;

	std::ofstream outFile("best_costs.txt");
	if (!outFile) {
		std::cerr << "Failed to open file for writing.\n";
		exit(1); // Handle error as needed
	}

	int iteration = 1;
	while (t > (0.01 * initT) && routable == false) {
		if (iteration == 3) {
			cout << "flag" << endl;
		}
		cout << "Iteration " << iteration << "; Temp = " << t << endl;
		float s = 0.3, c = 0.3, m = 0.25;
		new_pop = perturb(population, nets, w1, w2, w3, wireConstraint, nodes, s, c, m);
//NEED PERTURB FUNCTION //NEEDS TO RETURN LIST OF GRIDS : COST : ROUTABLE?
		Result nbc = bestCost(new_pop);
		auto it = std::find_if(new_pop.begin(), new_pop.end(),
			[&nbc](const Result& mem) { return nbc.cost == mem.cost; });
		int index = 0;
		string st = "";
		if (it != new_pop.end()) {
			int index = distance(new_pop.begin(), it);
		}
		if (index < s * population.size()) {
```

```
            st = "Selection";
}
else if (index < (s + c) * population.size()) {
            st = "Crossover";
}
else {
            st = "Mutation.";
}
if (nbc.routable == true) {
            outFile.close();
            return nbc;
}
deltaC = nbc.cost - bestCost(population).cost;
cout << "\t Best Delta C = " << deltaC << endl;

random_device rd;
mt19937 gen(rd());
uniform_real_distribution<double> dis(0.0, 1.0);
double r = dis(gen);
double e = exp(deltaC / t);

sort(population.begin(), population.end(), compareByFloat);
sort(new_pop.begin(), new_pop.end(), compareByFloat);

double sum1 = 0, sum2 = 0;
for (int i = 0; i < 3; i++) {
            sum1 += population.at(i).cost;
            sum2 += new_pop.at(i).cost;
}
bool explore = false;
if (sum2 < sum1) {
            population = new_pop;
            if (nbc.cost < bestCost(best_pop).cost) {
                        best_pop = new_pop;
                        cout << "\t \t new best population!" << endl;
            }
}

else if (r > e) {
            population = new_pop;
            explore = true;
}
```

```
                // Updated logging line including temperature and best delta C
                outFile << "Iteration: " << iteration << ", Temperature: " << t << ", Best Cost: " <<
bestCost(best_pop).cost << ", Best Delta C: " << deltaC << "Best Result from: " << st << ",
Exploration?: " << explore << endl;

                t = schedule(t, initT);
                iteration++;
        }

        outFile.close();
        return bestCost(best_pop);
}
```

Figure 29. Simulated Annealing Function.

To put everything together is to use the simulated Annealing function. As inputs, we take our wire constraints and weights, as well as the net list. Then we call our aforementioned generateInitalTemperature() method to get our initial temperature. Before this function is called, we'll call the generateInitialGrids() function which will be shown in the next section. In summary it creates some number k of grids and puts them into a vector<Result>. After we generate an initial temperature, we'll save it to a separate variable from t called 'initT' so it can be used for our scheduling method. We'll also prepare some empty vectors new_pop and best_pop. This is where we'll store our best population if it has the best cost. New pop will be used to put our new population into so we can compare our new pop cost to the cost of the original population. After this we begin a while loop that will continue until a routable solution is found. For each loop iteration a delta Cost value will be found using the cost from the best grid in the old population - the cost from the best grid in the new population. After we find our cost difference, if it's better for the new population we will replace the old with the new, if it's worse we'll calculate whether we should explore based on temperature. An additional feature different from the process presented in class is that we compare a better new population to our best population to see if it's better and will save it if true. The only new function presented in Figure 30. is bestCost() which will take a population as input and return the Result struct object with the best cost:

```
Result bestCost(vector<Result> results) {
        Result best;
        double mincost = results.at(0).cost;
        if (results.at(0).routable) return results.at(0);
        for (int i = 1; i < results.size(); i++) {
                if (results.at(i).routable) {
                        return results.at(i);
                }
                else if (results.at(i).cost < mincost) {
                        mincost = results.at(i).cost;
```

```
                    best = results.at(i);
                }
            }
            return best;
}
```

Figure 30. bestCost() function.

As we can see by the Figure 30. We simply iterate through every Result object. The first thing checked is whether the result is routable, since this automatically makes it the best placement. If none are routable then we will return the result with the minimal cost.

5.7 Main

Figure 31. shows our main function, where we input our directories the read from said directories with read(). After we've effectively read in a net and node list, we're able to call top10Percent() to identify what nets should be considered critical, using the amount of nodes in the net as a criteria. After we've marked critical nets its time to create an initial population of grids with the createInitialGrids function. This created 10 randomly generated grids, however the pins of which are all the same based on the .pl file. Then we take a record of our initial best grid from the created population. Now that we've obtained an initial population, we can repeatedly cast perturb on it depending on temperature using the simulatedAnnealing function. Once we find a 'best' grid from simulatedAnnealing, we take a record of the netlist bounds once again so we may plot this to the original and compare. Finally the output file records the costs of everything, when a routable solution is found, how often explore is used, and which subfunction creates the new best grid for each simulatedAnnealing iteration.

```
int main() {
        auto start = std::chrono::high_resolution_clock::now();

        std::string netfile = "P2Benchmarks\\ibm01\\ibm01.nets";
        std::string nodefile = "P2Benchmarks\\ibm01\\ibm01.nodes";
        std::string plfile = "P2Benchmarks\\ibm01\\ibm01.pl";

        std::map<std::string, Node> nodes;
        std::map<std::string, Net> nets;
        int numNets = 0, numPins = 0, numNodes = 0, numTerminals = 0;

        read(netfile, nodefile, plfile, nodes, nets, numNets, numPins, numNodes, numTerminals);

        // Open summary file early to write benchmark summary before proceeding
        std::ofstream summaryFile("optimization_summary.txt", std::ios_base::app);
        if (summaryFile) {
                // Benchmark information written at the start
                summaryFile << "Benchmark Summary:\n";
                summaryFile << "Total Nodes: " << numNodes << "\n";
```

```
            summaryFile << "Total Nets: " << numNets << "\n";
            summaryFile << "Total Pins: " << numPins << "\n";
            summaryFile << "Total Terminals: " << numTerminals << "\n\n";
        }
        else {
            std::cerr << "Failed to open summary log file for appending.\n";
            return 1;
        }

        findTop10Percent(nets);
        std::vector<Result> init = createInitialGrids(nodes, 10, 1.0, 0.5, 1.0, nets, 4);
        double initialCost = init.empty() ? 0 : bestCost(init).cost;
        exportForVisualization(init[0], "initial.txt");

        if (init.empty()) {
            std::cerr << "Failed to create initial grids. Aborting optimization.\n";
            summaryFile.close();
            return 1;
        }

        Result bestResult = simulatedAnnealing(init, 1.0, 0.5, 1.0, nets, 4, nodes);
        exportForVisualization(bestResult, "results.txt");
        if (!bestResult.routable) {
            std::cerr << "Failed to find a routable solution.\n";
            summaryFile << "Failed to find a routable solution.\n";
        }
        else {
            std::cout << "Optimization successful. Best cost: " << bestResult.cost << ".\n";
        }

        auto stop = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);
        double improvement = initialCost - bestResult.cost;
        double improvementPercentage = initialCost > 0 ? (improvement / initialCost) * 100 : 0;

        summaryFile << "Optimization Results:\n";
        summaryFile << "Initial Cost: " << initialCost << "\n";  // Log the initial cost
        summaryFile << "Final Cost: " << bestResult.cost << "\n";
        summaryFile << "Improvement: " << improvement << " (" << improvementPercentage <<
"%)\n";
        summaryFile << "Routable Solution Found: " << (bestResult.routable ? "Yes" : "No") << "\n";
        summaryFile << "Total Execution Time: " << duration.count() / 1000.0 << " seconds\n\n";
        summaryFile.close();
```

```
        return bestResult.routable ? 0 : 1;
}
```

Figure 31. Main function

```
vector<Result> createInitialGrids(const std::map<std::string, Node>& nodes, int k, float const w1, float
const w2, map<string, Net> const nets, int wireConstraint) {
        vector<Result> init;
        std::random_device rd;
        std::mt19937 g(rd());

        for (int i = 0; i < k; ++i) {
                Grid grid(nodes); // Use `grid` instead of `g` to avoid confusion with `std::mt19937 g`
                bool routable = false;
                vector<Bounds> bounds;
                float cost = grid.calcCost(w1, w2, nets, routable, wireConstraint, bounds);

                init.emplace_back(std::move(grid), cost, routable, std::move(bounds));
        }
        return init;
}
```

Figure 32. createInitialGrids function

Here is the aforementioned function that will create our initial population. Here we simply call the constructor for k different grids, get their cost and whether they're routable, then push it back into our initial population vector.

6. Implementation Issues

After implementation, there were a large amount of bugs, but the main issue was the crossover function. It took us considerable time to find a good way to handle duplicates, as well as fill in any missed nodes from the parent members. An example of a node that may be missing is if we are calling crossover on parent 1 and parent 2, and when we do this the child inherits the first half of parent 1 and the bottom half of parent 2, there may be a node at the very end of parent 1 and very beginning of parent 2. Therefore this node will be lost after crossover since it's not seen during the first or second halves of the child. In this case we needed to add a set that would keep track of what nodes have been placed, then look through the parent grids and add the nodes that are not in this set. The fixed code can be seen below in Figure 33:

```
Grid crossover(Grid* parent1, Grid* parent2, const std::map<std::string, Net>& nets, const
std::map<std::string, Node> nodes) {
        Grid child = Grid(nodes.size());
```

```
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, parent1->getGridSize() - 1);

    int crossoverPoint = dis(gen);
    std::set<std::string> placedNodeNames; // To track which nodes have been placed
    set<string> toBePlaced;
    for (auto node : nodes) {
        toBePlaced.insert(node.second.getName());
    }
    // First pass: place nodes from both parents up to crossover point
    for (int i = 0; i <= crossoverPoint; ++i) {
        for (int j = 0; j < child.getGridY(); j++) {
            if (parent1->getSquare(i, j).getNode() != nullptr) {
                Node* node = new Node;
                *node = nodes.at(parent1->getGrid()[i][j].getNode()->getName());
                tryPlaceNode(node, i, j, placedNodeNames, child, toBePlaced);
            }
        }
    }
    for (int i = crossoverPoint + 1; i < child.getGridSize(); i++) {
        for (int j = 0; j < child.getGridY(); j++) {
            if (parent2->getSquare(i, j).getNode() != nullptr) {
                Node* node = new Node;
                *node = nodes.at(parent2->getGrid()[i][j].getNode()->getName());
                tryPlaceNode(node, i, j, placedNodeNames, child, toBePlaced);
            }
        }
    }
// Second Pass: include all potentially missed nodes
    for (int i = 0; i < child.getGridX(); i++) {
        for (int j = 0; j < child.getGridY(); j++) {
            if (child.getSquare(i, j).getNode() == nullptr && !toBePlaced.empty()) {
                Node* node = new Node;
                *node = nodes.at(*toBePlaced.begin());
                tryPlaceNode(node, i, j, placedNodeNames, child, toBePlaced);
            }
        }
    }

    child.updateEnodes();
```

```
        return child;
}
```

Figure 33. Adjusted crossover().

In order to thoroughly make sure all nodes that are missed are placed, we decided to add the use of sets to mark which nodes are still yet to be placed, and which nodes have already been placed. Once we make our first pass through both parents, our set toBePlaced contains all nodes that still need to be placed. We then make a second pass through the child grid and iteratively place nodes from the set wherever an empty node is found.

Besides bugs in the code there were also theoretical issues that came up during our implementation. The main issue is as stated in the forum post by user 'trailmax' [8]. The issue described is the appearance of duplicates in a population, causing a convergence. Our issue is that mutation would only move or swap nodes once. Moving one or two nodes in a grid has a really small effect on the change in a cost function. This means if we mutate the best member several times, then the selection for the next population will select multiple nodes of near the same cost and are exactly the same give or take a couple nodes. Our solution for this was to schedule the cooling to be much slower so there is more time to mutate, as well as make more radical mutations. Instead of mutating one or two nodes, we would mutate 100-200 nodes, but even this results in only a ~0.015% change for the first benchmark. The cooling schedule was moving much too quickly, since the population size is only ten, so for every small change the temperature decreases considerably rather frequently.

An issue faced is the lack of improvement found when using normal mutation. For this we added an original function titled smartMutation, as found in Figure 34. The difference in this function is that instead of move a random node to any random location, the method would first sift through the nets the randomly selected node is a part of, choose the net with the larges size, then move the node somewhere within the bounding box of the net. Finding the largest net is done by sorting the netlist based on it's nodes list size, then choosing the first valid net. If there is no valid net then regular mutation will be called.

```
void Grid::smartMutation(int x1, int y1, vector<Bounds> bo, map<string, Net> nets) {
    int ran = rand() % 2;
    int rand_x = 0;
    int rand_y = 0;
    int ran_i = 0;
    Bounds x;
    bool flag = false;
    if (grid[x1][y1].getNode() == nullptr) {
        flag = true;
    }
    if (flag) {
        mutation(x1, y1);
        return;
```

```
    }
    for (Bounds b : bo) {
        for (auto net : grid[x1][y1].getNode()->getNets()) {
            if (b.name == net->name) {
                x = b;
                exit;
            }
        }


    }


    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> xdistribution(x.x1, x.x2);
    std::uniform_int_distribution<int> ydistribution(x.y1, x.y2);
    if (x.x2 > 125 || x.y2 > 125) {
        cout << "Crazy Bounds" << endl;
    }
    if (ran % 2 == 0) { //Even will use the swap function
        rand_x = xdistribution(gen);
        rand_y = ydistribution(gen);

        swap(x1, y1, rand_x, rand_y);
    }
    else {  //Odd will use the move function
        for (auto c : enodes) {
            if (c.x < x.x2 && c.x > x.x1 && c.y < x.y2 && c.y > x.y1) {
                if (c.x > 125 || c.y > 125) {
                    cout << "Crazy Coordinate" << endl;
                }
                move(x1, y1, c.x, c.y);
                return;
            }
        }
        ran_i = rand() % enodes.size();
        rand_x = enodes.at(ran_i).x;
        rand_y = enodes.at(ran_i).y;
        move(x1, y1, rand_x, rand_y);
    }
}
```

Figure 34. SmartMutation

Another issue is how long it takes for populations to get better through the mutate function. From here we introduced a new function called smartMutation() where nodes can only be swapped and moved within their nets bounds. This solution is not perfect since it only looks at one net, however it guarantees that at least one Net will be improved in length.

A dilemma we were faced with was whether smartMutation could really be considered part of a genetic algorithm, or if it strays from this idea since it prioritizes improvement, but lead to a neglect of exploration. This dilemma however was passed quickly since the performance of the normal mutation surpassed the smartMutation.

There was also an issue with the results as seen when testing ibm02 in the next section, that cost improvement completely spiked towards the end of our algorithm. We did investigate, but could not come to a definitive conclusion on why this would occur. This is elaborated further in the next section.

7. Experimental Results

7.1 Testing general hyperparameters on IBM01

Before we begin running experiments on every benchmark, it's import to find what makes our algorithm optimal, in order to do this we shall experiment with the mutation function used, makes changes to the cost function weights to test the ratio of effectiveness of overlap to wire length, and make changes to the size of the target grid. These tests will only be conducted on the first benchmark since it's the smallest and will take the least time. Then after this point we will include whatever changes show improvement, then apply this to other benchmarks.

Mutation	W1	W2	W3	GridSize Factor	Improvement %	Total Time (Minutes)	Routable?
Normal	1	1	1	1.1	8.19	47	No
Smart	1	1	1	1.1	3.25	48	No
Normal	1.5	1	1	1.1	3.28	50	No
Normal	0.5	1	1	1.1	2.34	62	No
Normal	1	1	1	1.5	12.57	53	No

Table x. Testing of hyperparameters on benchmark ibm01.

To clarify what is shown in Table x, first we indicate whether smartMutation or normal mutation is being used, the values of all weights, where W1 is wirelength, W2 is overlap, and W3 is critical Net length. GridSize factor refers to the size of the grid in relation to the size of the list of nodes. Therefore if we have 100 nodes there will be 110 grid spaces. Improvement % shows the amount of decrease in cost, total time refers to the total amount of time taken to run the code, and routable indicates whether a routable solution has been found. I have highlighted what will be changed in each row, and left no highlights in the first row to indicate that this will be our constant. From debugging the value of normalized net length is

around 100 and the value for overlap is around 400, meaning overlap heavily out weighs net length. This way if we set W2 to 0.25 overlap and net length will have a similar effect on the change in cost. We've also found that normalized critLength comes out to around 9 so if multiply W3 by 3 then critlength will also have a comparable effect on cost.

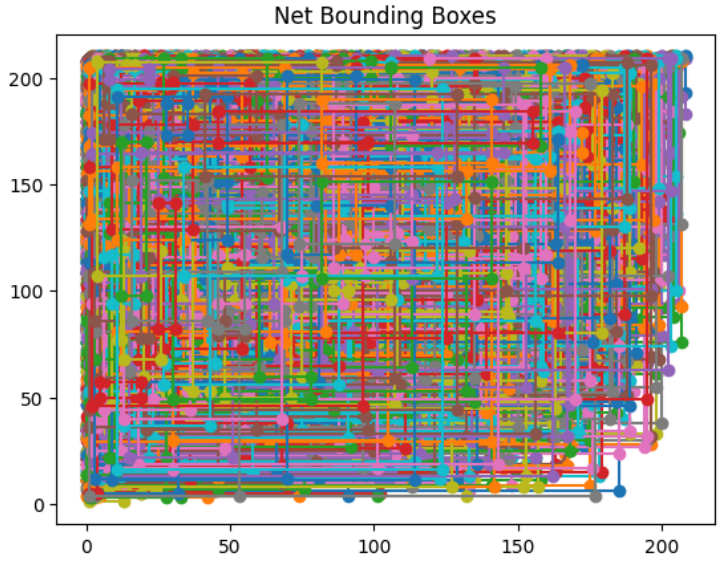
7.2 Testing and fine Tuning with ibm02

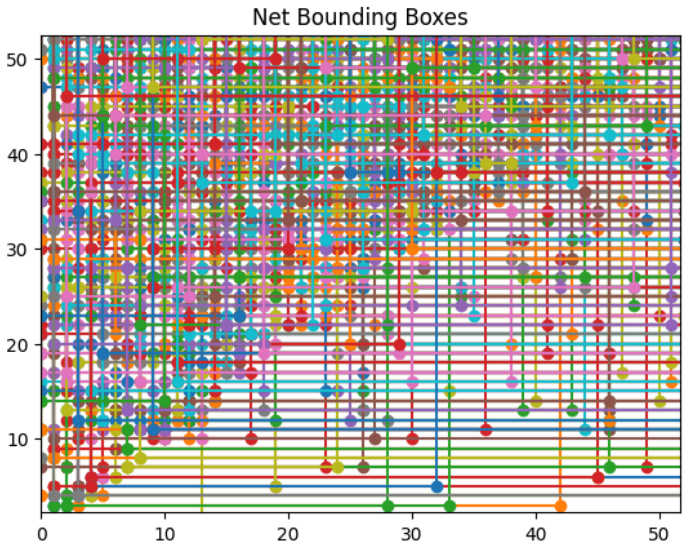
Here we'll use imb02 as an example to show the placements before and after optimization. This is with the weights suggested before. Table x. shows our found optimal parameters:

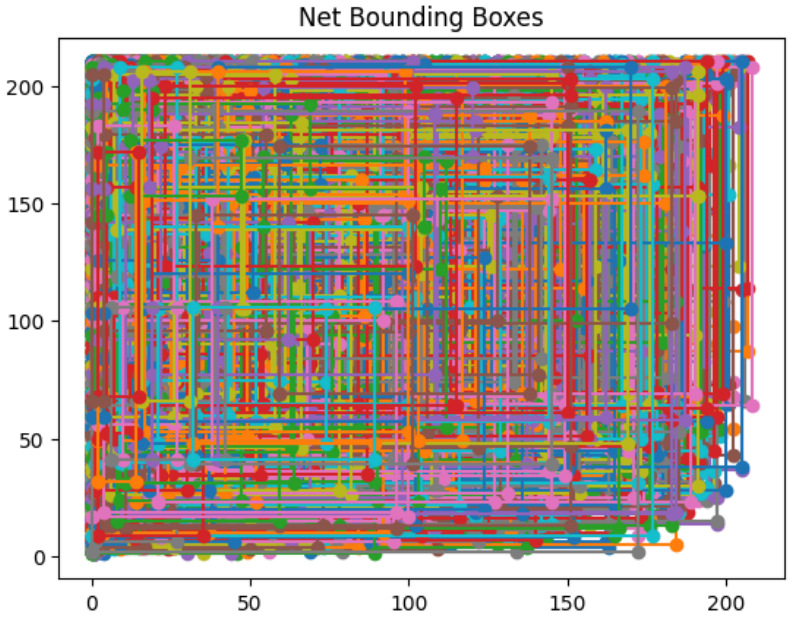
Hyperparameter	Value
Mutation	Normal mutation
w1	1.0
w2	0.25
w3	10
GridSizeFactor	1.5

Table 1. Our determined optimal hyperparameters

To provide a visual for how our results should look, we will use ibm02. Figure 34. show the bounding net boxes before and after optimization:




Net Bounding Boxes
50
40
30
20
10
0
10
20
30
40
50


Net Bounding Boxes
200
150
100
50
0
0
50
100
150
200

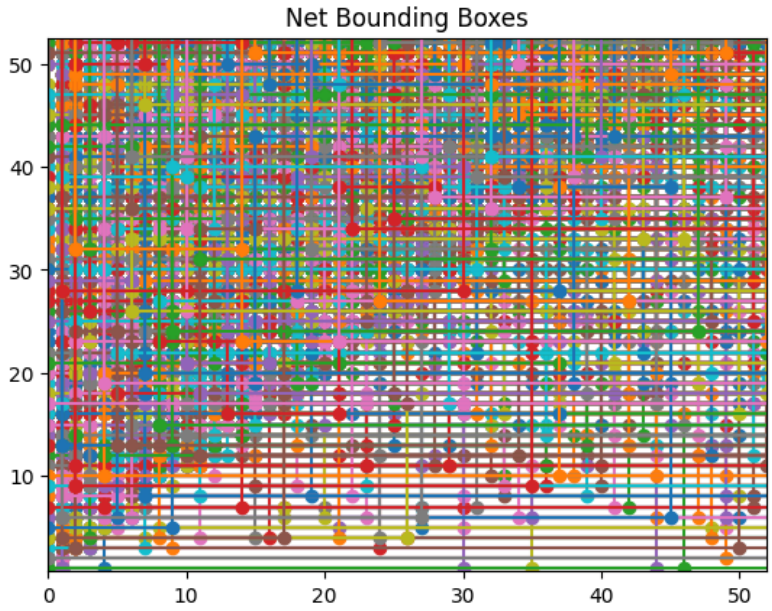
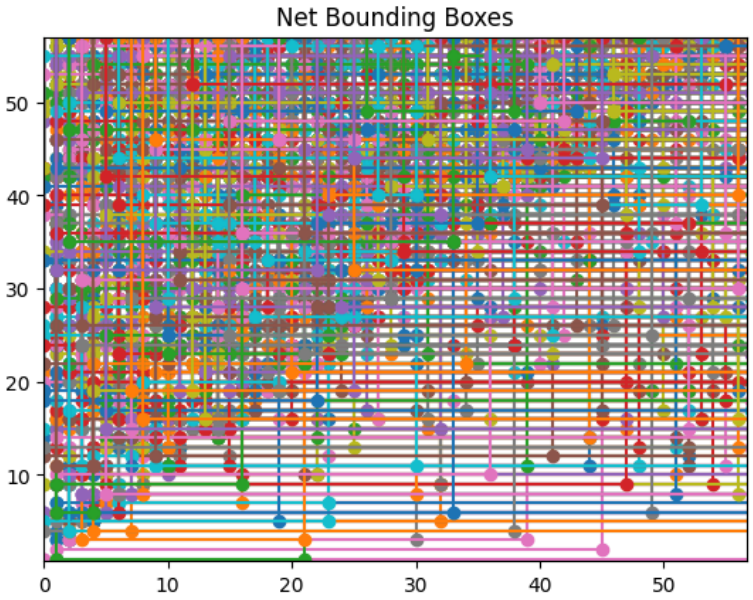


Figure 34. Ibm02 before (top 2) and after (bottom 2) with a total view and zoomed in view.

Its difficult to tell improvement from the plots, however we can see that there is not much difference between the two. This is where we realized we'll need to increase the weight of net length in the cost function so these bounding boxes would have incentive to shrink. Below in Figure 35. we show the output of ibm03 before and after optimization to shows the effectiveness of these weights:



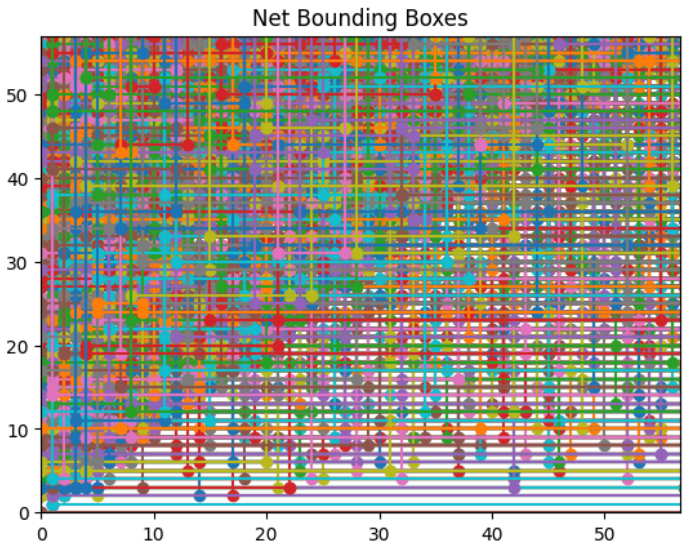


Figure 35. IBM03 with new weights before and after optimization.

As we can see from the plot, the results do not look like much has improved and that we're closer to routability, however the concentration of nets seems a bit more distributed and the lower right corner is more filled than before.

Figure 36. shows a plot of the cost in relation to temperature:

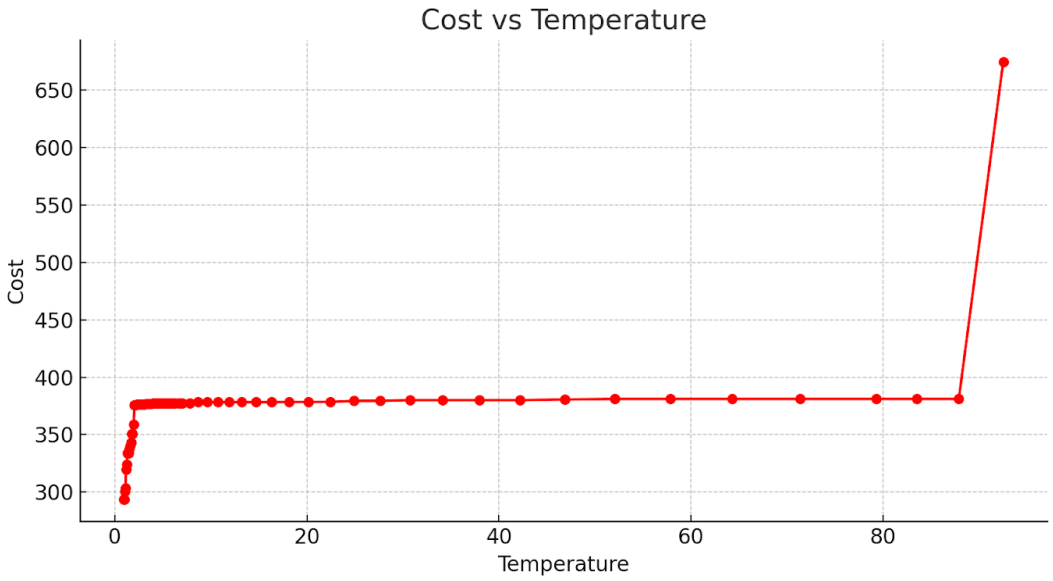


Figure 36. Ibm02 improvement over temperature.

After seeing these results we found it appropriate to retest with temperature at a significantly lower temperature. To investigate the large drop seen when temperature is approximately two. After running the experiment on ibm2 again in Figure 37. we find the same thing happens even after changing temperature, leading us to believe that temperature is not the cause of this drop near the end. The drop in cost is actually less extreme in this case, it only looks worse due to the scaling.

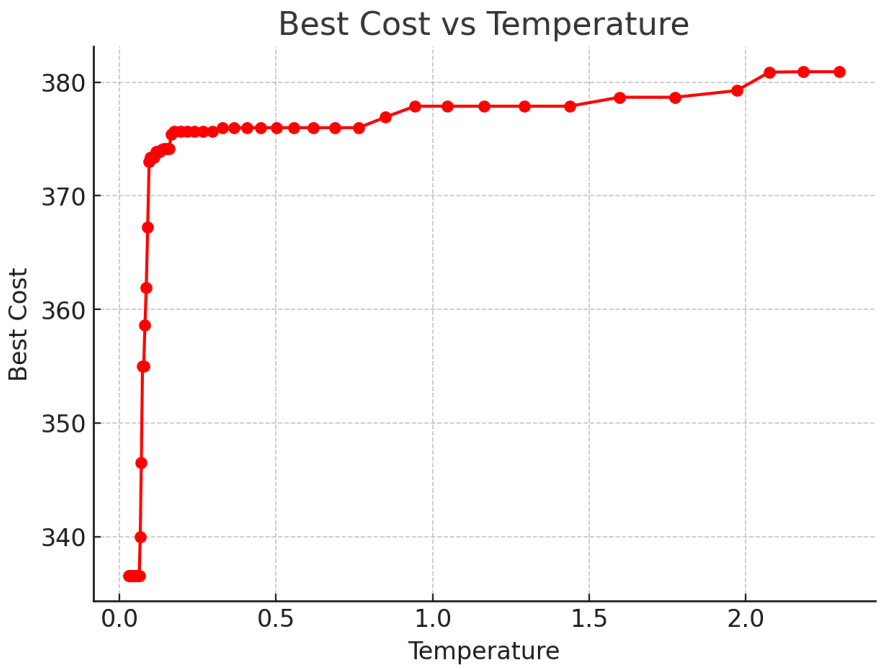


Figure 37. Ibm02 with initial temperature = 2.3

Figure 38. shows proof our temperature scheduling was appropriate:

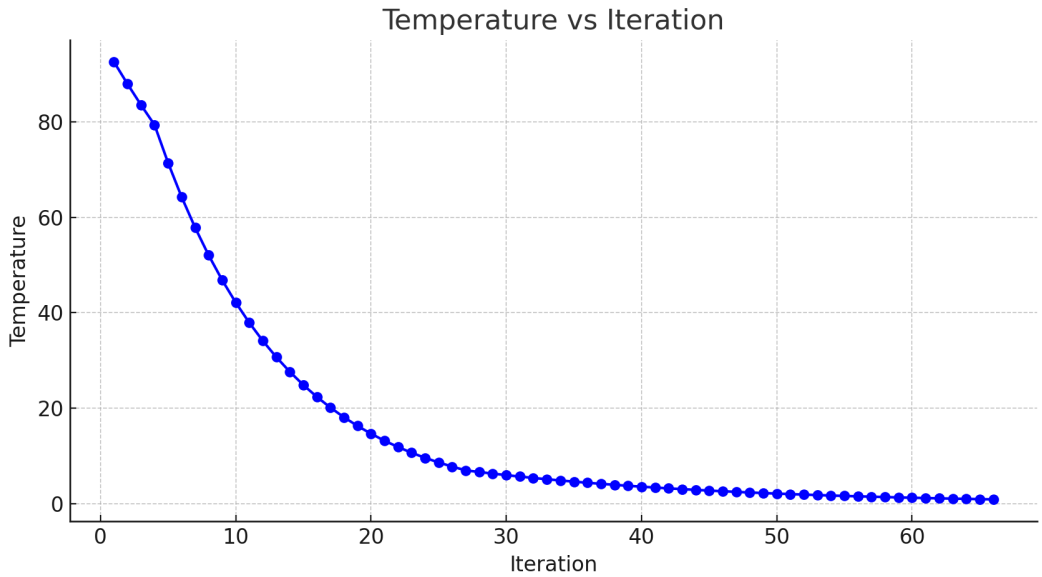


Figure 38. Temperature vs. Iteration

7.3 Using final Hyperparameters on ibm01 - ibm04

We were only able to gather final results on four of the benchmarks. As can be seen in Table 2. runtime has significant increase with an increase in the amount of nodes to be placed. Below are our results for each *ibmxx* benchmark:

Benchmark	Initial Worst Cost	Iter 1 Cost	Final Cost	Improvement %	Runtime (Mins)	Routable?
ibm01	325.42	325.379	274.953	15.50	44.83	No
ibm02	394.887	392.055	337.379	14.56	115.0165	No
ibm03	674.817	308.313	241.279	21.74	168.56	No
ibm04	433.329	433.329	415.599	4.09	207.48	No

Table 2. Benchmark Results

To provide context to the labels in the table, improvement is only based on the difference between the first iteration and the last. We show the initial worst case to show the diversity in the population when the initial population is randomly generated.

7.4 Comparing Results

Al-Kawam and Harmanani [13] actually use a very similar algorithm so our with the following cost function:

$$C = \sum_{i \in Nets} (w_i^H \times X_i + w_i^V \times Y_i) + w_2 \sum_{i \neq j} [O_{ij}]^2 + w_3 \sum_{rows} |l_R - \overline{L_R}| \quad (4)$$

To provide some interpretation, the cost function is the summation of the cost of every net. Like us, they defined the net length as the area of the bounding box of the connections. They add more detail to this however, since there are different weights for the x and y dimensions of their 'modules,' which is how they refer to their version of the bounding box. The next component O_ij, represents the overlap between 'modules.' The main difference between our cost function is that our third components is the length of the critical nets, or nets with most connections, but their third component "indicates unevenness as uneven distribution of row lengths results in wastage of chip area." [13] This actually presents a good idea of not introducing a hard grid size at the very beginning, making minimization of the chip size easier to achieve. It's difficult to compare our results to theirs in figure 39, since they don't present the cost values, but instead show how much faster their program is the larger the benchmark used is. This source does validate that the idea behind our cost function is on the right track.

Sechen and Sangiovanni-Vincentelli [14] provide results on their attempt at implementation of timberwolf algorithm, however it is on standard cell placement rather than FPGA. Their results are shown in figure 40:

TABLE I
TIMBERWOLF STANDARD CELL PLACEMENT OPTIMIZATION PROGRAM

Circuit	# Cells	Total Wire Length Reduction	Final Chip Area Reduction	CPU Time in Hours VAX 780
CktF	2700	66%	57%	84
CktG	1500	**	40%	36
CktA1	1500	45%	30%	20
CktA2	1500	37%	25%	10
CktB	1000	57%	31%	8
CktC	200	41%	15%*	2
CktD	100	37%	15%*	0.5

*pad-limited **not recorded

Figure 40. [14] results.

Here we can see with an increase in the amount of cells, wire length reduction would prove better with every iteration. We did not receive this trend, and were unable to reach a similar improvement percentage. This trend makes sense, as in the more randomly placed nodes are present the larger the total net length will be, but after reducing nets to as small as they can be, each net will have approximately the same size independently of chip size. To elaborate further, if there are two nodes in a net, the optimal solution is to have them placed right next to each other, and this does not depend on chip size at all. Chip size comes into play since larger chip size means more cells, which means less ability to get perfect nets like the one described, however this doesn't make it impossible.

8. Conclusion

After working on and attempting to implement a genetic algorithm ourselves, we've seen that there are a lot of moving parts to what makes the algorithm work. We've also noticed when it comes to exploring the hyperspace of results, it's very easy for the algorithm to quickly converge, and then be trapped with specific 'genes.' It's much like a real life example, where the strongest members of the population will survive past the weaker members. Consequently, these stronger genes will quickly take over the population, allowing for only mutation and crossover to provide some improvement.

We ultimately had several issues that would hinder improvement, and after investigating we'd' improve cost reduction with iterations, however we were unable to fully resolve the issues. These two main issues were reducing overlap, and the spike in decreasing delta C towards the end of the process. If we had been able to identify and resolve the root causes/use them to our advantage, we'd have a much more successful algorithm.

9. Bibliography

[1] Zeidman, Bob. “All about Fpgas.” EE Times, DESIGNLINES, 22 Mar. 2006, www.eetimes.com/all-about-fpgas/.

[2] Ben-Ameur, Walid. “Computing the initial temperature of simulated annealing.” Computational Optimization and Applications, vol. 29, no. 3, Dec. 2004, pp. 369–385, https://doi.org/10.1023/b:coap.0000044187.23143.bd.

[3] Johnson, David S., et al. “Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning.” *Operations Research*, vol. 37, no. 6, Dec. 1989, pp. 865–892, https://doi.org/10.1287/opre.37.6.865.

[4] Roy, Jarrod, and Igor Markov. VLSICAD Page, vlsicad.eecs.umich.edu/BK/ISPD06bench/BookshelfFormat.txt. Accessed 4 Apr. 2024.

[5] Doboli, Alex. “main5.”

[6] Mallawaarachchi, Vijini. “Introduction to Genetic Algorithms - Including Example Code.” Medium, Towards Data Science, 1 Mar. 2020, towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3.

[7] Whitley, Darrell. “A genetic algorithm tutorial.” Statistics and Computing, vol. 4, no. 2, June 1994, https://doi.org/10.1007/bf00175354.

[8] ‘trailmax.’ “Dealing with Duplicates in Genetic Algorithm.” *Theoretical Computer Science Stack Exchange*, 18 August 2011, cstheory.stackexchange.com/questions/7849/dealing-with-duplicates-in-genetic-algorithm.

[9] N. Sherwani. Algorithms for Physical Design Automation. Kluwer Academic Publishers, Boston, MA, 1992

[10] K. Prasad. I. Koren, The Effect of Placement on Yield for Standard Cell Design, University of Massachusetts, Amherst, MA, 2000

[11] E*fficient and Effective Placement for Very Large Circuits ...*, limsk.ece.gatech.edu/course/ece6133/papers/timberwolf.pdf. Accessed 7 Apr. 2024.

[12] J. A. Khan and S. M. Sait, "Fuzzy aggregating functions for multiobjective VLSI placement," 2002 IEEE World Congress on Computational Intelligence. 2002 IEEE International Conference on Fuzzy Systems. FUZZ-IEEE'02. Proceedings (Cat. No.02CH37291), Honolulu, HI, USA,

[13] Al-Kawam, Ahmad, and Haidar M. Harmanani. “A parallel GPU implementation of the timber wolf placement algorithm.” *2015 12th International Conference on Information Technology - New Generations*, Apr. 2015, https://doi.org/10.1109/itng.2015.144.

[14] Sechen, C., and A. Sangiovanni-Vincentelli. “The Timberwolf Placement and routing package.” *IEEE Journal of Solid-State Circuits*, vol. 20, no. 2, Apr. 1985, pp. 510–522, https://doi.org/10.1109/jssc.1985.1052337.

10. Appendix

Objects.hpp

```
#pragma once
#include <iostream>
#include <vector>
#include <map>
#include <set>
#include<memory>
#ifndef DATASTRUCTURES_HPP
#define DATASTRUCTURES_HPP
using namespace std;
// Forward declaration of Node to resolve circular dependency
class Node;
// Full definition of Net
struct Net {
    Net() = default; // Enable default construction
    std::string name; // Name of the net
    std::vector<Node*> Nodes; // Pointers to Nodes connected to this net
    int nodesSize;
    bool isCritical;
    Net(const std::string& name, int w = 0) : name(name), nodesSize(w) {}

};

struct Coords {
    int x, y;
    Coords() = default;
    Coords(int x, int y) : x(x), y(y) {}
    bool operator==(const Coords& other) const {//ChatGPT
        return x == other.x && y == other.y;
    }
    Node* n;
};

struct Bounds {
    std::string name;
    int x1, x2, y1, y2;

    // Default constructor
    Bounds() : name(""), x1(0), x2(0), y1(0), y2(0) {}
```

```
    // Constructor with parameters
    Bounds(const std::string& n, int minX, int minY, int maxX, int maxY)
        : name(n), x1(minX), x2(maxX), y1(minY), y2(maxY) {}
};


class Node {
private:
    std::string name;
    std::vector<Net*> nets;
    int x = 0, y = 0, z = 0;
    bool isTerminalFlag = false; // Renamed to avoid conflict with
isTerminal() function

public:
    Node(); // Default constructor
    Node(std::string name, std::vector<Net*> nets, int x = 0, int y = 0,
bool isTerminal = false);
    ~Node(); // Destructor
    int getX() const;
    int getY() const;
    int getZ() const;
    void setXY(int newX, int newY);
    std::vector<Net*> getNets() const;
    void addNet(Net* net);
    void removeNet(Net* net);
    std::string getName() const;
    bool isTerminal() const; // Fixed return type and made const
    // In Node class in Objects.hpp
    void setTerminal(bool terminal);
    float weight; // Weight attribute for nodes

};


enum class squareType {
    Terminal, //pin e.g. p123 ***in this case the square can hold 2 pins
e.g. [p1, p2]
    Node,//gate e.g. a123
    Routing // space for wires
};

class square {
```

```
private:
    squareType type;
    const Node* node;
    int wires; //count of wires
public:
    square();
    square(squareType type, const Node* n = nullptr, int wires = 0);
    void setType(squareType st);
    squareType getType();
    void incWires(); //increment wire count if wire type
    void decWires(); //dec if wire type
    void setNode(const Node* n); //set node if terminal or gate
    const Node* getNode();
    bool isEmpty();
    friend class Node;
};

class utilGrid {
private:
    vector<vector<square>> grid;
public:
    utilGrid();
    utilGrid(vector<vector<square>> const ogrid);
    void write(int x, int y, square s); //writing a node into a square
    void swap(int x1, int y1, int x2, int y2); //swapping two nodes
    void move(int x1o, int y1o, int x2o, int y2o);//moving a node to an
empty space
    friend class square;
    friend class Grid;
};

class Grid {
private:
    vector<vector<square>> grid;
    utilGrid ug;
    vector<Coords> enodes; //coords of empty nodes in grid
    map<string, Coords> nodeCoords;//map name to coordinates
public:
    Grid();
    Grid(const std::map<std::string, Node>& nodes); // Updated constructor
    Grid(int totalNodes); //empty grid of certain size
    void write(int x, int y, square s);
    void swap(int x1, int y1, int x2, int y2);
```

```
    void move(int x1, int y1, int x2, int y2);
    void mutation(int x1, int y1);
    void smartMutation(int x1, int y1, vector<Bounds> b, map<string, Net>
nets);
    void initialPlacement(const std::map<std::string, Node>& nodes);
    square getSquare(int x, int y); //get square with coordinates
    float calcCost(float const w1, float const w2, float const w3, const
map<string, Net>& nets, bool& routable, int wireConstraint, vector<Bounds>&
bounded) const;
    float updateCost(float const w1, float const w2, float const w3, bool&
routable, int wireConstraint, vector<Bounds>& bounded, bool isSwap, int x1,
int x2, int y1, int y2);
    int getGridX();
    int getGridY();
    void updateEnodes();
    void updateEmpties(int x1, int y1, int x2, int y2, bool isTerminal);
    // New methods for crossover support
    //int getGridSize() const;
    void placeNode(int x, int y, const Node* node);
    bool isNodePlaced(const Node* node) const;
    int getGridSize() const { return grid.size(); }
    vector<vector<square>> getGrid();
    map<string, Coords> getCoords();
    float calculateOverlaps(const vector<Bounds>& bounds, int
wireConstraint, bool& routable) const;
    // If you decide to make crossover a member function
    //static Grid* crossover(const Grid& parent1, const Grid& parent2,
const std::map<std::string, Net>& nets);
    friend class square;
    friend class utilGrid;
};
struct Result {
    Grid g;
    float cost;
    bool routable = false;
    vector<Bounds> bounds;
    Result(Grid g, float cost, bool routable, vector<Bounds> bounds) :
g(g), cost(cost), routable(routable), bounds(bounds) {}
    Result() : cost(0), routable(false) {}
};


#endif // DATASTRUCTURES_HPP
```

Objects.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <map>
#include <cmath>
#include "Objects.hpp"
#include <sstream>
#include <random>
using namespace std;

//NODE CLASS

Node::Node() {
    name = "";
    vector<Net*> netlist;
    nets = netlist;
    x = 0;
    y = 0;
    z = 0;
}

Node::Node(string name, vector<Net*> nets, int x, int y, bool isTerminal) {
    this->name = name;
    this->nets = nets;
    this->x = x;  // Corrected from xcoord
    this->y = y;  // Corrected from ycoord
    this->isTerminalFlag = isTerminal;  // Assuming you have a member
variable isTerminalFlag
}
Node::~Node() {
    // Cleanup if needed
}
std::vector<Net*> Node::getNets() const {
    return nets;
}
void Node::addNet(Net* net) {
    nets.push_back(net);
}
```

```cpp
int Node::getX() const {
    return x; // Assuming 'x' is an integer member variable of Node
}

int Node::getY() const {
    return y; // Assuming 'y' is an integer member variable of Node
}

int Node::getZ() const {
    return z;
}
void Node::setXY(int newX, int newY) {
    this->x = newX;
    this->y = newY;
}

void Node::removeNet(Net* net) {
    nets.erase(std::remove(nets.begin(), nets.end(), net), nets.end());
    for (size_t i = 0; i < nets.size(); ++i) {
        if (nets[i] == net) {
            nets.erase(nets.begin() + i);
            break;
        }
    }
}

string Node::getName() const {
    if (this->name.empty() || x < -33 || y < -33) {
        cout << "uh oh" << endl;
        return "uh oh";
    }
    else {
        return this->name;
    }

}


bool Node::isTerminal() const {
    return name[0] == 'p';
}

void Node::setTerminal(bool terminal) {
```

```
    this->isTerminalFlag = terminal;
}


//SQUARE CLASS

square::square() {
    type = squareType::Node;
    node = nullptr;
    wires = 0;
}

square::square(squareType type, const Node* n, int wires) {
    this->type = type;
    node = n;
    this->wires = wires;
}

void square::setType(squareType s) {
    type = s;
}

squareType square::getType() {
    return type;
}

void square::incWires() {
    wires++;
}

void square::decWires() {
    wires--;
}

void square::setNode(const Node* n) {
    node = n;
}

const Node* square::getNode() {
    return node;
}

bool square::isEmpty() {
```

```
    if (node == nullptr) return true;
    else return false;
}


//UTILGRID CLASS

utilGrid::utilGrid() {

}

utilGrid::utilGrid(vector<vector<square>> ogrid) {//chatGPT aided
    int n = ogrid.size();
    std::vector<vector<square>> spacedMatrix(2 * n - 1, vector<square>(2 *
n - 1));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            spacedMatrix[2 * i][2 * j] = ogrid[i][j];
        }
    }
    for (int i = 0; i < (n * 2) - 1; ++i) {
        for (int j = 0; j < (n * 2) - 1; ++j) {
            if (!(spacedMatrix[i][j].getType() == squareType::Node ||
spacedMatrix[i][j].getType() == squareType::Terminal)) {
                square s = square(squareType::Routing, 0);
                spacedMatrix[i][j] = s;
            }
        }
    }
    grid = spacedMatrix;
}

void utilGrid::write(int x, int y, square s) {
    if (x > grid.size() || y > grid[0].size()) cout << "Write dims outside
of grid." << endl;
    else grid[x][y] = s;
}

void utilGrid::swap(int x1o, int y1o, int x2o, int y2o) {
    int x1 = x1o * 2;
    int x2 = x2o * 2;
    int y1 = y1o * 2;
```

```
    int y2 = y2o * 2;
    if ((grid[x1][y1].getType() == squareType::Node &&
grid[x2][y2].getType() == squareType::Node) || (grid[x1][y1].getType() ==
squareType::Terminal && grid[x2][y2].getType() == squareType::Terminal)) {
// if appropiate type and matching
        if (x1 > grid.size() || y1 > grid[0].size() || x2 > grid.size() ||
y2 > grid[0].size()) cout << "Swap dims outside of grid." << endl;
        else {
            square temp = grid[x1][y1];
            grid[x1][y1] = grid[x2][y2];
            grid[x2][y2] = temp;
        }
    }
    else if (grid[x1][y1].getType() == squareType::Node ||
grid[x2][y2].getType() == squareType::Node || grid[x1][y1].getType() ==
squareType::Terminal || grid[x2][y2].getType() == squareType::Terminal)
cout << "Error: Trying to swap non-matching types." << endl;
    else cout << "Error: Trying to swap either Blank or Routing type
square." << endl;
}

void utilGrid::move(int x1o, int y1o, int x2o, int y2o) {
    int x1 = x1o * 2;
    int x2 = x2o * 2;
    int y1 = y1o * 2;
    int y2 = y2o * 2;

    if ((grid[x1][y1].getType() == squareType::Node &&
grid[x2][y2].getType() == squareType::Node) || (grid[x1][y1].getType() ==
squareType::Terminal && grid[x2][y2].getType() == squareType::Terminal)) {
// if one is node and the other is empty
        if (x1 > grid.size() || y1 > grid[0].size() || x2 > grid.size() ||
y2 > grid[0].size()) cout << "movement dims outside of grid." << endl;
        else {
            if (grid[x2][y2].getNode() == nullptr) { grid[x2][y2] =
grid[x1][y1]; }
            else if (grid[x1][y1].getNode() == nullptr) { grid[x1][y1] =
grid[x2][y2]; }
        }
    }
    else if (grid[x1][y1].getType() == squareType::Node ||
grid[x2][y2].getType() == squareType::Node || grid[x1][y1].getType() ==
squareType::Terminal || grid[x2][y2].getType() == squareType::Terminal)
```

```
cout << "Error: Trying to move non-matching types." << endl;
}

//GRID CLASS

Grid::Grid() {
    // Initialization or other actions
}
// Adjust Grid constructor to automatically calculate dimensions and
perform initial placement
Grid::Grid(const std::map<std::string, Node>& nodes) {
    int totalNodes = nodes.size();
    // Estimate grid size: square root of total nodes times a factor (>1)
for spacing, rounded up
    int gridSize = ceil(sqrt(totalNodes) * 1.1); // Adjust the 1.1 factor
as needed

    // Initialize an empty grid
    grid = vector<vector<square>>(gridSize, vector<square>(gridSize,
square()));

    // Perform initial placement
    initialPlacement(nodes);
}

Grid::Grid(int totalNodes) {
    int gridSize = ceil(sqrt(totalNodes) * 1.1);
    grid = vector<vector<square>>(gridSize, vector<square>(gridSize,
square()));
}

void Grid::write(int x, int y, square s) {
    if (x >= 0 && x < grid.size() && y >= 0 && y < grid[x].size()) {
        grid[x][y] = s;
        Coords c(x, y);
        nodeCoords[s.getNode()->getName()] = c;
        if (2 * x < ug.grid.size() && 2 * y < ug.grid[0].size()) { //
Assuming ug.grid is public; adjust if it's private
            ug.write(x * 2, y * 2, s);
        }
    }
    else {
        // Handle out-of-bounds access appropriately
```

```
    }
}

void Grid::updateEmpties(int x1, int y1, int x2, int y2, bool isTerminal) {
    Coords a(x2, y2);
    Coords b(x1, y1);
    enodes.push_back(b);
    grid[x2][y2].setNode(nullptr);
    for (auto it = enodes.begin(); it != enodes.end(); ++it) {
        if ((*it).x == a.x && (*it).y == a.y) {
            enodes.erase(it);
            break; // Optional, if you know there is only one element with
the value 3
        }
    } //somtimes causes errors
}

void Grid::updateEnodes() {
    enodes.clear();
    for (int i = 1; i < (grid.size() - 1); i++) {
        for (int j = 1; j < (grid[0].size() - 1); j++) {
            if (grid[i][j].getNode() == nullptr) {
                Coords c(i, j);
                enodes.push_back(c);
            }
        }
    }

}

void Grid::move(int x1, int y1, int x2, int y2) {
    if (grid[x1][y1].getType() == squareType::Terminal ||
grid[x2][y2].getType() == squareType::Terminal) {
        cout << "Cannot cast move on terminal nodes." << endl;
    }
    else if (grid[x1][y1].getType() == squareType::Node &&
grid[x2][y2].getType() == squareType::Node) {
        if (x1 > grid.size() || y1 > grid[0].size() || x2 > grid.size() ||
y2 > grid[0].size()) cout << "Trying to Move out of bounds" << endl;
        else  if (grid[x2][y2].getNode() == nullptr) {
            if (grid[x1][y1].getNode() != nullptr) {
                nodeCoords[grid[x1][y1].getNode()->getName()].x = x2;
                nodeCoords[grid[x1][y1].getNode()->getName()].y = y2;
```

```
            }
            grid[x2][y2] = grid[x1][y1];
            grid[x1][y1].setNode(nullptr);
            updateEmpties(x1, y1, x2, y2, grid[x1][y1].getType() ==
squareType::Terminal);
        }
        else if (grid[x1][y1].getNode() == nullptr) {
            nodeCoords[grid[x2][y2].getNode()->getName()].x = x1;
            nodeCoords[grid[x2][y2].getNode()->getName()].y = y1;
            grid[x1][y1] = grid[x2][y2];
            grid[x2][y2].setNode(nullptr);

            updateEmpties(x2, y2, x1, y1, grid[x1][y1].getType() ==
squareType::Terminal);
        }
        else cout << "Trying to move to none empty Node square" << endl;
    }
}

void Grid::swap(int x1, int y1, int x2, int y2) {
    if (grid[x1][y1].getType() == squareType::Terminal ||
grid[x2][y2].getType() == squareType::Terminal) {
        cout << "Cannot cast move on terminal nodes." << endl;
    }
    else if (grid[x1][y1].getType() == squareType::Node &&
grid[x2][y2].getType() == squareType::Node) {
        if (x1 > grid.size() || y1 > grid[0].size() || x2 > grid.size() ||
y2 > grid[0].size()) cout << "Trying to Swap out of bounds" << endl;
        else if (grid[x2][y2].getNode() == nullptr ||
grid[x1][y1].getNode() == nullptr) cout << "Trying to swap with a nullptr"
<< endl;
        else {
            if (grid[x2][y2].getNode() != nullptr) {
                nodeCoords[grid[x2][y2].getNode()->getName()].x = x1;
                nodeCoords[grid[x2][y2].getNode()->getName()].y = y1;
            }
            if (grid[x1][y1].getNode() != nullptr) {
                nodeCoords[grid[x1][y1].getNode()->getName()].x = x2;
                nodeCoords[grid[x1][y1].getNode()->getName()].y = y2;
            }
            square temp = grid[x1][y1];
            grid[x1][y1] = grid[x2][y2];
            grid[x2][y2] = temp;
```

```
            //ug.swap(x1, y1, x2, y2);
        }
    }
}

void Grid::smartMutation(int x1, int y1, vector<Bounds> bo, map<string,
Net> nets) {
    int ran = rand() % 2;
    int rand_x = 0;
    int rand_y = 0;
    int ran_i = 0;
    Bounds x;
    bool flag = false;
    if (grid[x1][y1].getNode() == nullptr) {
        flag = true;
    }
    if (flag) {
        mutation(x1, y1);
        return;
    }
    for (Bounds b : bo) {
        for (auto net : grid[x1][y1].getNode()->getNets()) {
            if (b.name == net->name) {
                x = b;
                exit;
            }
        }


    }


    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> xdistribution(x.x1, x.x2);
    std::uniform_int_distribution<int> ydistribution(x.y1, x.y2);
    if (x.x2 > 125 || x.y2 > 125) {
        cout << "Crazy Bounds" << endl;
    }
    if (ran % 2 == 0) { //Even will use the swap function
        rand_x = xdistribution(gen);
        rand_y = ydistribution(gen);
```

```
            swap(x1, y1, rand_x, rand_y);
        }
        else {  //Odd will use the move function
            for (auto c : enodes) {
                if (c.x < x.x2 && c.x > x.x1 && c.y < x.y2 && c.y > x.y1) {
                    if (c.x > 125 || c.y > 125) {
                        cout << "Crazy Coordinate" << endl;
                    }
                    move(x1, y1, c.x, c.y);
                    return;
                }
            }
            ran_i = rand() % enodes.size();
            rand_x = enodes.at(ran_i).x;
            rand_y = enodes.at(ran_i).y;
            move(x1, y1, rand_x, rand_y);
        }
    }

    void Grid::mutation(int x1, int y1) {
        int ran = rand() % 2;
        int rand_x = 0;
        int rand_y = 0;
        int ran_i = 0;

        if (ran % 2 == 0) { //Even will use the swap function
            rand_x = rand() % grid.size();
            rand_y = rand() % grid[0].size();
            swap(x1, y1, rand_x, rand_y);
        }
        else {  //Odd will use the move function
            ran_i = rand() % enodes.size();
            rand_x = enodes.at(ran_i).x;
            rand_y = enodes.at(ran_i).y;
            move(x1, y1, rand_x, rand_y);
        }
    }

    square Grid::getSquare(int x, int y) {
        return grid[x][y];
    }

    void Grid::initialPlacement(const std::map<std::string, Node>& nodes) {
```

```
    // Adjust the coordinate system to start from 0,0 if minimum is -33.
    int coordinateShift = 33; // Assuming -33 is the minimum coordinate.
    int maxX = 0, maxY = 0, minX = 0, minY = 0;
    // Containers for edge terminals.
    std::vector<const Node*> topEdge, bottomEdge, leftEdge, rightEdge,
isTerminal;
    // For random placement
    std::mt19937 rng{ std::random_device{}() };
    std::set<std::pair<int, int>> occupiedPositions;

    for (auto pair : nodes) {
        if (pair.second.isTerminal()) {
            isTerminal.push_back(&nodes.at(pair.first));
        }
    }

    auto maxElementX = max_element(isTerminal.begin(), isTerminal.end(),
[](const Node* a, const Node* b) {//ChatGPT "how to find a max struct in a
vector by its int value"
        return a->getX() < b->getX();
        });
    maxX = (*maxElementX)->getX();
    auto maxElementY = max_element(isTerminal.begin(), isTerminal.end(),
[](const Node* a, const Node* b) {
        return a->getY() < b->getY();
        });
    maxY = (*maxElementY)->getY();
    auto minElementX = min_element(isTerminal.begin(), isTerminal.end(),
[](const Node* a, const Node* b) {
        return a->getX() < b->getX();
        });
    minX = (*minElementX)->getX();
    auto minElementY = min_element(isTerminal.begin(), isTerminal.end(),
[](const Node* a, const Node* b) {
        return a->getY() < b->getY();
        });
    minY = (*minElementY)->getY();

    for (auto pair : nodes) {
        Node& node = pair.second;
        if (node.isTerminal()) { // Corrected to use function call syntax
            // Determine the edge for each terminal using getters
            if (node.getY() == maxY)
```

```
topEdge.push_back(&nodes.at(pair.first)); // Top edge
            else if (node.getY() == minY)
bottomEdge.push_back(&nodes.at(pair.first)); // Bottom edge
            else if (node.getX() == minX)
leftEdge.push_back(&nodes.at(pair.first)); // Left edge
            else if (node.getX() == maxX)
rightEdge.push_back(&nodes.at(pair.first)); // Right edge
        }
    }

    auto distributeTerminals = [&](const std::vector<const Node*>&
edgeTerminals, char edge) {
        int numTerminals = edgeTerminals.size();
        int spacing = (edge == 't' || edge == 'b') ? grid[0].size() /
(numTerminals + 1)
            : grid.size() / (numTerminals + 1);
        for (int i = 0; i < numTerminals; ++i) {
            int pos = 0;
            if (spacing == 1)  pos = ((i + 1) * spacing);
            else  pos = ((i + 1) * spacing) - 1; //want to start at index 1
no terminals at corners
            int x = 0, y = 0;
            if (edge == 't') { x = pos; y = 0; }
            else if (edge == 'b') { x = pos; y = grid.size() - 1; }
            else if (edge == 'l') { x = 0; y = pos; }
            else if (edge == 'r') { x = grid[0].size() - 1; y = pos; }
            write(y, x, square(squareType::Terminal, edgeTerminals[i]));
        }
    };

    distributeTerminals(topEdge, 't');
    distributeTerminals(bottomEdge, 'b');
    distributeTerminals(leftEdge, 'l');
    distributeTerminals(rightEdge, 'r');

    // Place non-terminal nodes randomly
    std::uniform_int_distribution<int> distX(1, grid.size() - 2), distY(1,
grid[0].size() - 2);

    for (auto pair : nodes) {
        auto& node = pair.second;
        if (!node.isTerminal()) {
            bool placed = false;
```

```
            while (!placed) {
                int randomX = distX(rng);
                int randomY = distY(rng);
                if (occupiedPositions.find({ randomX, randomY }) ==
occupiedPositions.end()) {
                    // If position is not occupied, place the node
                    write(randomX, randomY, square(squareType::Node,
&nodes.at(node.getName())));
                    occupiedPositions.insert({ randomX, randomY });
                    placed = true;
                }
            }
        }
    }

    for (int i = 1; i < grid.size() - 1; i++) {
        for (int j = 1; j < grid[0].size() - 1; j++) {
            if (grid[i][j].getNode() == nullptr) {
                Coords c(i, j);
                enodes.push_back(c);
            }
        }
    }
}

float Grid::calcCost(float const w1, float const w2, float const w3, const
map<string, Net>& nets, bool& routable, int wireConstraint, vector<Bounds>&
bounded) const {
    float totalCost = 0, totalLength = 0, overlapCount = 0, critCost = 0;
    bounded.clear(); // Clear previous bounds

    // Process each net to calculate wirelength and critical net cost
    for (const auto& [netName, net] : nets) {
        if (net.Nodes.empty()) continue;

        int xmin = INT_MAX, xmax = INT_MIN, ymin = INT_MAX, ymax = INT_MIN;

        // Determine the bounding box for each net
        for (const auto* nodePtr : net.Nodes) {
            const auto& coords = nodeCoords.at(nodePtr->getName());
            xmin = std::min(xmin, coords.x);
            xmax = std::max(xmax, coords.x);
            ymin = std::min(ymin, coords.y);
```

```
            ymax = std::max(ymax, coords.y);
        }

        // Calculate total wirelength for the net
        int wirelength = (xmax - xmin) + (ymax - ymin);
        totalLength += wirelength;

        // Add additional cost for critical nets
        if (net.isCritical) {
            critCost += wirelength * 0.5; // 50% additional cost for
critical nets
        }

        Bounds bounds{ netName, xmin, ymin, xmax, ymax };
        bounded.push_back(bounds);
    }

    // Calculate overlaps and update routability
    overlapCount = calculateOverlaps(bounded, wireConstraint, routable);

    // Normalize cost components
    float gridArea = static_cast<float>(grid.size() * grid[0].size());
    float normalizedLength = totalLength / gridArea;
    float normalizedOverlap = overlapCount / (nets.size() * gridArea);
    float normalizedCritCost = critCost / gridArea;

    // Compute total cost
    totalCost = w1 * normalizedLength + w2 * normalizedOverlap + w3 *
normalizedCritCost;

    std::cout << "Total Cost:" << totalCost << ", Routable: " << (routable
? "Yes" : "No") << std::endl;
    return totalCost;
}

float Grid::calculateOverlaps(const vector<Bounds>& bounds, int
wireConstraint, bool& routable) const {
    float overlapCount = 0;
    routable = true;

    for (size_t i = 0; i < bounds.size(); ++i) {
        for (size_t j = i + 1; j < bounds.size(); ++j) {
            int x_overlap = std::max(0, std::min(bounds[i].x2,
```

```
bounds[j].x2) - std::max(bounds[i].x1, bounds[j].x1));
            int y_overlap = std::max(0, std::min(bounds[i].y2,
bounds[j].y2) - std::max(bounds[i].y1, bounds[j].y1));
            float overlapArea = x_overlap * y_overlap;

            if (overlapArea > 0) {
                overlapCount += overlapArea;
                if (--wireConstraint < 0) routable = false;
            }
        }
    }

    return overlapCount;
}


float updateCost(float const w1, float const w2, float const w3, bool&
routable, int wireConstraint, vector<Bounds>& bounded, bool isSwap, int x1,
int x2, int y1, int y2) {
    return 0;
}

void Grid::placeNode(int x, int y, const Node* node) {
    if (x >= 0 && x < grid.size() && y >= 0 && y < grid[0].size()) {
        grid[x][y].setNode(node);
        Coords c(x, y);
        nodeCoords[node->getName()] = c;
    }
    else {
        // Handle the error: position out of bounds
        std::cout << "Error: Position (" << x << ", " << y << ") is out of
bounds for placing a node.\n";
    }
}

bool Grid::isNodePlaced(const Node* node) const {
    for (vector<square> row : grid) {
        for (auto& square : row) {
            if (square.getNode() == node) return true;
        }
    }
    return false;
}
```

```
int Grid::getGridY() {
    return grid.size();
}

int Grid::getGridX() {
    return grid.at(0).size();
}

vector<vector<square>> Grid::getGrid() {
    return grid;
}

map<string, Coords> Grid::getCoords() {
    return nodeCoords;
}
```

Scanner.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <map>
#include <random>
#include <set>
#include <cmath>
#include <thread>
#include <mutex>
#include "Objects.hpp"
#include <sstream> // Add this at the top of Scanner.cpp

using namespace std;

void read(const string netfile, const string nodefile, const string plfile,
map<string, Node>& nodes, map<string, Net>& nets, int& numNets, int&
numPins, int& numNodes, int& numTerminals) {
	nodes.clear();
	nets.clear();

	string line;
	fstream myFile;

	//NODE FILE
	int iter = 0;
	myFile.open(nodefile);
	if (myFile.is_open()) {
		while (getline(myFile, line)) {
			if (iter < 5) iter++;
			else {
				vector<string> words;
				string temp;
				stringstream ss(line);

				while (ss >> temp) {
					words.push_back(temp); //delimits string into
words by spaces
				}
```

```
                        if (words[0] == "NumNodes") numNodes =
stoi(words[2]);
                        else if (words[0] == "NumTerminals") numTerminals =
stoi(words[2]);
                        else {
                              vector<Net*> newlist;
                              Node newnode(words[0], newlist);
                              nodes[words[0]] = newnode;
                        }
                  }
            }
            myFile.close();
      }
      else {
            cout << "Failed to open node file." << endl;
      }


      //NETFILE
      myFile.open(netfile);
      iter = 0;
      int netcount = 0;
      if (myFile.is_open()) {
            while (getline(myFile, line)) {
                  if (iter < 5) {
                        iter++;
                        continue;
                  }
                  else {
                        vector<string> words;
                        string temp;
                        stringstream ss(line);

                        while (ss >> temp) {
                              words.push_back(temp); //delimits string into
words by spaces
                        }

                        if (words[0] == "NumNets") numNets =
stoi(words[2]);
                        else if (words[0] == "NumPins") numPins =
stoi(words[2]);
                        else if (words[0] == "NetDegree") {
```

```
                              Net newnet("n" + to_string(netcount));
                              nets[newnet.name] = newnet;
                              nets[newnet.name].isCritical = false;

                              netcount++;
                        }
                        else {
                              nets["n" + to_string(netcount -
1)].Nodes.push_back(&nodes[words[0]]);
                              nets["n" + to_string(netcount -
1)].nodesSize++;
                              nodes[words[0]].addNet(&nets["n" +
to_string(netcount - 1)]);
                        }
                  }
            }
            myFile.close();
      }
      else cout << "Failed to open net file." << endl;

      //read plfile
      myFile.open(plfile);
      iter = 0;
      if (myFile.is_open()) {
            while (getline(myFile, line)) {
                  if (iter < 6) {
                        iter++;
                        continue;
                  }
                  else {
                        vector<string> words;
                        string temp;
                        stringstream ss(line);

                        while (ss >> temp) {
                              words.push_back(temp); //delimits string into
words by spaces
                        }

                        nodes[words[0]].setXY(stoi(words[1]),
stoi(words[2]));

                  }
```

```
            }
            myFile.close();
      }
}
bool sortByValueDescending(const pair<string, Net>& a, const pair<string,
Net>& b) {
      return a.second.nodesSize > b.second.nodesSize;
}

void findTop10Percent(map<string, Net>& inputMap) {
      if (inputMap.empty()) {
            cout << "Input map is empty." << endl;
            return;
      }

      size_t top10PercentSize = ceil(inputMap.size() * 0.1);
      top10PercentSize = max(top10PercentSize, size_t(1)); // Ensure at
least one element is considered

      vector<pair<string, Net*>> vec; // Use pointers to avoid copying and
to modify original map objects
      for (auto& pair : inputMap) {
            vec.emplace_back(pair.first, &pair.second);
      }

      // Partially sort to find the top elements only
      nth_element(vec.begin(), vec.begin() + top10PercentSize, vec.end(),
            [](const auto& a, const auto& b) { return a.second->nodesSize >
b.second->nodesSize; });

      cout << "Top 10% elements:" << endl;
      for (size_t i = 0; i < top10PercentSize; ++i) {
            vec[i].second->isCritical = true;
            cout << "Net: " << vec[i].first << " is marked as critical." <<
endl;
      }
}
bool isNumeric(const std::string& str) {
      return !str.empty() && std::all_of(str.begin(), str.end(), [](char c)
{ return std::isdigit(c) || c == '-'; });
}
```

```
vector<Result> createInitialGrids(const std::map<std::string, Node>& nodes,
int k, float const w1, float const w2, float const w3, map<string, Net>
const nets, int wireConstraint) {
	vector<Result> init;
	std::random_device rd;
	std::mt19937 g(rd());

	for (int i = 0; i < k; ++i) {
		Grid grid(nodes); // Use `grid` instead of `g` to avoid
confusion with `std::mt19937 g`
		bool routable = false;
		vector<Bounds> bounds;
		bounds.reserve(50000);
		float cost = grid.calcCost(w1, w2, w3, nets, routable,
wireConstraint, bounds);
		init.emplace_back(std::move(grid), cost, routable,
std::move(bounds));
	}
	cout << "Finished creating Initial Grids" << endl;
	return init;
}

Result bestCost(vector<Result> results) {
	Result best;
	double mincost = results.at(0).cost;
	if (results.at(0).routable) return results.at(0);
	for (int i = 0; i < results.size(); i++) {
		if (results.at(i).routable) {
			return results.at(i);
		}
		else if (results.at(i).cost <= mincost) {
			mincost = results.at(i).cost;
			best = results.at(i);
		}
	}
	return best;
}

void tryPlaceNode(const Node* node, int i, int j, set<std::string>&
placedNodeNames, Grid& child, set<string>& toBePlaced) {
	if (!node || placedNodeNames.count(node->getName())) return; //
Already placed
	// Find a position for the node
```

```
	if (child.getSquare(i, j).getNode() == nullptr) { // Position is
empty
		if (node->getName().empty()) {
			cout << "WARNING" << endl;
		}
		child.placeNode(i, j, node);

		placedNodeNames.insert(node->getName());
		auto it = toBePlaced.find(node->getName());
		if (it != toBePlaced.end()) toBePlaced.erase(it);

		return;
	}
}

Grid crossover(Grid* parent1, Grid* parent2, const std::map<std::string,
Net>& nets, const std::map<std::string, Node> nodes) {
	Grid child = Grid(nodes.size());

	std::random_device rd;
	std::mt19937 gen(rd());
	std::uniform_int_distribution<> dis(0, parent1->getGridSize() - 1);

	int crossoverPoint = dis(gen);
	std::set<std::string> placedNodeNames; // To track which nodes have
been placed
	set<string> toBePlaced;
	for (auto node : nodes) {
		toBePlaced.insert(node.second.getName());
	}
	// First pass: place nodes from both parents up to crossover point
	for (int i = 0; i <= crossoverPoint; ++i) {
		for (int j = 0; j < child.getGridY(); j++) {
			if (parent1->getSquare(i, j).getNode() != nullptr) {
				Node* node = new Node;
				*node =
nodes.at(parent1->getGrid()[i][j].getNode()->getName());
				tryPlaceNode(node, i, j, placedNodeNames, child,
toBePlaced);
			}
		}
	}
	for (int i = crossoverPoint + 1; i < child.getGridSize(); i++) {
```

```
            for (int j = 0; j < child.getGridY(); j++) {
                  if (parent2->getSquare(i, j).getNode() != nullptr) {
                        Node* node = new Node;
                        *node =
nodes.at(parent2->getGrid()[i][j].getNode()->getName());
                        tryPlaceNode(node, i, j, placedNodeNames, child,
toBePlaced);
                  }
            }
      }
      /*
      // Second pass: ensure all nodes from both parents are considered
      for (int i = 0; i < child.getGridX(); ++i) {
            for (int j = 0; j < child.getGridY(); j++) {
                  if (parent2->getSquare(i, j).getNode() != nullptr)
tryPlaceNode(parent2->getSquare(i, j).getNode(), i, j, placedNodeNames,
child);
            }
      }
      for (int i = 0; i < child.getGridX(); i++) {
            for (int j = 0; j < child.getGridY(); j++) {
                  if (parent1->getSquare(i, j).getNode() != nullptr)
tryPlaceNode(parent1->getSquare(i, j).getNode(), i, j, placedNodeNames,
child);
            }
      }
      */
      for (int i = 0; i < child.getGridX(); i++) {
            for (int j = 0; j < child.getGridY(); j++) {
                  if (child.getSquare(i, j).getNode() == nullptr &&
!toBePlaced.empty()) {
                        Node* node = new Node;
                        *node = nodes.at(*toBePlaced.begin());
                        tryPlaceNode(node, i, j, placedNodeNames, child,
toBePlaced);
                  }
            }
      }

      child.updateEnodes();

      return child;
}
```

```
void exportForVisualization(Result r, const std::string& filename) {
	std::ofstream file(filename); // Open the file for writing
	if (!file.is_open()) { // Check if the file was opened successfully
		std::cerr << "Error: Unable to open file " << filename << " for
writing." << std::endl;
		return;
	}
	// Write the header to the file
	file << "Net,Xmin,Ymin,Xmax,Ymax\n";
	// Write the bounds data to the file
	for (auto& b : r.bounds) {
		// Assume bounds are calculated and stored somewhere accessible
		file << b.name << "," << b.x1 << "," << b.y1 << "," << b.x2 <<
"," << b.y2 << "\n";
	}
	file.close(); // Close the file
}

void performCrossoversThread(std::vector<Result>& offspring,
vector<Result>& parents, map<std::string, Net> nets, int startIdx, int
endIdx, std::mutex& offspringMutex, map<string, Node> nodes, float w1,
float w2, float w3, int wireConstraint) {
	for (int i = startIdx; i < endIdx && (i + 1) < parents.size(); i +=
2) {
		// Perform crossover on parents[i] and parents[i+1]
		Grid child = crossover(&parents[i].g, &parents[i + 1].g, nets,
nodes); // Ensure your crossover function is thread-safe.
		bool rout = true;
		std::lock_guard<mutex> lock(offspringMutex); // Protecting
shared access to the offspring vector.
		vector<Bounds> b;
		float cost = child.calcCost(w1, w2, w3, nets, rout,
wireConstraint, b);
		Result r(child, cost, rout, b);
		offspring.push_back(r);
	}
}

void multithreadedCrossover(vector<Result>& offspring, vector<Result>&
parents, map<std::string, Net> nets, unsigned int numThreads,
map<std::string, Node> nodes, float w1, float w2, float w3, int
wireConstraint) {
```

```
	std::vector<std::thread> threads; // Thread vector
	std::mutex offspringMutex; // Mutex for protecting access to the
offspring vector

	int segmentSize = parents.size() / numThreads; // Calculate the
segment size per thread

	for (unsigned int i = 0; i < numThreads; ++i) {
		int startIdx = i * segmentSize;
		int endIdx = (i == numThreads - 1) ? parents.size() : (i + 1) *
segmentSize; // Adjust the last segment to cover all remaining parents

		// Launch a thread for each segment of the parents vector
		threads.emplace_back([&, startIdx, endIdx]() {
			performCrossoversThread(offspring, parents, nets,
startIdx, endIdx, offspringMutex, nodes, w1, w2, w3, wireConstraint);
			});
	}

	// Wait for all threads to complete their tasks
	for (std::thread& t : threads) {
		if (t.joinable()) {
			t.join();
		}
	}
}

bool compareByFloat(const Result& a, const Result& b) { //ChatGPT
	return a.cost < b.cost; // Change to < for ascending order
}

vector<Result> tournamentSelection(std::vector<Result> population, const
std::map<std::string, Net>& nets, float percentage) { //ChatGPT
	std::random_device rd;
	std::mt19937 gen(rd());
	std::uniform_int_distribution<> dist(0, population.size() - 1);

	int topKNum = population.size() * percentage;
	vector<Result> topFive = population;
	sort(topFive.begin(), topFive.end(), compareByFloat);
	auto h = topFive.begin();
	advance(h, topKNum);
	vector<Result> newTopFive(topFive.begin(), h);
```

```
	for (auto i : newTopFive) {
		cout << "Selected member with cost: " << i.cost << endl;
	}
	return newTopFive;
}

vector<Result> perturb(std::vector<Result>& population, map<std::string,
Net>& nets, float w1, float w2, float w3, int wireConstraint, map<string,
Node> nodes, float selectP, float crossP, float mutP) {
	std::vector<Result> nextGeneration;
	std::random_device rd;
	std::mt19937 gen(rd());

	int count = 0; //to keep track of how many offspring created already

	//Selection:
	nextGeneration = tournamentSelection(population, nets, selectP);
	count += nextGeneration.size();
	//Crossover
	int cCount = population.size() * crossP;
	vector<Result> crossoverOffspring;
	//multithreadedCrossover(crossoverOffspring, nextGeneration, nets,
cCount, nodes, w1, w2, w3, wireConstraint);

	for (int i = 0; i < cCount; ++i) {
		int i1 = rand() % count;
		int i2 = 0;
		while (i2 != i1) {
			i2 = rand() % count;
		}

		Grid* parent1 = &nextGeneration[i1].g;
		Grid* parent2 = &nextGeneration[i2].g;
		Grid child = crossover(parent1, parent2, nets, nodes);
		bool rout = false;
		vector<Bounds> b;
		float cost = child.calcCost(w1, w2, w3, nets, rout,
wireConstraint, b);
		Result r(child, cost, rout, b);
		nextGeneration.push_back(r);

	}
```

```
	count += cCount;
	//Mutation

	for (int i = count; i < population.size(); i++) {
		int in = rand() % count;
		Grid* copy = &nextGeneration[in].g;
		for (int i = 0; i < (nodes.size() * 0.1); i++) { //1 mutation
has a miniscule effect on cost, changed to avoid early convergence
			std::uniform_int_distribution<> distx(1, copy->getGridX()
- 2);
			std::uniform_int_distribution<> disty(1, copy->getGridY()
- 2);
			int rx = distx(gen);
			int ry = disty(gen);
			copy->smartMutation(rx, ry, nextGeneration[in].bounds,
nets);
			//copy->mutation(rx, ry);
		}
		//copy->mutation(rx, ry);
		bool rout = false;
		vector<Bounds> b;
		float cost = copy->calcCost(w1, w2, w3, nets, rout,
wireConstraint, b);
		Result r(*copy, cost, rout, b);
		nextGeneration.push_back(r);
	}
	return nextGeneration;
}

bool isGridEqual(map<string, Coords> g1, map<string, Coords> g2) {
	for (auto pair : g1) {
		if (g1[pair.first].x != g2[pair.first].x && g1[pair.first].y !=
g2[pair.first].y) {
			return false;
		}
	}
	return true;
}
double generateInitialTemp(vector<Result> init, double prob, float const
w1, float const w2, float const w3, map<string, Net> const nets, bool&
routable, int wireConstraint, map<string, Node> nodes) {
	double emax = 0., emin = 0.;
	double esum = 0;
```

```
	vector<double> es;
	for (Result r : init) {
		double e = 0.;
		while (e <= 0.) { //get an positive transition
			int ra = rand() % 3;
			int ri = rand() % init.size();
			int ri2 = rand() % init.size();
			if (ra == 0) { //select
				e = r.cost - init.at(ri).cost;
			}
			else if (ra == 1) { //crossover
				Grid copy = crossover(&init.at(ri).g,
&init.at(ri2).g, nets, nodes); //repleace with multithreaded version
				vector<Bounds> b;
				e = copy.calcCost(w1, w2, w3, nets, routable,
wireConstraint, b) - r.cost;
			}
			else {
				Grid copy = r.g;
				int rx = rand() % copy.getGridX(); //create initial
grid x param;
				int ry = rand() % copy.getGridY();//create initial
grid y param;
				for (int i = 0; i < 150; i++) {
					copy.mutation(rx, ry);
				}
				//copy.smartMutation(rx, ry, r.bounds);
				bool route = false;
				vector<Bounds> b;
				float cost = copy.calcCost(w1, w2, w3, nets,
routable, wireConstraint, b);
				Result n(copy, cost, route, b);
				e = n.cost - r.cost;
			}
		}
		es.push_back(e);
		esum += e;
	}
	auto emaxit = max_element(es.begin(), es.end());
	auto eminit = min_element(es.begin(), es.end());
	emax = *emaxit;
	emin = *eminit;
	double t = emax;
```

```
	double xo = 0.8;
	double x = 0.;
	while (x < xo) {
		x = exp(-(emax / t)) / exp(-(emin / t));
		t = t * pow((log(x) / log(xo)), (1. / prob));
	}
	return t;
}



double schedule(double temp, double initialTemp) {
	double percentComplete = (initialTemp - temp) / initialTemp;
	if (percentComplete < 0.1 || percentComplete > 0.92) {
		return 0.95 * temp;
	}
	else return 0.9 * temp; //changed to cool slower
}

Result simulatedAnnealing(vector<Result> initialGrids, float const w1,
float const w2, float const w3, map<string, Net> nets, int wireConstraint,
map<string, Node> nodes) {
	bool routable = false; // Ensure it's declared
	//double t = generateInitialTemp(initialGrids, 5., w1, w2, w3, nets,
routable, wireConstraint, nodes);
	double t = 92.52;
	double initT = t;
	vector<Result> population = initialGrids;
	vector<Result> new_pop;
	vector<Result> best_pop = population;
	double deltaC = 0;
	cout << "Initial Cost: " << bestCost(population).cost << endl;

	std::ofstream outFile("best_costs.txt");
	if (!outFile) {
		std::cerr << "Failed to open file for writing.\n";
		exit(1); // Handle error as needed
	}

	int iteration = 1;
	while (t > (0.01 * initT) && routable == false) {
		if (iteration == 3) {
			cout << "flag" << endl;
```

```
            }
            cout << "Iteration " << iteration << "; Temp = " << t << endl;
            float s = 0.3, c = 0.3, m = 0.25;
            new_pop = perturb(population, nets, w1, w2, w3, wireConstraint,
nodes, s, c, m); //NEED PERTURB FUNCTION //NEEDS TO RETURN LIST OF GRIDS :
COST : ROUTABLE?
            Result nbc = bestCost(new_pop);
            auto it = std::find_if(new_pop.begin(), new_pop.end(),
                  [&nbc](const Result& mem) { return nbc.cost == mem.cost;
});
            int index = 0;
            string st = "";
            if (it != new_pop.end()) {
                  int index = distance(new_pop.begin(), it);
            }
            if (index < s * population.size()) {
                  st = "Selection";
            }
            else if (index < (s + c) * population.size()) {
                  st = "Crossover";
            }
            else {
                  st = "Mutation.";
            }
            if (nbc.routable == true) {
                  outFile.close();
                  return nbc;
            }
            deltaC = nbc.cost - bestCost(population).cost;
            cout << "\t Best Delta C = " << deltaC << endl;

            random_device rd;
            mt19937 gen(rd());
            uniform_real_distribution<double> dis(0.0, 1.0);
            double r = dis(gen);
            double e = exp(deltaC / t);

            sort(population.begin(), population.end(), compareByFloat);
            sort(new_pop.begin(), new_pop.end(), compareByFloat);

            double sum1 = 0, sum2 = 0;
            for (int i = 0; i < 3; i++) {
                  sum1 += population.at(i).cost;
```

```
                sum2 += new_pop.at(i).cost;
            }
            bool explore = false;
            if (sum2 < sum1) {
                population = new_pop;
                if (nbc.cost < bestCost(best_pop).cost) {
                    best_pop = new_pop;
                    cout << "\t \t new best population!" << endl;
                }
            }

            else if (r > e) {
                population = new_pop;
                explore = true;
            }

            // Updated logging line including temperature and best delta C
            outFile << "Iteration: " << iteration << ", Temperature: " << t
<< ", Best Cost: " << bestCost(best_pop).cost << ", Best Delta C: " <<
deltaC << "Best Result from: " << st << ", Exploration?: " << explore <<
endl;

            t = schedule(t, initT);
            iteration++;
        }

        outFile.close();
        return bestCost(best_pop);
}


int main() {
        auto start = std::chrono::high_resolution_clock::now();

        std::string netfile = "P2Benchmarks\\ibm01\\ibm01.nets";
        std::string nodefile = "P2Benchmarks\\ibm01\\ibm01.nodes";
        std::string plfile = "P2Benchmarks\\ibm01\\ibm01.pl";

        std::map<std::string, Node> nodes;
        std::map<std::string, Net> nets;
        int numNets = 0, numPins = 0, numNodes = 0, numTerminals = 0;

        read(netfile, nodefile, plfile, nodes, nets, numNets, numPins,
```

```
numNodes, numTerminals);

      // Open summary file early to write benchmark summary before
proceeding
      std::ofstream summaryFile("optimization_summary.txt",
std::ios_base::app);
      if (summaryFile) {
            // Benchmark information written at the start
            summaryFile << "Benchmark Summary:\n";
            summaryFile << "Total Nodes: " << numNodes << "\n";
            summaryFile << "Total Nets: " << numNets << "\n";
            summaryFile << "Total Pins: " << numPins << "\n";
            summaryFile << "Total Terminals: " << numTerminals << "\n\n";
      }
      else {
            std::cerr << "Failed to open summary log file for
appending.\n";
            return 1;
      }

      findTop10Percent(nets);
      std::vector<Result> init = createInitialGrids(nodes, 10, 1.0, 0.5,
1.0, nets, 4);
      double initialCost = init.empty() ? 0 : bestCost(init).cost;
      exportForVisualization(init[0], "initial.txt");

      if (init.empty()) {
            std::cerr << "Failed to create initial grids. Aborting
optimization.\n";
            summaryFile.close();
            return 1;
      }

      Result bestResult = simulatedAnnealing(init, 1.0, 0.5, 1.0, nets, 4,
nodes);
      exportForVisualization(bestResult, "results.txt");
      if (!bestResult.routable) {
            std::cerr << "Failed to find a routable solution.\n";
            summaryFile << "Failed to find a routable solution.\n";
      }
      else {
            std::cout << "Optimization successful. Best cost: " <<
bestResult.cost << ".\n";
```

```
        }

        auto stop = std::chrono::high_resolution_clock::now();
        auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);
        double improvement = initialCost - bestResult.cost;
        double improvementPercentage = initialCost > 0 ? (improvement /
initialCost) * 100 : 0;

        summaryFile << "Optimization Results:\n";
        summaryFile << "Initial Cost: " << initialCost << "\n";  // Log the
initial cost
        summaryFile << "Final Cost: " << bestResult.cost << "\n";
        summaryFile << "Improvement: " << improvement << " (" <<
improvementPercentage << "%)\n";
        summaryFile << "Routable Solution Found: " << (bestResult.routable ?
"Yes" : "No") << "\n";
        summaryFile << "Total Execution Time: " << duration.count() / 1000.0
<< " seconds\n\n";
        summaryFile.close();

        return bestResult.routable ? 0 : 1;
}
```