Report

Digital Design II Project

Amr Hussein | 900201827
John William | 900193002
Omar Elwaliely | 900202853
Seifeldin Elshabshiri| 900202310

Fall 2023
Dr. Mohamed Shalan

10th December, 2023

Table of Contents

1. Project Description

This project is a simple simulated annealing-based placer that minimizes the total wirelength. It studies the effect of cooling rate on the quality of the placement. In this project, we assume the following:

- HPWL (half-perimeter of the smallest bounding box containing all pins for a net) is used to estimate the wirelength of any net
- The core area is a 2D array of empty squares (sites)
- Each cell is a square and matches the site size.
- The site size is 1x1.
- No site is assigned to more than one cell.
- The distance between two cells is measured from the center of one cell to the center of the other

The input to the placer is a netlist file in text format with the following format

- ☐ The first line contains 4 values:
    - The number of cells to be placed.
    - The number of connections between the cells.
    - The number of rows (ny) upon which the circuit should be placed.
    - The number of columns (nx) upon which the circuit should be placed.
- ☐ Each of the following lines represents a net, and it contains the following:
    - The number of components attached to the net
    - The list of components attached to the net

The program parses the input and prints a grid showing the initial placement as follows: it displays the cell ID for filled entries and "--" for empty cells. In addition to these views, the initial wire length is displayed. Then, the program starts searching for an optimal placement by applying simulated annealing to find the sought optimal placement. After the search, the placement is shown again, and the total wire length after placement is also shown.

2. Algorithm and Implementation

This section discusses the code, data structures, and algorithms used for each program stage. The program starts by reading the first line of the test file. It reads the number of cells, number of nets, number of rows, and number of columns. Then, it initializes the data structures used. Then, the program reads the rest of the file and randomly places the cells. After this, the temperature is initialized, and the main loop starts where two cells are picked randomly and then swapped. The swap is accepted or rejected according to the change in cost and the current temperature. Now, we discuss in detail the aspects of the program.

2.1. Data Structures

The primary data structures used in this code are:

- [ ] Structs:
    - `Cell`: Represents a cell in the circuit with attributes such as value, position (posX, posY), and a vector of nets connected to it.
    - `Placer`: Contains information about the placement problem, including the number of components, number of nets, dimensions of the grid, initial and final temperatures for simulated annealing, vectors for storing nets and cell positions, and vectors for tracking the bounding box of each net.
- [ ] Vectors:
    - `vector<vector<int>> nets`: Represents the netlist, where each row corresponds to a net and contains the values of the cells connected in that net.
    - `vector<Cell> cellPositions`: Represents the positions of cells on the grid, where each cell is an instance of the `Cell` struct.
    - `vector<int> maxX, maxY, minX, minY`: Store the bounding box information for each net in terms of maximum and minimum X and Y coordinates.

- ☐ Random Number Generation::
    - `random_device rd; mt19937 gen(rd());`: Used for generating random numbers.
    - `uniform_int_distribution<int> disy, disx, disC`: Used for generating random row and column indices and random cell indices.
    - `uniform_real_distribution<double> dis2`: Used for generating a random real number between 0.0 and 1.0 for acceptance probability in simulated annealing.
- ☐ Constants:
    - Constant Integers:
        - `const int EMPTY_CELL, INIT_TEMP, MOVES`: Represent constants for empty cells, initial temperature in simulated annealing, and the number of moves per temperature, respectively.
    - Constant Double:
        - `const double FINAL_TEMP, COOLING_RATE`: Represent constants for the final temperature and cooling rate in simulated annealing, respectively.

2.2. Initialization Process

In the provided code, the initialization process involves two key functions: `makeCore` and `placeRandomly`. Let's discuss each of these functions:

2.2.1. **`makeCore`** Function:

```
void makeCore(Placer& p)

{

        p.cellPositions.resize(p.nx * p.ny);

        for (int i = 0; i < p.nx * p.ny; i++)

        {

                p.cellPositions[i].posX = 0;

                p.cellPositions[i].posY = 0;

        }

}
```

This function initializes the chip's core by resizing the `cellPositions` vector to accommodate the total number of cells in the chip (`p.nx * p.ny`). It then sets the initial position of each cell to (0, 0).

Complexity Analysis:

- Time Complexity: O(p.nx * p.ny)
- Space Complexity: O(p.nx * p.ny)

2.2.2 placeRandomly Function:

```
void placeRandomly(Placer& p) {

    vector<tuple<int, int>> inserted;

    for (int i = 0; i < p.cellPositions.size(); i++) {

     int row = rand() % p.nx;

     int col = rand() % p.ny;

     bool placed = false;

     while (!placed) {

     if(find(inserted.begin(),inserted.end(),make_tuple(row,col))==inserted.end())

     {

             p.cellPositions[i].posX = row;

             p.cellPositions[i].posY = col;

             p.cellPositions[i].value = i;

             inserted.push_back({ row, col });

             placed = true;

      }

      row = rand() % p.nx;

      col = rand() % p.ny;

      }

    }

 for (int i = 0; i < p.nets.size(); i++) {

      for (int j = 0; j < p.nets[i].size(); j++) {

             p.cellPositions[p.nets[i][j]].nets.push_back(i);

      }

   }

}
```

This function randomly places cells on the chip, ensuring each cell is placed uniquely. It also associates cells with the nets they belong to.

Complexity Analysis:

- Time Complexity: O(p.numOfComponents * (p.nx * p.ny))
    - The worst-case scenario occurs if there's a need to retry placing each component until a unique position is found.
- Space Complexity: O(p.numOfComponents * (p.nx * p.ny))

2.3. Simulated Annealing function

The `simulatedAnealing` function implements the Simulated Annealing optimization algorithm, which is a probabilistic optimization technique used to find a good approximation to the global optimum of a given function. In the context of the provided code, the objective is to optimize the placement of components on a chip by minimizing the Half Perimeter Wirelength (HPWL) metric.

Let's break down the key components of the function:

Temperature Initialization:

- The initial temperature (`temp`) is set based on the placement's initial cost (HPWL).
- The final temperature (`p.finalTemp`) is calculated as a fraction of the initial cost.

Random Number Generators:

- Random number generators (`random_device` and `mt19937`) and various uniform distributions are used to generate random integers and real numbers needed for random cell selection and probability calculations.

Main Loop (Simulated Annealing Process):

- The function enters a loop that continues until the temperature falls below the final temperature.
- Within this loop, a set number of moves per temperature (`p.movesPerTemp`) are performed to explore the solution space.

Nested Loop (Moves Per Temperature):

- A nested loop is executed for each iteration to perform a specified number of moves.
- In each move, two random cells (`cell1` and `cell2`) are selected for potential swapping.

Swapping and Cost Calculation:

- The positions of the selected cells are swapped in the placement (`p.cellPositions`).
- The HPWL is updated based on only the nets of the swapped cells and not all the nets(`updateHpwl` function).
- The change in cost (`deltaCost`) is calculated.

Acceptance or Rejection of Move:

- The move is accepted or rejected based on a probabilistic condition.
- If the move increases the cost (deltaCost > 0), it may still be accepted with a certain probability determined by the simulated annealing schedule and a random number.

Temperature Update:

- The temperature is updated according to the simulated annealing schedule (temperature *= 0.95).

Final Result:

- The final placement with minimized HPWL is achieved when the temperature falls below the final temperature.

Complexity Analysis:

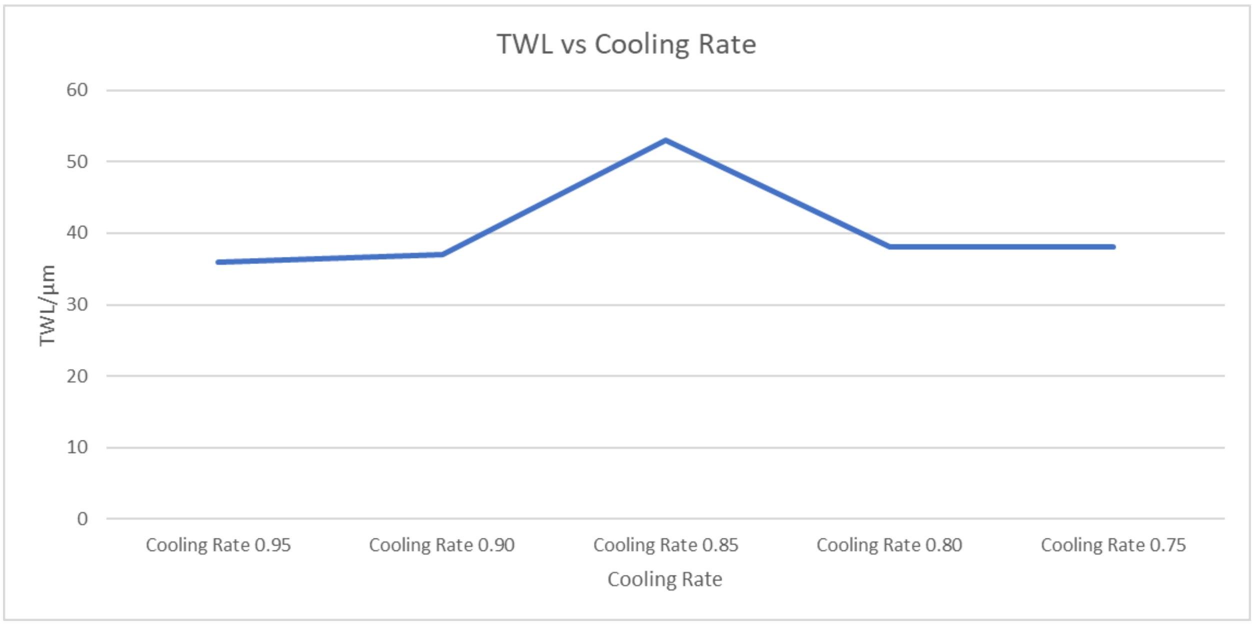
- **Time Complexity:**
  - The time complexity of the function is influenced by the number of iterations in the main loop (controlled by temperature reduction) and the number of moves per iteration (`p.movesPerTemp`).
  - The overall time complexity is O(Iterations * MovesPerIteration).
- **Space Complexity:**
  - The space complexity is influenced by the storage of auxiliary variables, vectors (`MaxXtemp`, `MinXtemp`, etc.), and the `oldVec` vector.
  - The overall space complexity is O(N), where N is the number of cells in the placement.

3. Results

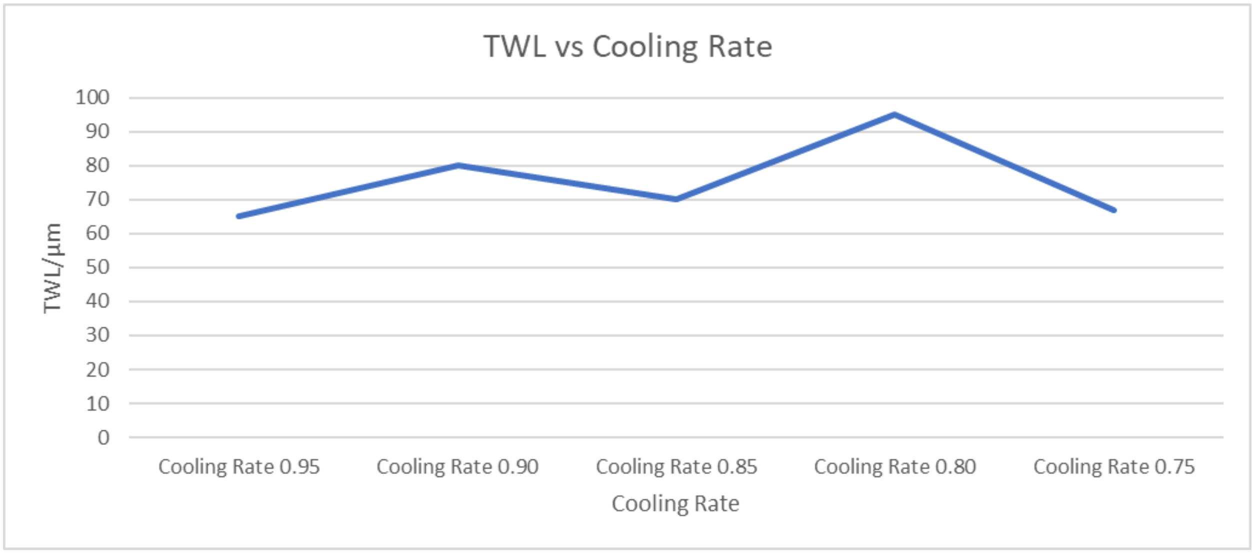
3.1. TWL vs Cooling Rate Graph

We employed a consistent random seed for every design and executed it across five distinct cooling rates: 0.95, 0.9, 0.85, 0.8, 0.75. The total wire length for each design was computed at each cooling rate, yielding the following outcomes:

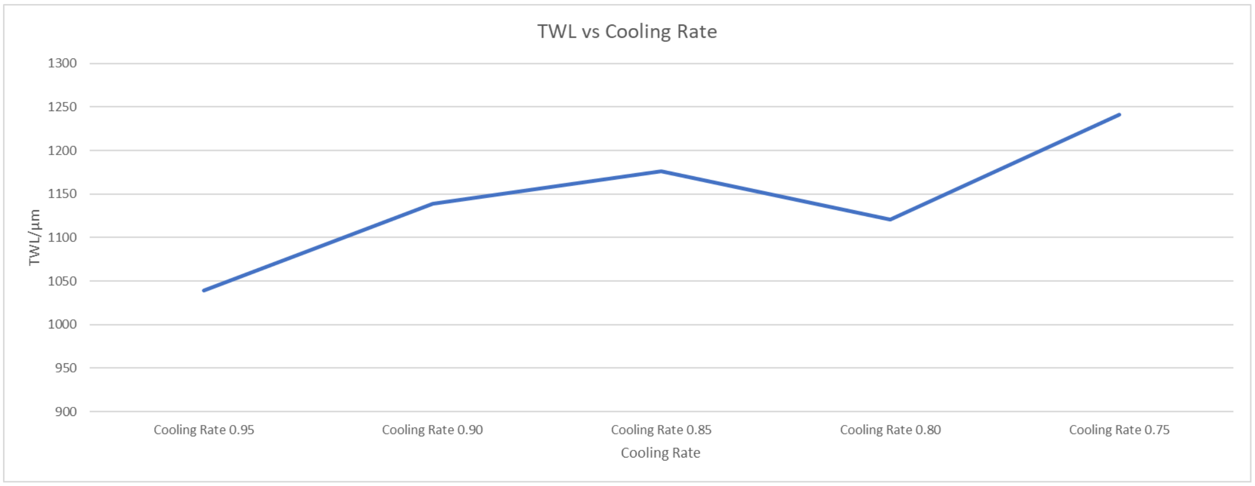
3.1.1. d0



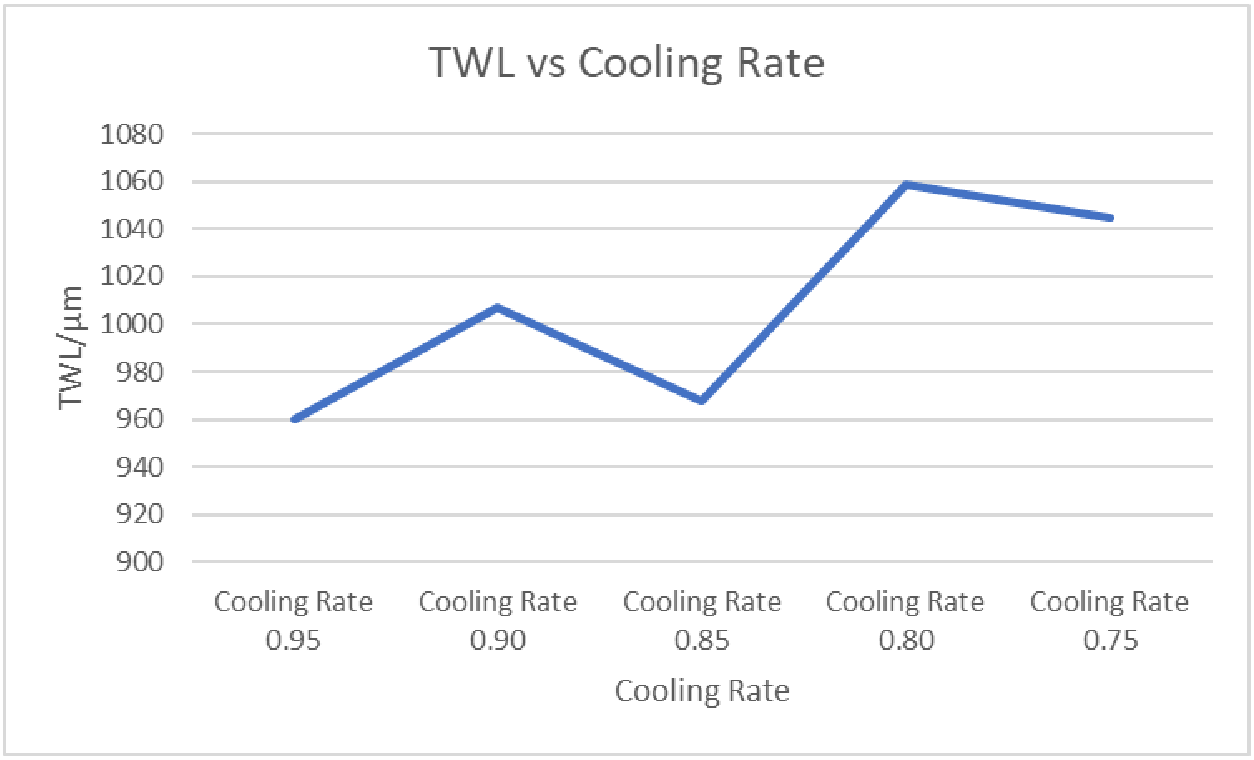
3.1.2. d1


TWL vs Cooling Rate
TWL/μm
100
90
80
70
60
50
40
30
20
10
0
Cooling Rate 0.95
Cooling Rate 0.90
Cooling Rate 0.85
Cooling Rate 0.80
Cooling Rate 0.75
Cooling Rate

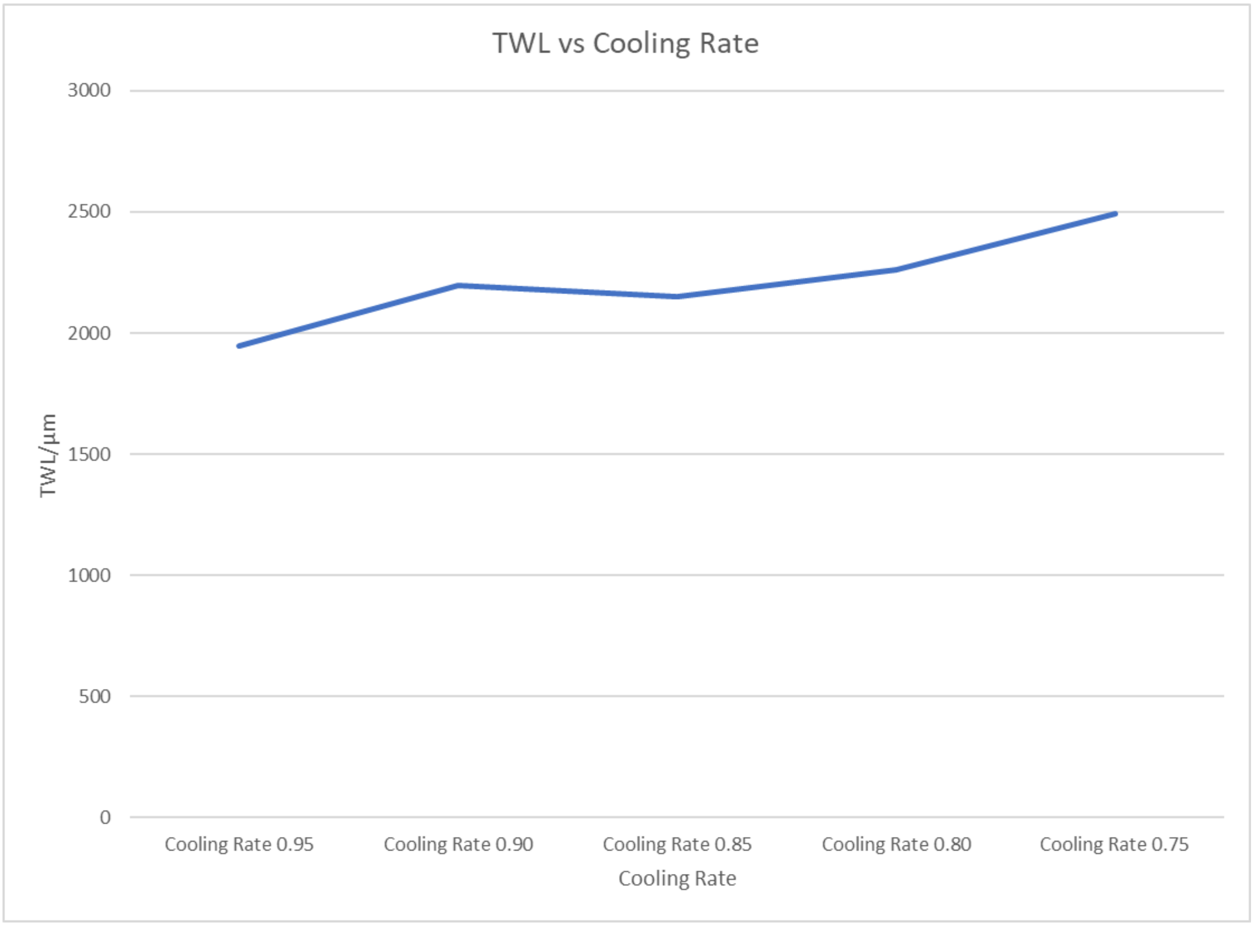
3.1.3. d2


TWL vs Cooling Rate
TWL/μm
1300
1250
1200
1150
1100
1050
1000
950
900
Cooling Rate 0.95
Cooling Rate 0.90
Cooling Rate 0.85
Cooling Rate 0.80
Cooling Rate 0.75
Cooling Rate

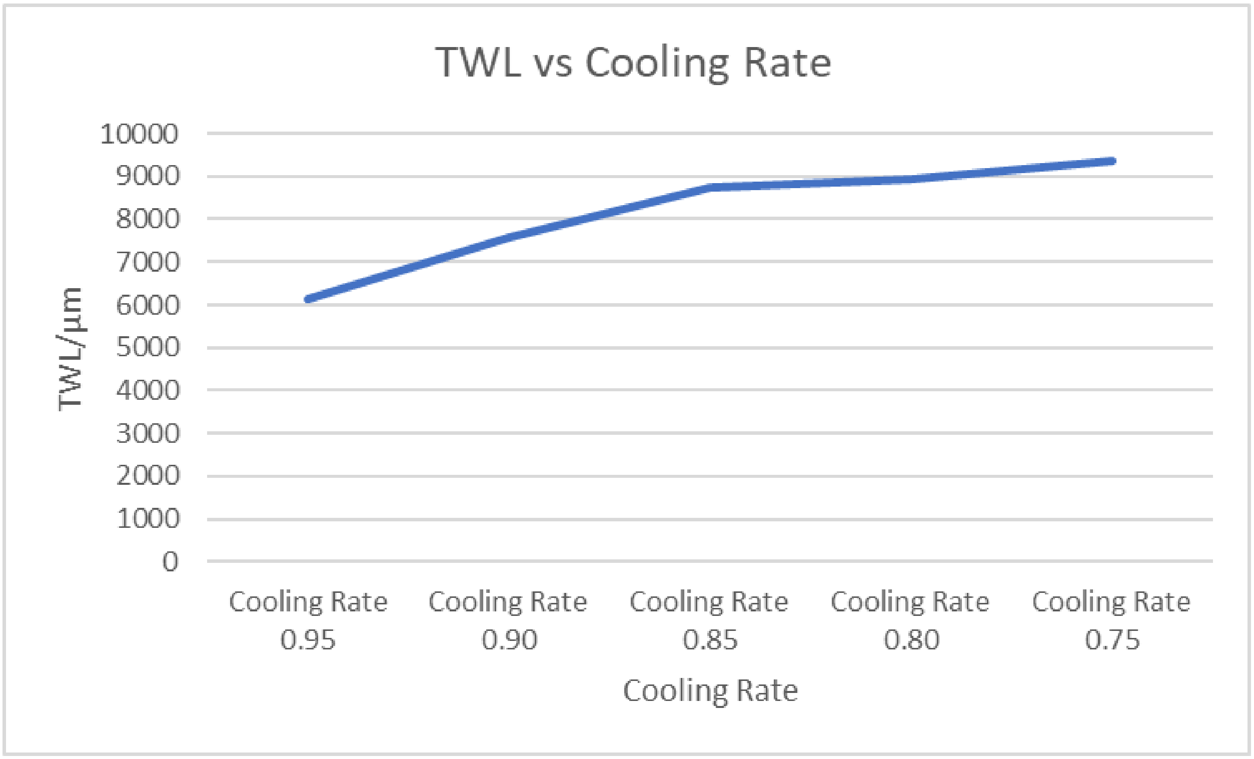
3.1.4. d3


TWL vs Cooling Rate
TWL/µm
1080
1060
1040
1020
1000
980
960
940
920
900
Cooling Rate 0.95
Cooling Rate 0.90
Cooling Rate 0.85
Cooling Rate 0.80
Cooling Rate 0.75
Cooling Rate

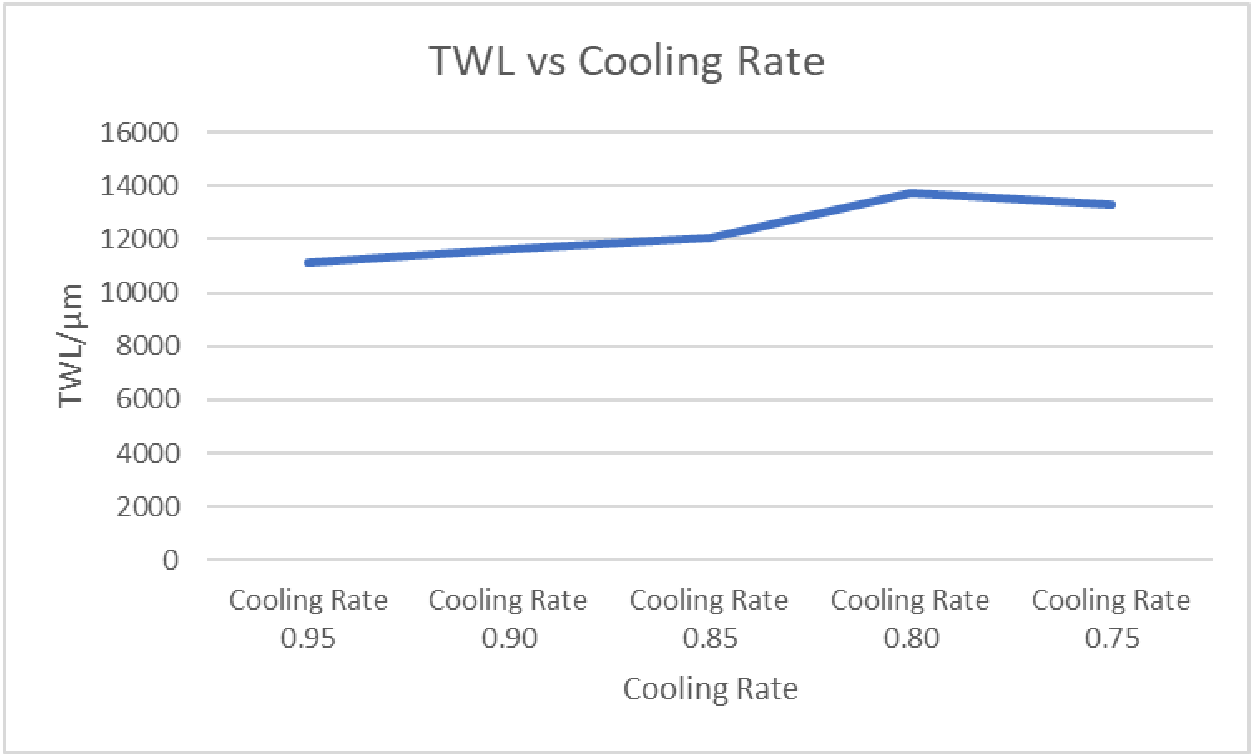
3.1.5. t1


TWL vs Cooling Rate
3000
2500
2000
1500
1000
500
0
TWL/µm
Cooling Rate 0.95
Cooling Rate 0.90
Cooling Rate 0.85
Cooling Rate 0.80
Cooling Rate 0.75
Cooling Rate

3.1.6. t2


TWL vs Cooling Rate
TWL/µm
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Cooling Rate 0.95
Cooling Rate 0.90
Cooling Rate 0.85
Cooling Rate 0.80
Cooling Rate 0.75
Cooling Rate

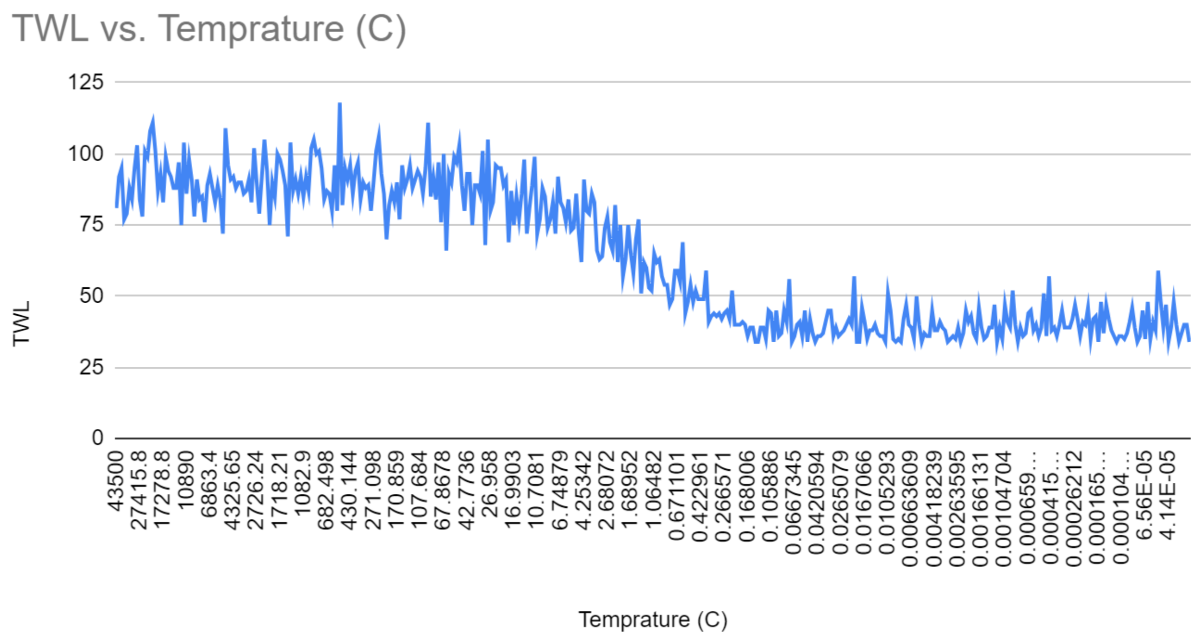
3.1.7. t3


TWL vs Cooling Rate
16000
14000
12000
10000
8000
6000
4000
2000
0
TWL/μm
Cooling Rate 0.95
Cooling Rate 0.90
Cooling Rate 0.85
Cooling Rate 0.80
Cooling Rate 0.75
Cooling Rate

3.2. TWL vs Temperature Graph

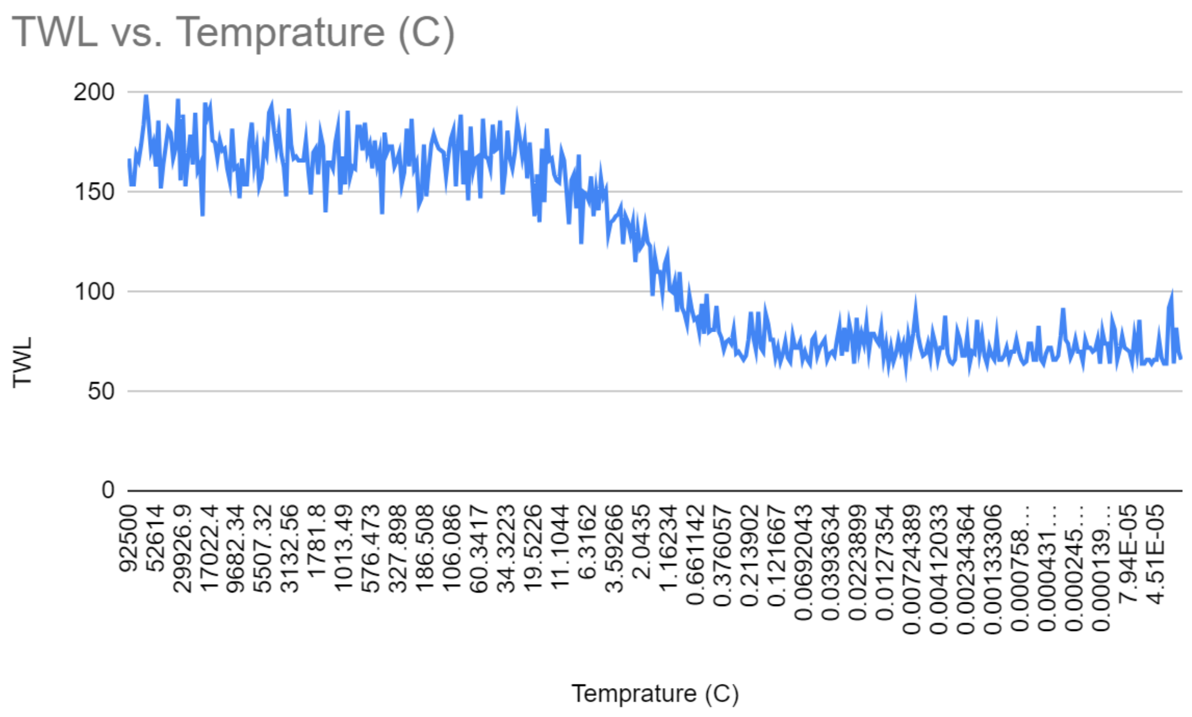
Calculating the TWL at each cooling rate for each design helped us determine the optimal cooling rate for each design. To plot the temperature against the total wire length, we ran each design at its optimal cooling rate and extracted the TWL at each temperature. For all the data sets collected, the temperature decreases exponentially. Therefore, we used a log scale in our graphs. All graphs start off with oscillations where the algorithm makes wrong decisions. As the temperature decreases, the graphs stabilize and reach a local minimum. The graphs obtained from the data collected were as follows:

3.2.1. d0



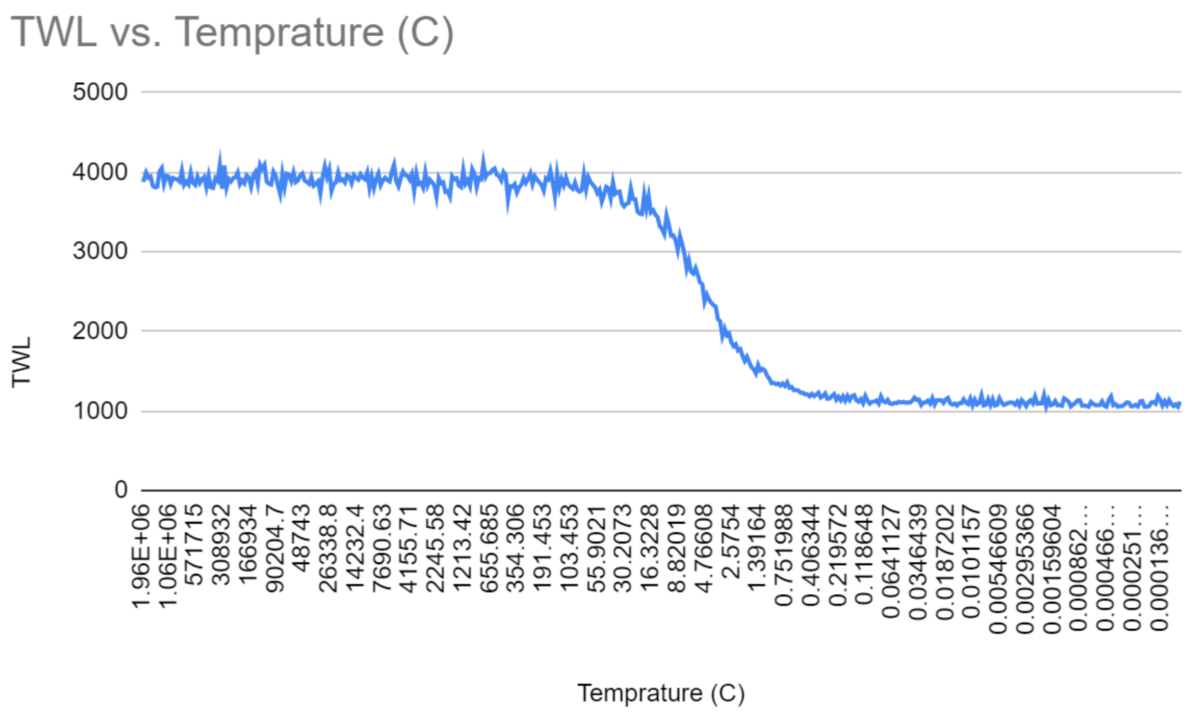
The graph reaches a stable point with a total wire length (TWL) of approximately 40.

3.2.2. d1



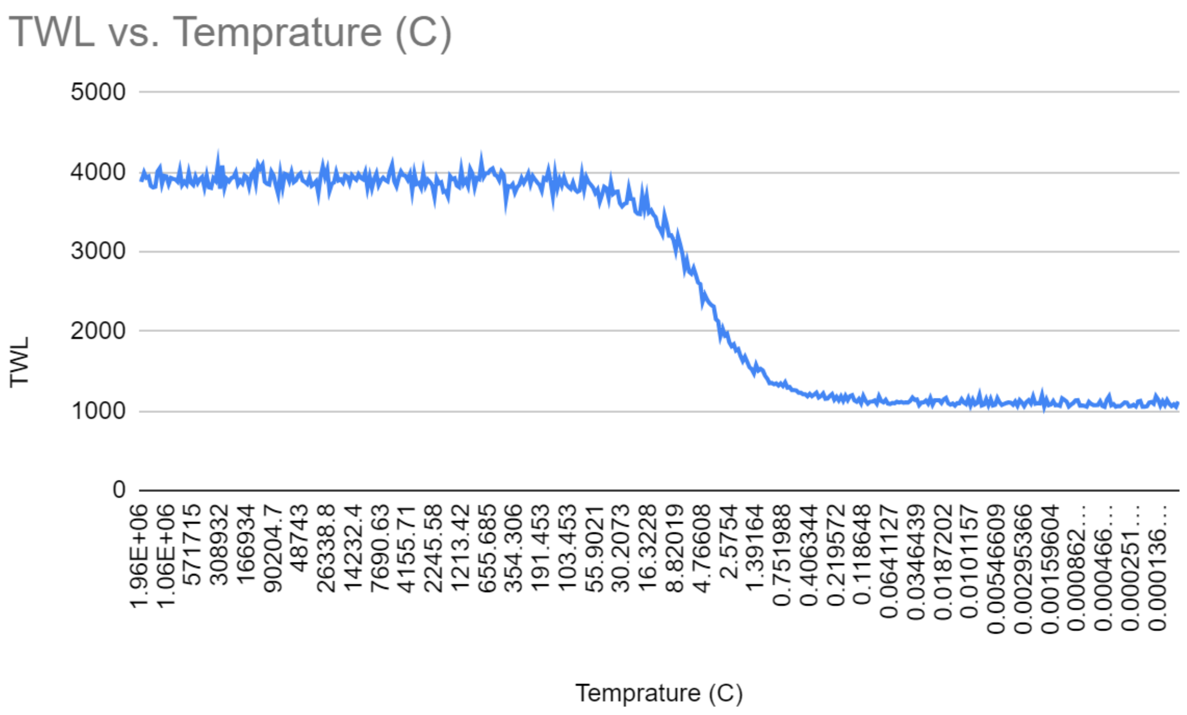
The graph reaches a stable point with a total wire length (TWL) of approximately 65.

3.2.3. d2



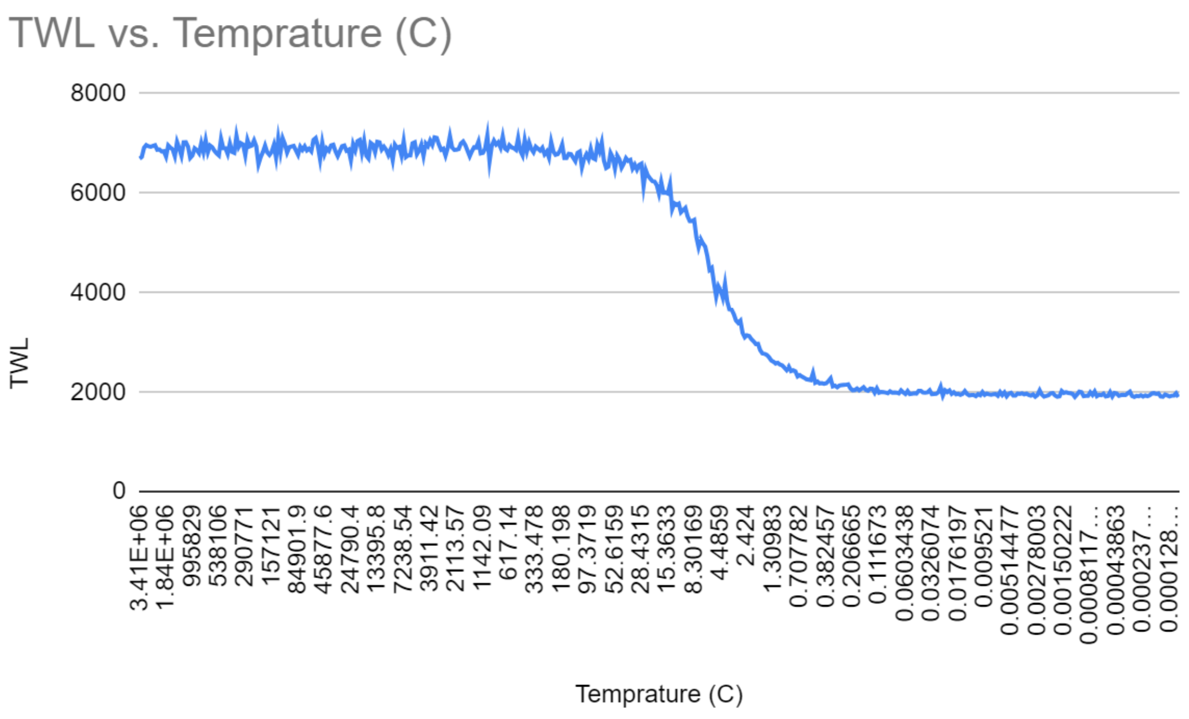
The graph reaches a stable point with a total wire length (TWL) of approximately 1100.

3.2.4. d3



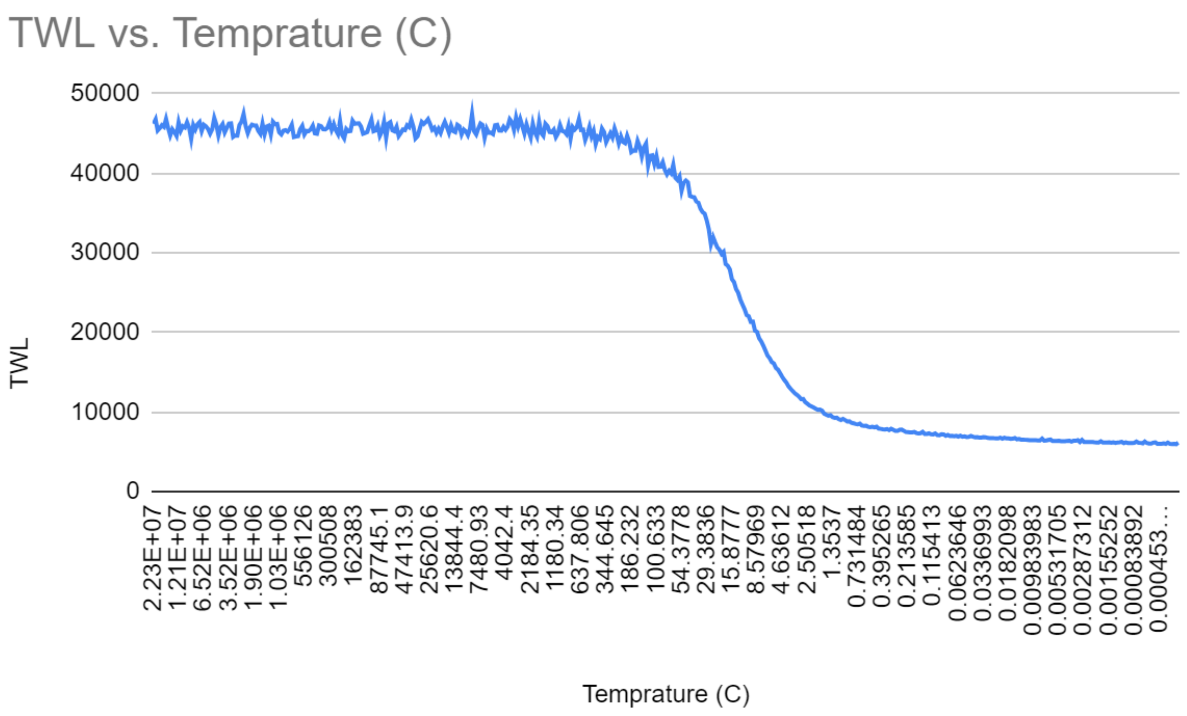
The graph reaches a stable point with a total wire length (TWL) of approximately 950.

3.2.5. t1



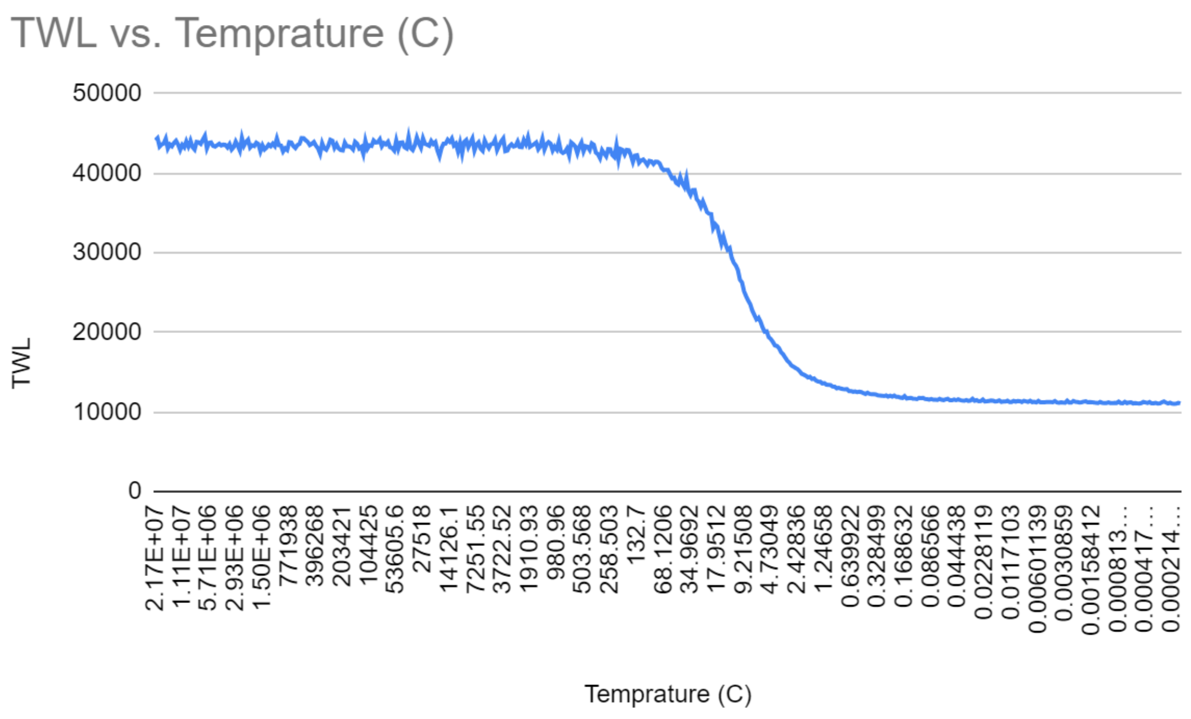
The graph reaches a stable point with a total wire length (TWL) of approximately 1900.

3.2.6. t2



The graph reaches a stable point with a total wire length (TWL) of approximately 6000.

3.2.7. t3



The graph reaches a stable point with a total wire length (TWL) of approximately 11000.

GitHub Main Repo Link

https://github.com/Seif2001/simulatedAnealling

GitHub GUI Project Repo Link

https://github.com/Seif2001/SmulatedAneallingGUI
Note: link SFML library to run