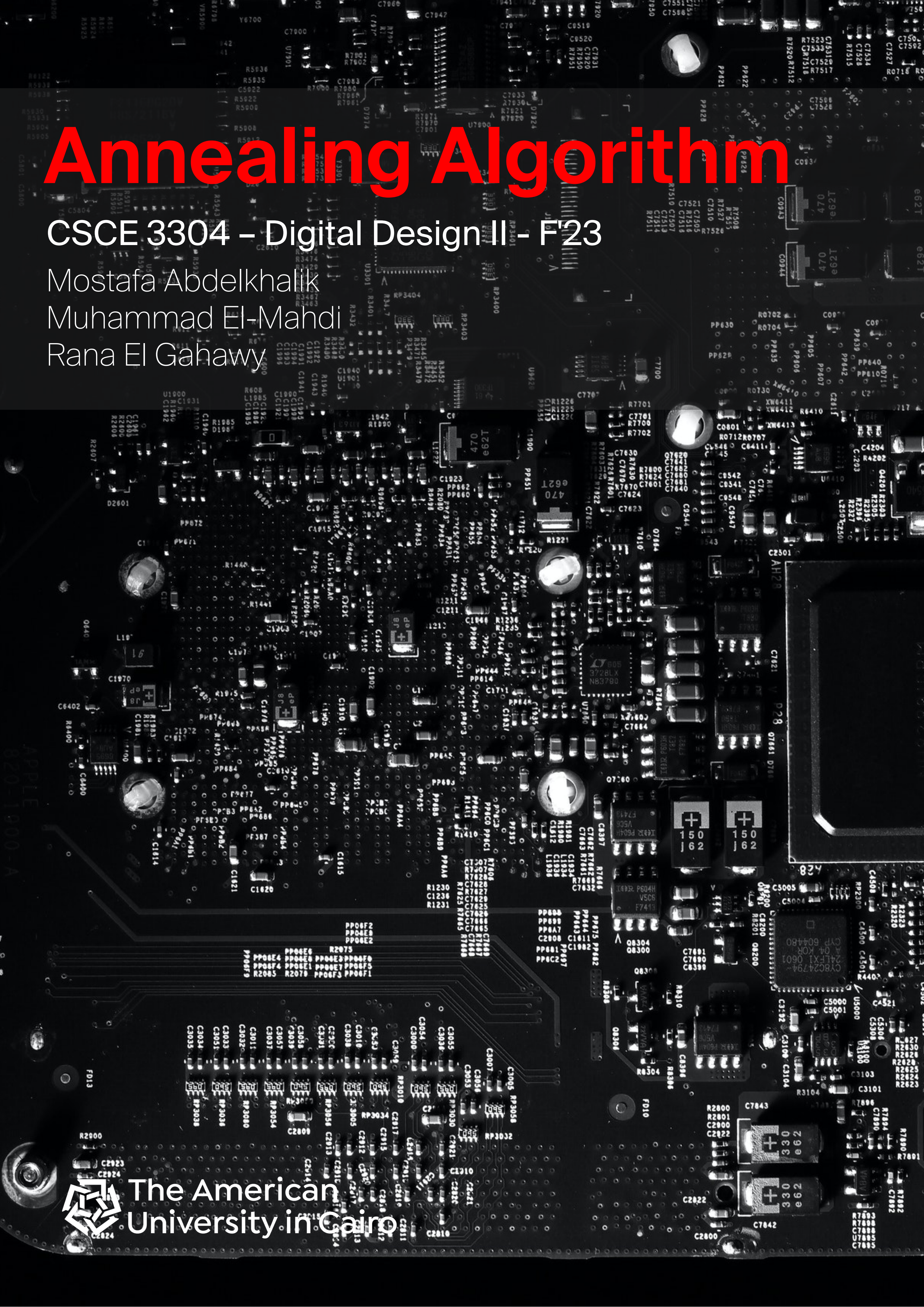


Annealing Algorithm

CSCE 3304 – Digital Design II - F'23

Mostafa Abdelkhalik
Muhammad El-Mahdi
Rana El Gahawy

The American University in Cairo

Annealing Algorithm

by

Mostafa Abdelkhalik
Muhammad El-Mahdi
Rana El Gahawy

Student Name	Student ID
Abdelkhalik	900203356
El-Mahdi	900202967
ElGahawy	900202822

Faculty: School of Science and Engineering, AUC
Instructor: Dr. Mohamed Shalan | Fall 2023

Contents

1 Introduction

Simulated annealing is a probabilistic optimization technique inspired by the annealing process in metallurgy. The project aims to apply simulated annealing to minimize wire length in circuit layouts, an essential problem in electronic design automation. Wire length directly influences the performance and efficiency of electronic circuits.

1.1. Objectives

The primary objective of this project is to experiment the annealing methodology for placement with an optimized code that consumes the lowest possible time. We want also to analyze how variations in the cooling factor during the simulated annealing process affect the resulting wire length. The cooling factor is a crucial parameter in the annealing algorithm, influencing the probability of accepting worse solutions at higher temperatures.

2
Methodology

2.1. Problem Formulation

As mentioned before the objective of this project is to optimize wire length in electronic circuit layouts using simulated annealing. Hence, a thorough process was introduced to implement, test, and visualize the algorithm and the data generated from it.

2.2. Algorithm Implementation

The core of the project involved implementing the simulated annealing algorithm in C++. The algorithmic implementation considered the key components of simulated annealing, including the initial temperature, cooling factor, and acceptance probability function. The temperature schedule and annealing parameters were varied to explore their effect on the wire length.

2.3. Visualization Using Matplotlib

For a comprehensive analysis of the annealing process, the Matplotlib library in Python was employed to create visualizations. Graphs depicting the change in wire length over iterations were generated. These visualizations proved instrumental in understanding the algorithm's convergence behavior. Then another set of graphs were generated by varying the cooling factor to identify its effect on the final wire length.

2.4. GIF Animation Generation

To enhance the presentation and communication of the annealing process, a GIF animation was generated. The animation showcased the evolution of the circuit layout and wire length over successive iterations. This dynamic representation allowed for a more intuitive understanding of how the algorithm explored and optimized the solution space.

2.5. Experimental Setup

Experiments were conducted using various instances of circuit layouts with different complexities. The impact of changing the cooling factor on the wire length was systematically analyzed. The algorithm's performance was evaluated based on the convergence speed and the quality of the final solution.

2.6. Data Collection

Data on wire length at each iteration, as well as the final wire length for different cooling factors, were collected and analyzed. The collected data formed the basis for generating graphs and drawing conclusions about the relationship between the cooling factor and wire length.

2.7. Code Optimization

Efforts were made to optimize the C++ code to ensure efficient execution, especially for larger and more complex circuit layouts. The goal was to achieve a balance between algorithmic effectiveness and computational efficiency.

The combination of algorithmic implementation, visualization using Matplotlib, and the creation of a GIF animation provided a comprehensive methodology for studying and presenting the simulated annealing optimization process for wire length in electronic circuit layouts.

3

Implementation

3.1. User Defined Types

3.1.1. Structures

The building unit of the placement algorithm is the cells of the component. Since each cell has different features that are essential to it whenever it is used, it made sense to use structs to group these features together in one unit.

```
struct Cell {
    int number;
    int x;
    int y;
    vector < int > nets;
    };
```

- **number:** Stores the number of this component.
- **x:** Stores the x coordinate of this cell's position.
- **y:** Stores the y coordinate of this cell's position.
- **nets:** Stores all the nets that can be affected by the change in this cell's position.

3.1.2. Classes

In this project, object-oriented programming (OOP) was used to ensure an organized and readable code that can be easily maintained and can smoothly adapt to future changes and enhancements. One class was used for the placement for the whole project. This class supports parsing an input netlist, calculating the half perimeter wire length (HPWL), and annealing algorithm. Different placement algorithms can be added to this class later besides annealing.

```
class placer {
    public:
    placer(string filename);
    void run();

    private:
    int
        numRows,
        numColumns;
    int
        totalcomponents,
        totalnets;
    int totalHPWL;
    vector < Cell * > components;
    vector < vector < Cell * >> netlist;
    vector < vector < Cell * >> netlist_y;
    vector < vector < int >> grid;
    vector < int >
                    HPWL_X,
                    HPWL_Y;
    vector < int >
                    HPWL_Y_I,
                    HPWL_X_I,
                    HPWL_Y_I2,
```

```
                    HPWL_X_I2;

    void parseInput(const string filename, int & totalcomponents);
    void initialPlacement(int numTotalComponents);
    static bool compare_X_coordinate(Cell * A, Cell * B);
    static bool compare_Y_coordinate(Cell * A, Cell * B);
    int calculateHPWL_X(vector < Cell * > & X);
    int calculateHPWL_Y(vector < Cell * > & Y);
    void calculateInitialHPWL();
    void checkHPWL_X(const Cell * cell, int net, int c);
    void checkHPWL_Y(const Cell * cell, int net, int c);
    bool checker(const int IHPWL1, const Cell * X, const Cell * Y, double
        temperature);
    void annealing();
    void printPlacement();
    void printBinaryIPlacement();
};
```

The class has two public functions:

- **placer():** The constructor for the placer object that takes the netlist file name as input is responsible for doing the initial random placement and calculating the initial HPWL.
- **run():** The run function is responsible for calling the functions needed for annealing in addition to calculating the annealing time.

All the variables are declared in the private part in addition to the rest of the functions.

Variables like **numRows**, **numColumns**, **totalcomponents**, and **totalnets**, are initialized from the netlist file and are used throughout the code. On the other hand, **totalHPWL** is initialized in the instructor with 0 and its value keeps getting updated with each swap in the annealing function.
There are other variables that are also private to the class like: components, netlist, netlist_y, grid, HPWL_X, HPWL_Y, HPWL_Y_I, HPWL_X_I, HPWL_Y_I2, and HPWL_X_I2; however, their functionality will be explained in the following Data Structures section.

There are a total of thirteen functions in this class that will be discussed in detail in the algorithm section.

3.2. Data Structures

The main data structure that was used throughout the project is vectors in two different forms: 1D and 2D. Vectors were used for several reasons: ease of use, support for random access, efficient sorting, sequential storage, and flexibility.

The vectors used in the project and their functionality:

- **components:** This is the main vector in the project; it is a vector of pointers to objects of type Cell. It contains all cells sequentially in addition to their info. During swapping, the x and y values for the swapped components are updated in the components vector; thus, their values are automatically updated in all other places that point to the cells in this vector.
- **netlist:** The netlist data structure is represented as a vector of vectors, where each element is a pointer to a Cell object from the components vector. This two-dimensional vector, vector<vector<Cell*», is used to organize and store a matrix-like structure of Cell objects. Each row in the netlist corresponds to a specific net, and each column within a row represents a Cell associated with that net. The use of pointers allows for efficient access and modification of individual Cell objects within the netlist. In this version of the netlist, each net is sorted ascendingly according to the values of x of the cells in the net. This is to make the process of calculating the HPWL of each net easier and faster.
- **netlist_y:** The only difference between this netlist_y and the one discussed above is that the nets in this version are sorted ascendingly according to the values of y of the cells in the net. Other than that, everything from netlist applies on netlist_y;
- **grid:** This is a vector of vector of int that represents the placement grid; it is mainly used for printing, which is why it only has the number of the components without any extra info about it;
- **HPWL_X and HPWL_Y:** These are two vectors of type int that are used to store the values of the horizontal and vertical part of the HPWL, respectively;

- **HPWL_X_I and HPWL_Y_I:** These are two vectors of type int that are used to store a copy of the values of the Initial horizontal and vertical part of the HPWL, respectively, of the nets that are affected by one of the cells before swapping, so in case the swapping is rejected, they can be restored again;

3.3. Algorithm

We tried to achieve a minimal number of nested loops and extra calculations as much as possible to decrease the total time taken by the annealing function. In this section, we will discuss the functions used in the class and the implementation that was used to produce the minimum amount of time possible.

- **parseInput():** This function parses the netlist file to initialize all private variables in the class needed for the annealing process.
- **initialPlacement():** This is the function responsible for the initial placements of the components in the grid. To reduce the complexity of calculating different random x and y for each component, this function uses a linear approach to randomly place the components by declaring a local vector that has the same size as the total number of available places in the grid; this vector is then initialized with the number of components available sequentially, and the rest are set to -1. The vector is then shuffled to achieve the random placement. Finally, the elements in this vector are placed in the 2D grid in the same random order they have in the vector.

To reduce the complexity of the calculation of the HPWL, we calculate the x part and y part of the HPWL separately, and we save them in the previously discussed vectors, and since we have the nets affected by each cell in the Cell itself, whenever a cell is swapped we only recalculate the HPWL of these nets:

- **compare_X_coordinate and compare_Y_coordinate:** These are comparator functions to help sort cells according to the values of x and y, respectively;
- **calculateHPWL_X and calculateHPWL_Y:** These functions calculate the values x and y of the HPWL, respectively, of a specific net by sorting this net according to x (netlist_y for the y part).
- **calculateInitialHPWL:** This function is only called once in the constructor, as it calculates the initial HPWL by calling calculateHPWL_X and calculateHPWL_Y; consequently, it sorts all the nets in netlist and netlist_y according to the values of x and y respectively.
- **checkHPWL_X and checkHPWL_Y:** These functions check the effect of the change of a specific cell on the nets affected by this cell. It only recalculates the HPWL if needed, which is when this cell exists at the beginning or end of the sorted netlist if it is in the middle but its x (y) value is greater than that of the last cells of the net or less than that of the first cell of the net. In these cases only, we would need to sort the netlist again and recalculate the HPWL.
- **checker():** This function checks if the swap of two cells will be accepted or not with probability $1 - e^{\frac{\triangle HPWL}{T}}$. It first copies the HPWL that will be affected by swapping the two cells and stores them in HPWL_X_I and HPWL_X_I2, so in case of rejection, they can be restored. Then, it calls checkHPWL_X and checkHPWL_Y on all the affected nets from the two cells.
- **annealing():** The annealing function is responsible for initializing the initial cost, initial temp, and final temp. It generates two places in the grid and swaps them; then it calls the checker function; if the swap is rejected, it restores all the changed values resulting from the swap; otherwise, it keeps them and moves to the new iteration.
- **printPlacement():** The function prints the 2D grid where each value either has a number of a component or —- if it is an empty place.
- **printBinaryIPlacement():** It prints the 2D grid with a binary representation of 0 to empty cells and 1 to occupied cells, for easier visualization of large test cases.

3.4. Complexity

- **parseInput():** The largest complexity in this function is the nested loop for initializing the netlist. Consequently, it has a complexity of $O(totalnets * componentsPernet)$
- **initialPlacement():** It takes $O(numRows * numColumns)$ to initialize the grid.
- **calculateHPWL_X and calculateHPWL_Y:** Sorting the net take $O(nlogn)$

- **calculateInitialHPWL:** $O(totalnets * 2(nlogn))$
- **checkHPWL_X and checkHPWL_Y:** worst case: $O(calculateHPWL\_X)$ while best case: $O(3)$ for the 3 conditions before the return.
- **checker():** $O(numofnetsinacell * O(checkHPWL\_X + checkHPWL\_Y))$
- **annealing():** $O(numofdifferentvaluesoftemp * (10 * numRows * numColumns) * O(checker))$
- **printPlacement():** $O(numRows * numColumns)$
- **printBinaryIPlacement():** $O(numRows * numColumns)$

3.5. GIF generation

To better visualize the annealing process and understand the functionality of the algorithm, we used a method for creating a GIF that simulates the whole annealing process:

```
void savePlacementImage(const std::string& filename) const
{
    const int cellSize = 30;
    const int imageSizeX = cellSize * numColumns;
    const int imageSizeY = cellSize * numRows;

    CImg<unsigned char> img(imageSizeX, imageSizeY, 1, 3, 255);

    for (int i = 0; i < numRows; i++) {
        for (int j = 0; j < numColumns; j++) {
            int cell = grid[i][j];
            unsigned char color[3] = {255, 255, 255};

            if (cell != -1) {
                color[0] = color[1] = color[2] = 0;
            }

            img.draw_rectangle(j * cellSize, i * cellSize, (j + 1) * cellSize -
                1, (i + 1) * cellSize - 1, color);
        }
    }

    img.save_bmp(filename.c_str());
}
```

The function shown above with the usage of cimg library was responsible for creating images with black squares representing the components and white squares representing the empty cells.
Then this function was called in the annealing function inside the loop of the temperature scheduling to get an image after each and every temperature scheduling step.

```
void annealing() {
      cout << "started annealing" << endl;
      double initialCost = totalHPWL;
      double initialTemp = initialCost * 500;
      double finalTemp = 5 * pow(10, -6) * (initialCost / totalnets);
      double currentTemp = initialCost * 500;
      int IHPWL = initialCost;
  //    int THPWL = initialCost;
      int c = 0;
      //srand(time(0));
      while (currentTemp > finalTemp) {
      --
      --
      --
      --
        savePlacementImage("./images/image_" + to_string(c) + ".jpg");
        c++;
      }
```

Finally we used a tool named ImageMagick to combine the generated images into one gif file that shows the whole annealing process. ImageMagick was used using this command:

```
    convert -resize 50% -delay 10 -loop 0 image_{0..[lastimagenumber]}.jpg
        output.gif
```

This command is to run from the terminal in the directory where the images are stored. graphicx

4

Results and discussion

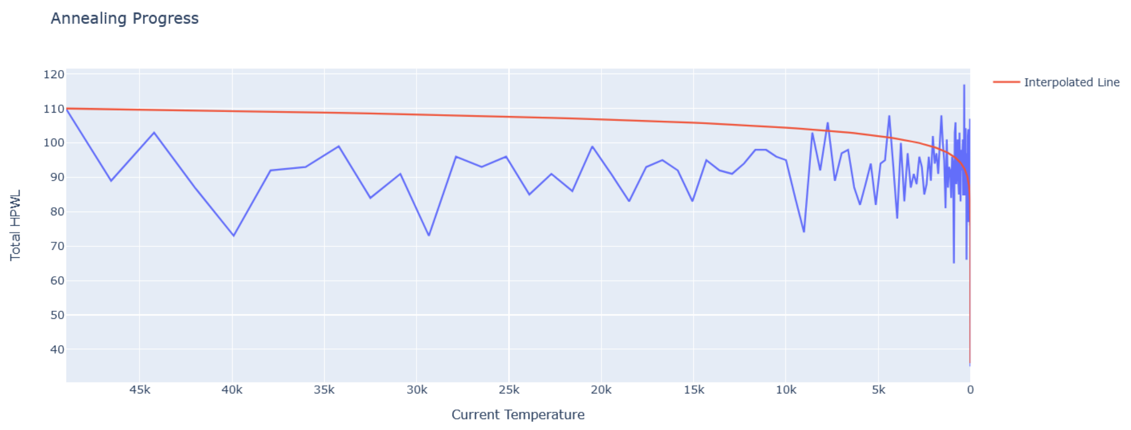
For the results of our annealing algorithm, we started by running the testcases that are provided in the handout. **Note**: for all the graphs of the HPWL vs temperature, we start the temperature on the x axis as reversed (from high to low) to show the progress of annealing and how the decrease of the temperature decreases the wire length.

4.1. d0.txt

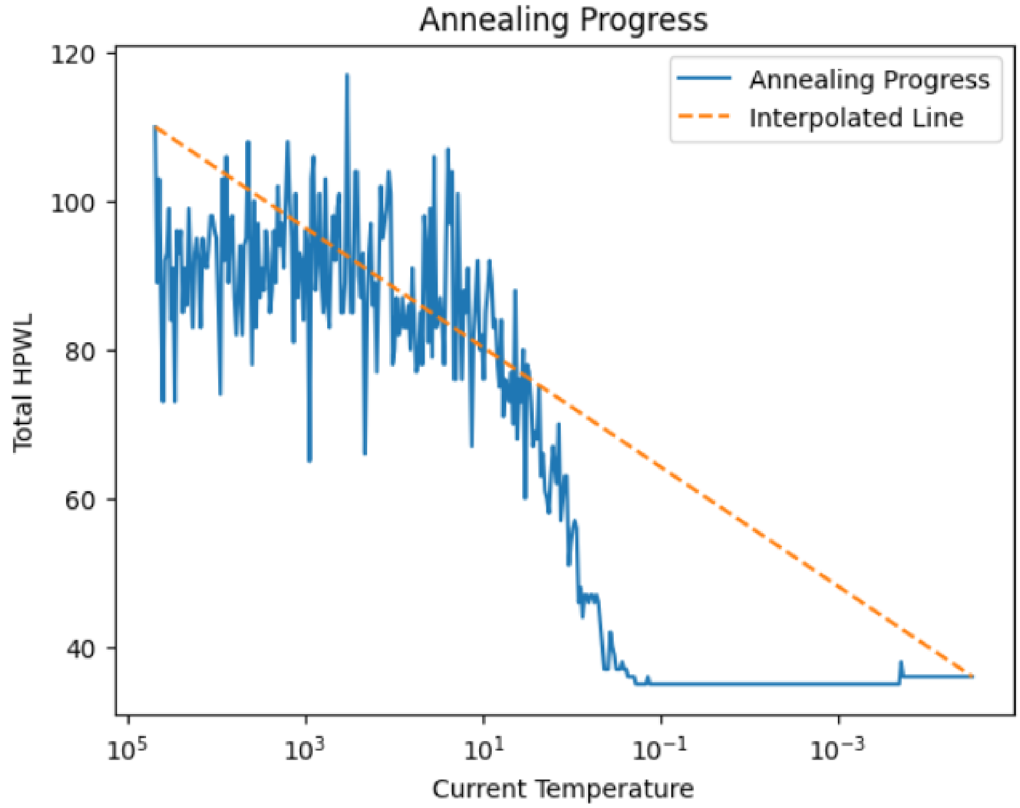
Below is what we got when we ran d0.txt on 0.95 cooling rate:

```
0013    0003    0017    ----    0009    0021    0018    0002
----    0000    0004    0001    0010    ----    0014    0019
----    ----    0006    0008    0012    0007    ----    0011
0015    ----    0023    0022    ----    0016    0005    0020
Time taken by function: 8226 microseconds
started annealing
Initial cost: 85
Final Cost: 35
Time taken by function: 446760 microseconds
----    ----    ----    0003    0014    0020    0004    ----
----    0016    0015    0008    0006    0022    0010    0000
----    0013    0007    0023    0019    0021    0017    0005
----    0011    0012    0002    0001    0009    0018    ----

Binary representation:

00011110
01111111
01111111
01111110
```

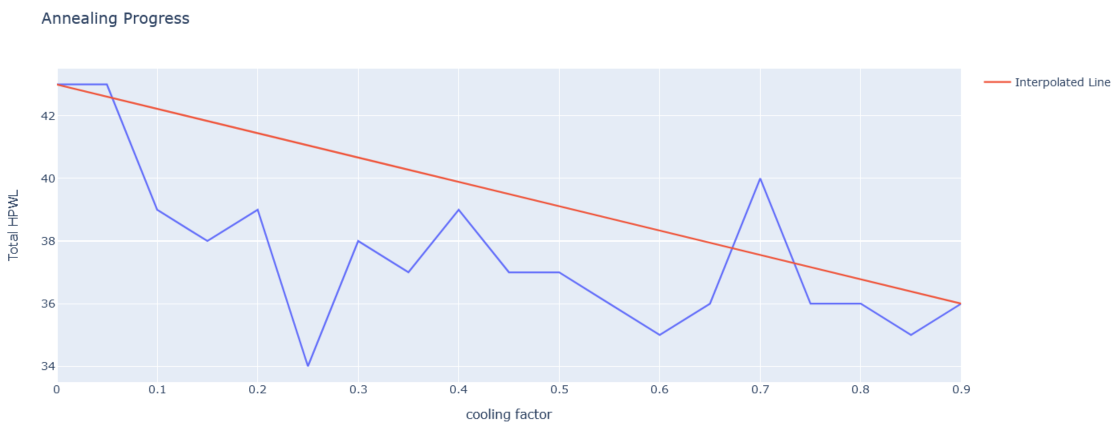
so as it's obvious, the initial random placement got adjusted to be optimal where the placed cells are clustered in the middle.
When we tried plotting the temperature versus the wire length during the annealing process this is what we got:-



And this shows how the wire length decreases when the temperature gets too small and the algorithm starts to converge and to verify this, we made the scale of the x axis (temperature) to be log scale to make it more obvious and readible



Next, we tried to use the same random seed and call the annealing algorithm on different cooling factor and see how this affects the final cost. Below is the resulted graph:-



4.1.1. Discussion

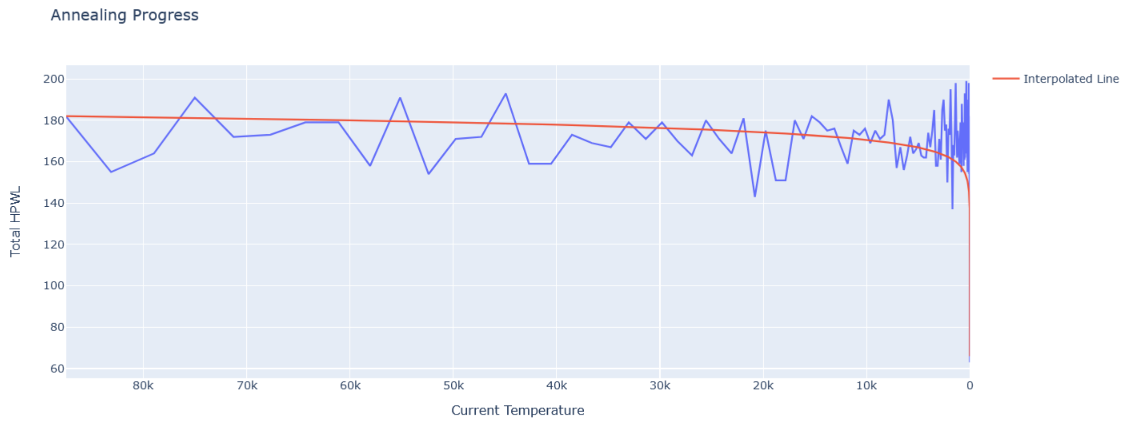
For the results obtained for this testcase, it's very obvious how the algorithm worked on placing the cells in their optimal positions by decreasing the wire length (HPWL) and leaving the corner spaces on the corners. Regarding the graph, we can see how the annealing progressed decreasing the wire length through the iterations and making it end at half it's initial cost. It's even more obvious on the graph having log scale on the x axis. Regarding the second graph, the results make sense as we go higher for the temperature factor, the final cost decreases. However, due to the small design the variation of the wire length is not that high.

4.2. d1.txt

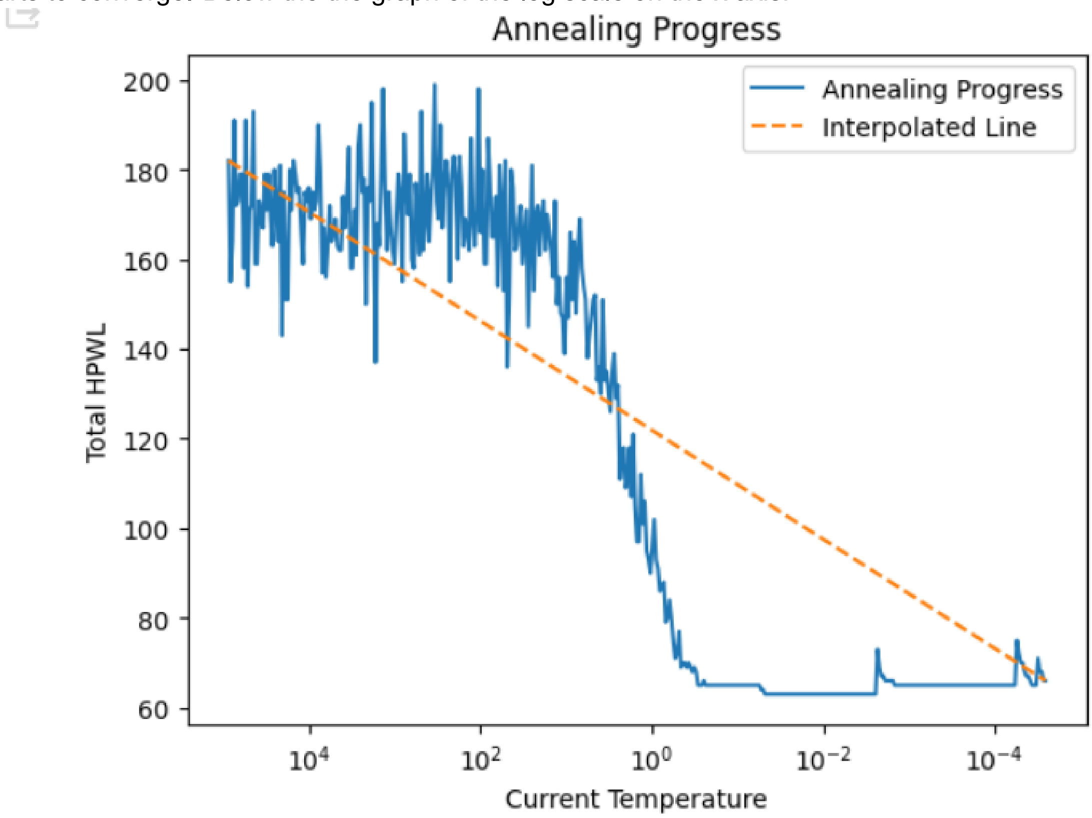
Similarly we did the same on d1.txt:

```
0020    0005    0021    0014    ----    0027    0019    0001
0018    0015    0007    0034    0024    0013    0031    0006
0003    0022    0030    0016    ----    0004    0032    0023
----    0010    ----    0035    0002    0017    0008    0029
0009    0026    0000    0025    0012    0028    0033    0011
Time taken by function: 6786 microseconds
started annealing
Initial cost: 156
Final Cost: 64
Time taken by function: 1008021 microseconds
0016    0005    0027    0035    0004    0006    0001    ----
0017    0031    0024    0021    0034    0025    0029    0008
----    0014    0022    0033    0019    0015    0007    0003
0013    0026    0030    0018    0002    0010    0028    0011
----    0000    0023    0032    0012    0009    0020    ----

Binary representation:

11111110
11111111
01111111
11111111
01111110

C:\Users\Muhammed\source\repos\Project61\x64\Debug\Project61.exe (process 20264) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

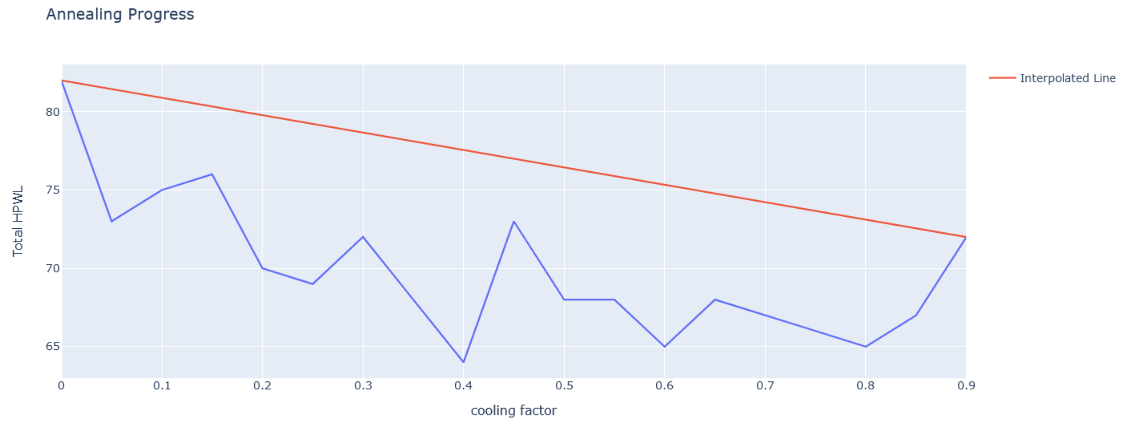
As expected, the initial random placement got adjusted to be optimal where the placed cells are clustered in the middle.
When we tried plotting the temperature versus the wire length during the annealing process this is what we got:-



And this shows how the wire length decreases when the temperature gets too small and the algorithm starts to converge. Below the the graph of the log scale on the x axis:-



Next, We did the other graph of the final cost vs the cooling rate and this is what we got:-

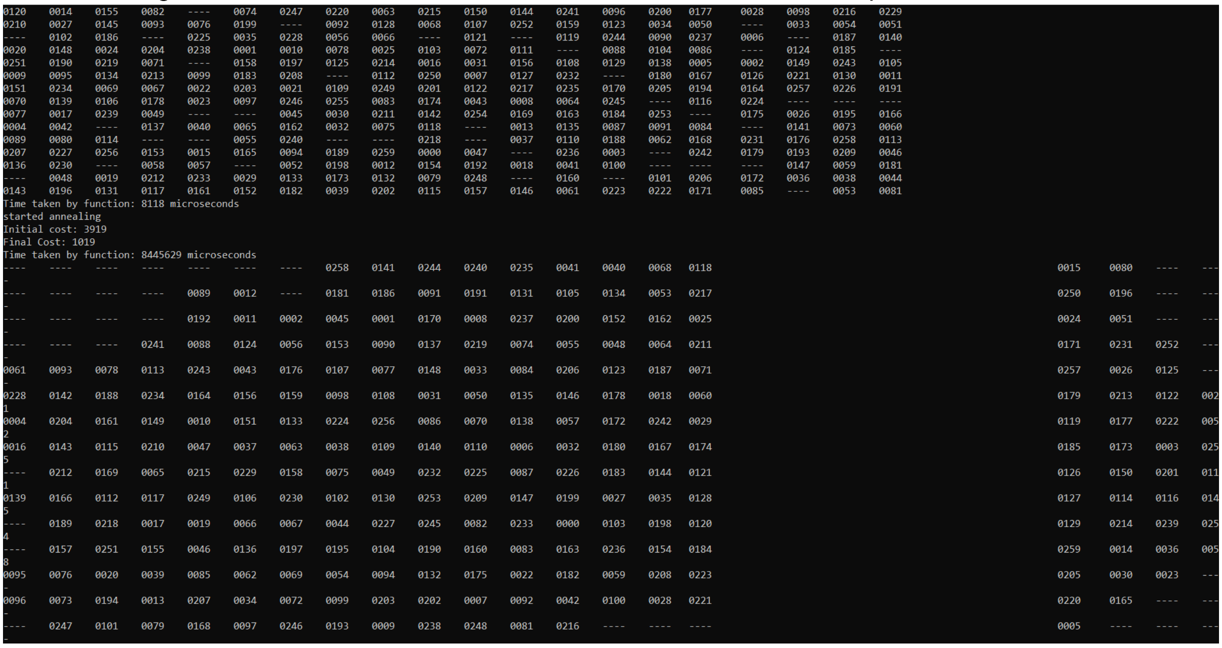


4.2.1. Discussion

The results also illustrate the algorithm effectiveness in optimizing the cell placement by minimizing the HPWL and allocating the cells strategically. The graph also demonstrates the annealing process by showcasing the steady decrease in the wire length and it's also more readible on the log scaled x-axis graph. The second graph also makes sense but similar to d0, the variation of the wire length is not obvious due to the small design.

4.3. d2.txt

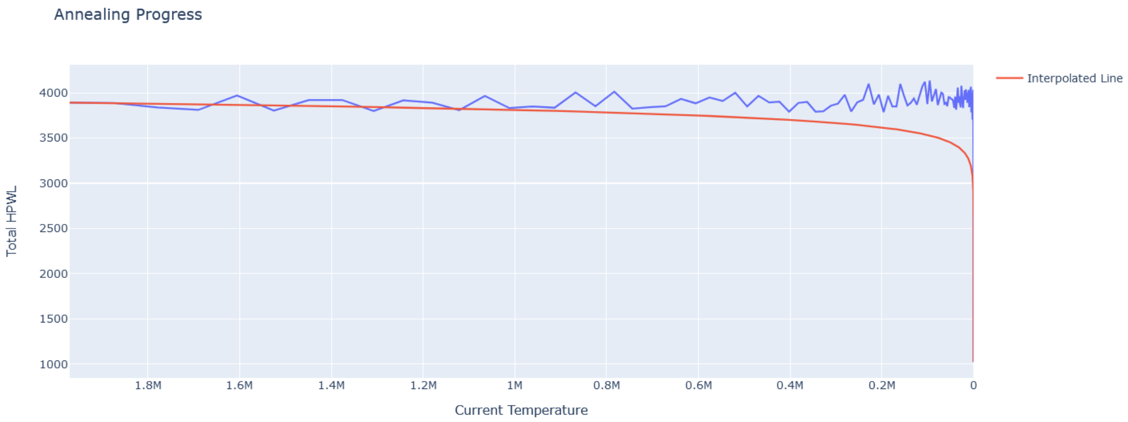
For the remaining test cases, , below are the results in the same order of the previous ones :-

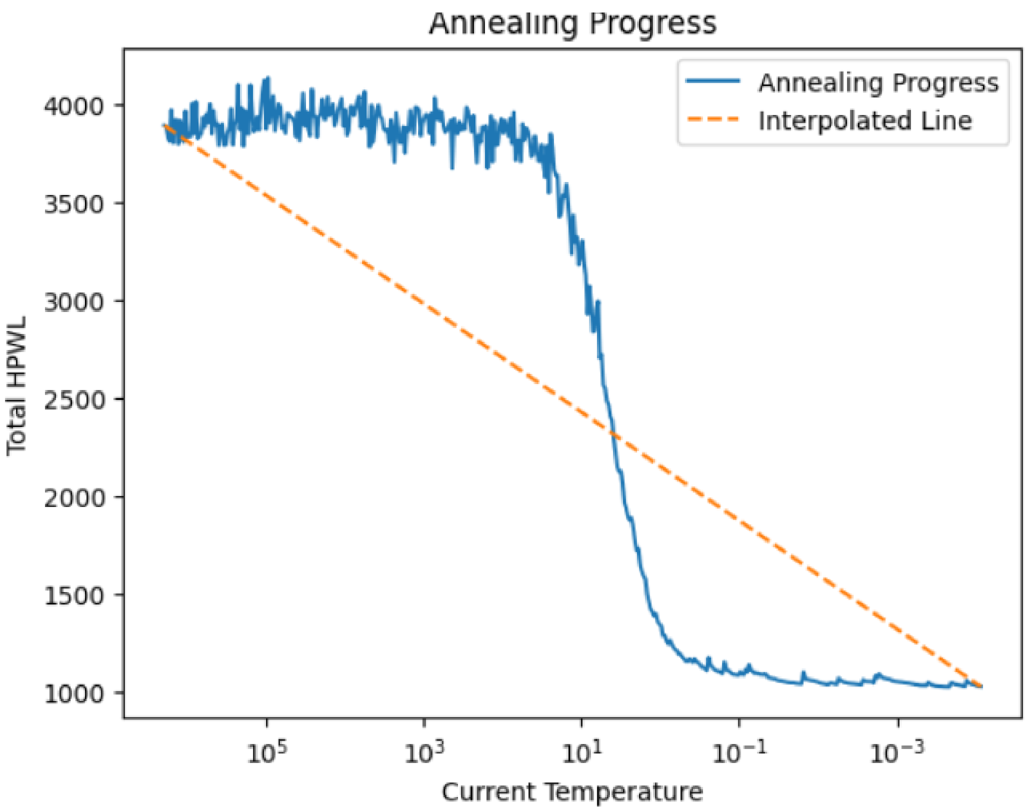


```
0120  0014  0155  0082  ----  0074  0247  0220  0063  0215  0150  0144  0241  0096  0200  0177  0028  0098  0216  0229
0210  0027  0145  0093  0076  0199  ----  0092  0128  0068  0107  0252  0159  0123  0034  0050  ----  0033  0054  0051
----  0102  0186  ----  0225  0035  0228  0056  0066  ----  0121  ----  0119  0244  0090  0237  0006  ----  0187  0140
0020  0148  0024  0204  0238  0001  0010  0078  0025  0103  0072  0111  ----  0088  0104  0086  ----  0124  0185  ----
0251  0190  0219  0071  ----  0158  0197  0125  0214  0016  0031  0156  0108  0129  0138  0005  0002  0149  0243  0105
0009  0095  0134  0213  0099  0183  0208  ----  0112  0250  0007  0127  0232  ----  0180  0167  0126  0221  0130  0011
0151  0234  0069  0067  0022  0203  0021  0109  0249  0201  0122  0217  0235  0170  0205  0194  0164  0257  0226  0191
0070  0139  0106  0178  0023  0097  0246  0255  0083  0174  0043  0008  0064  0245  ----  0116  0224  ----  ----  ----
0077  0017  0239  0049  ----  ----  0045  0030  0211  0142  0254  0169  0163  0184  0253  ----  0175  0026  0195  0166
0004  0042  ----  0137  0040  0065  0162  0032  0075  0118  ----  0013  0135  0087  0091  0084  ----  0141  0073  0060
0089  0080  0114  ----  ----  0055  0240  ----  ----  0218  ----  0037  0110  0188  0062  0168  0231  0176  0258  0113
0207  0227  0256  0153  0015  0165  0094  0189  0259  0000  0047  ----  0236  0003  ----  0242  0179  0193  0209  0046
0136  0230  ----  0058  0057  ----  0052  0198  0012  0154  0192  0018  0041  0100  ----  ----  ----  0147  0059  0181
----  0048  0019  0212  0233  0029  0133  0173  0132  0079  0248  ----  0160  ----  0101  0206  0172  0036  0038  0044
0143  0196  0131  0117  0161  0152  0182  0039  0202  0115  0157  0146  0061  0223  0222  0171  0085  ----  0053  0081
Time taken by function: 8118 microseconds
started annealing
Initial cost: 3919
Final Cost: 1019
Time taken by function: 8445629 microseconds
----  ----  ----  ----  ----  ----  ----  0258  0141  0244  0240  0235  0041  0040  0068  0118  0015  0080  ----  ---
-
----  ----  ----  ----  0089  0012  ----  0181  0186  0091  0191  0131  0105  0134  0053  0217  0250  0196  ----  ---
-
----  ----  ----  ----  0192  0011  0002  0045  0001  0170  0008  0237  0200  0152  0162  0025  0024  0051  ----  ---
-
----  ----  ----  0241  0088  0124  0056  0153  0090  0137  0219  0074  0055  0048  0064  0211  0171  0231  0252  ---
-
0061  0093  0078  0113  0243  0043  0176  0107  0077  0148  0033  0084  0206  0123  0187  0071  0257  0026  0125  ---
-
0228  0142  0188  0234  0164  0156  0159  0098  0108  0031  0050  0135  0146  0178  0018  0060  0179  0213  0122  002
1
0004  0204  0161  0149  0010  0151  0133  0224  0256  0086  0070  0138  0057  0172  0242  0029  0119  0177  0222  005
2
0016  0143  0115  0210  0047  0037  0063  0038  0109  0140  0110  0006  0032  0180  0167  0174  0185  0173  0003  025
5
----  0212  0169  0065  0215  0229  0158  0075  0049  0232  0225  0087  0226  0183  0144  0121  0126  0150  0201  011
1
0139  0166  0112  0117  0249  0106  0230  0102  0130  0253  0209  0147  0199  0027  0035  0128  0127  0114  0116  014
5
----  0189  0218  0017  0019  0066  0067  0044  0227  0245  0082  0233  0000  0103  0198  0120  0129  0214  0239  025
4
----  0157  0251  0155  0046  0136  0197  0195  0104  0190  0160  0083  0163  0236  0154  0184  0259  0014  0036  005
8
0095  0076  0020  0039  0085  0062  0069  0054  0094  0132  0175  0022  0182  0059  0208  0223  0205  0030  0023  ---
-
0096  0073  0194  0013  0207  0034  0072  0099  0203  0202  0007  0092  0042  0100  0028  0221  0220  0165  ----  ---
-
----  0247  0101  0079  0168  0097  0246  0193  0009  0238  0248  0081  0216  ----  ----  ----  0005  ----  ----  ---
-
```

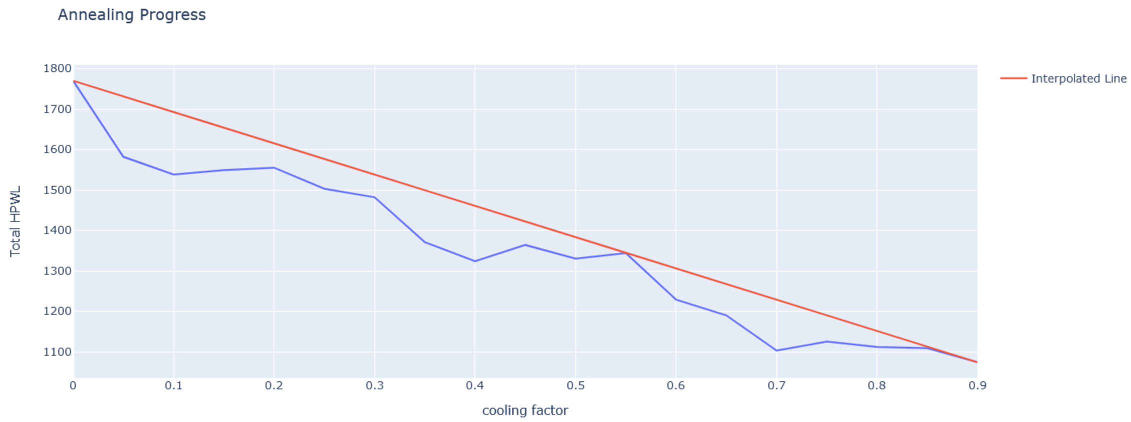
```
Binary representation:

00000001111111111100
00001101111111111100
00001111111111111100
00011111111111111110
11111111111111111110
11111111111111111111
11111111111111111111
11111111111111111111
01111111111111111111
11111111111111111111
01111111111111111111
01111111111111111111
11111111111111111110
11111111111111111100
01111111111110001000
```

The first graph:





The second graph:



4.3.1. Discussion

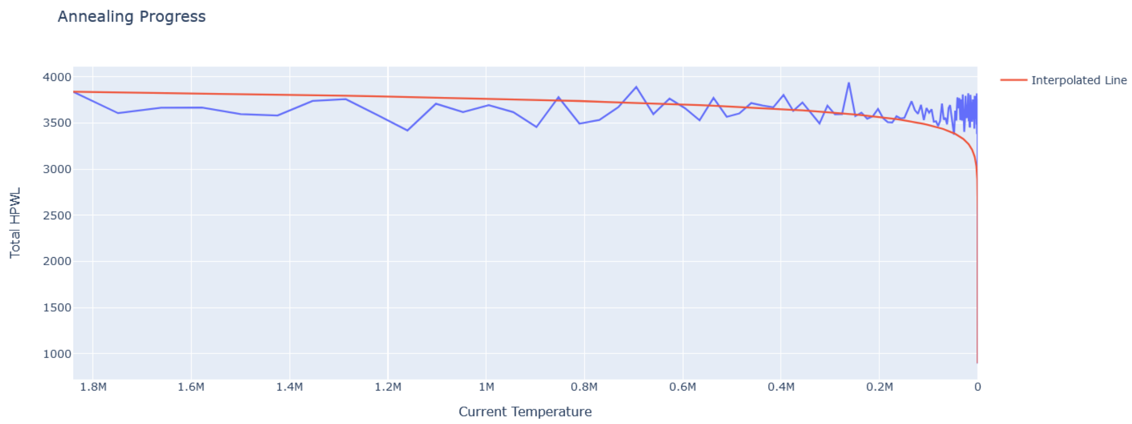
The outcomes here as well confirm the efficiency of the annealing algorithm and how it placed the cells effectively decreasing the wire length. The first graph also shows this in both its versions, the normal and log scale ones. The second graph in this test case starts to make more sense as the lowest cost corresponds the highest temperature factor.

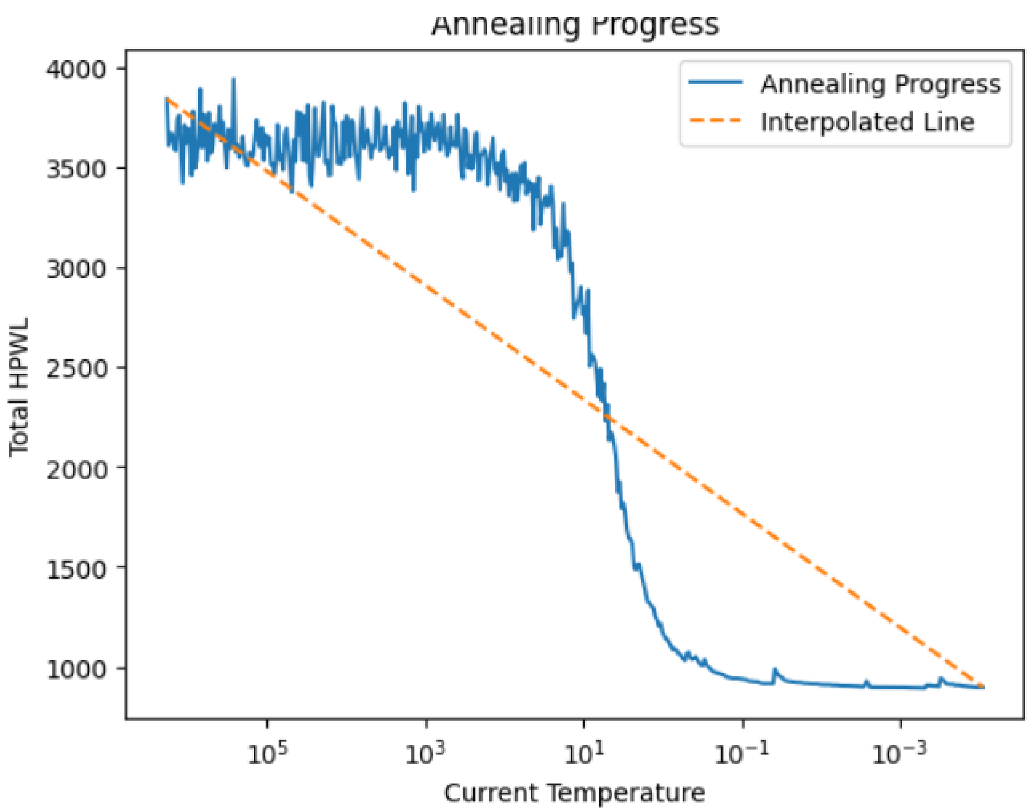
4.4. d3.txt

```
0204 0155 0102 0068 ---- 0166 ---- 0074 0025 ---- ---- 0186 0132 ---- ---- 0029 ---- 0051 ---- 0087 0107 0071 0159 ---- 0136
0122 ---- ---- ---- ---- 0140 ---- ---- 0056 0200 0023 0006 0163 0090 ---- ---- ---- 0187 ---- 0058 0026 0088 ---- ---- ----
0099 ---- 0049 ---- 0011 0125 0143 ---- 0001 ---- 0141 0180 ---- ---- ---- ---- 0181 0116 ---- ---- ---- 0193 0177 ---- 0197
0021 0133 ---- ---- 0169 0083 0036 ---- 0152 ---- 0196 0195 0175 ---- ---- 0054 0156 0174 0212 ---- 0149 0048 0050 0108 0007
0191 0017 0095 ---- ---- ---- ---- ---- ---- 0106 0082 ---- 0207 ---- 0192 ---- 0094 0135 ---- ---- 0183 0084 0081 0069 0034
0057 ---- 0033 ---- ---- 0118 ---- 0170 0063 0176 0158 0045 ---- ---- 0097 0119 0052 0038 0103 ---- 0154 0096 0168 0070 0160
---- ---- ---- ---- ---- ---- 0126 ---- ---- 0105 0065 0008 ---- ---- ---- 0046 ---- ---- 0129 ---- ---- 0037 0172 0060 0075
0114 ---- 0150 0010 0123 ---- ---- ---- 0043 0005 0134 ---- 0138 0205 ---- 0153 ---- 0053 ---- ---- ---- ---- 0144 0027 0003
---- 0202 0066 0100 0039 ---- ---- ---- 0161 0198 0024 0210 ---- 0131 ---- 0062 ---- ---- 0041 ---- 0109 0022 ---- 0190 0185
---- ---- 0162 ---- ---- ---- ---- 0211 0009 0044 ---- ---- 0012 0086 ---- 0002 0173 0179 ---- ---- 0072 ---- 0167 ---- ----
0111 ---- ---- 0093 ---- 0061 0164 ---- ---- ---- 0091 0110 ---- 0127 ---- ---- 0203 ---- ---- 0004 0194 0182 ---- 0171 0030
---- ---- ---- 0089 ---- ---- ---- ---- ---- 0078 0028 ---- 0199 0112 ---- 0073 0080 0032 0124 0064 0079 ---- 0013 0101 0042
0031 ---- 0148 0035 0016 0117 0184 ---- 0104 ---- 0098 ---- 0206 0145 0208 ---- 0067 0151 0130 ---- 0076 ---- ---- 0157 0188
---- ---- 0120 ---- 0018 0201 ---- 0137 ---- 0047 ---- ---- ---- 0085 ---- ---- 0019 0115 ---- 0077 0015 ---- 0113 0055 ----
0209 ---- ---- 0147 ---- ---- 0020 0000 0139 0189 0092 ---- 0040 0059 0146 0142 0014 0121 ---- ---- ---- 0165 0128 ---- 0178
Time taken by function: 8326 microseconds
started annealing
Initial cost: 3682
Final Cost: 909
Time taken by function: 9375837 microseconds
---- ---- ---- ---- ---- ---- ---- 0069 ---- 0009 ---- 0008 0143 0070 0061 0064 ---- ---- 0132 ---- ---- ---- ---- ---- ----
---- ---- ---- 0048 ---- 0098 ---- 0010 0136 0000 ---- 0036 0026 0111 0043 0072 0139 0115 0032 0170 0211 ---- ---- ---- ----
---- ---- 0001 ---- 0178 0184 0025 0134 0204 0095 0121 0012 0038 0068 0103 0091 0176 0189 0114 0033 0100 ---- ---- ---- ----
---- ---- ---- 0179 0005 0092 0059 0099 0166 0194 0090 0044 0003 0164 0073 0094 0086 0147 0041 0177 0023 ---- ---- ---- ----
---- ---- 0022 0034 0105 0021 0085 0168 0205 0152 0210 0202 0198 0035 0167 0083 0117 0037 0116 0195 0065 ---- ---- ---- ----
---- ---- 0102 0080 0128 0055 0200 0004 0129 0006 0190 0173 0209 0097 0113 0056 0140 0142 0016 0024 0067 ---- ---- ---- ----
---- ---- 0046 0171 0203 0145 0106 0197 0017 0137 0191 0057 0040 0188 0089 0007 0196 0052 0110 0045 0082 ---- ---- ---- ----
---- ---- 0062 0028 0192 0174 0175 0076 0180 0131 0185 0148 0156 0011 0135 0096 0186 0122 0123 0187 0053 ---- ---- ---- ----
---- ---- 0181 0049 0050 0093 0183 0130 0206 0075 0031 0158 0018 0138 0078 0019 0029 0079 0042 0157 0172 ---- ---- ---- ----
---- ---- 0161 0144 0087 0108 0163 0084 0182 0201 0027 0030 0155 0051 0149 0212 0013 0146 0160 0159 0127 ---- ---- ---- ----
---- ---- 0071 0208 0077 0020 0088 0154 0074 0151 0104 0199 0112 0133 0119 0118 0124 0002 0107 0125 0014 0169 ---- ---- ----
---- ---- 0081 ---- 0063 0109 0054 0101 0066 0162 0039 0193 ---- 0207 0165 ---- 0150 0120 ---- ---- ---- ---- ---- ---- ----
---- ---- ---- ---- ---- 0058 ---- 0153 0047 0015 ---- 0141 ---- ---- ---- ---- 0126 0060 ---- ---- ---- ---- ---- ---- ----
---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
```

```
Binary representation:

0000000101011111001000000
0001010111011111111110000
0010111111111111111110000
0001111111111111111110000
0011111111111111111110000
0011111111111111111110000
0011111111111111111110000
0011111111111111111110000
0011111111111111111110000
0011111111111111111110000
0011111111111111111111000
0010111111110110110000000
0000010111010000110000000
0000000000000000000000000
0000000000000000000000000
```

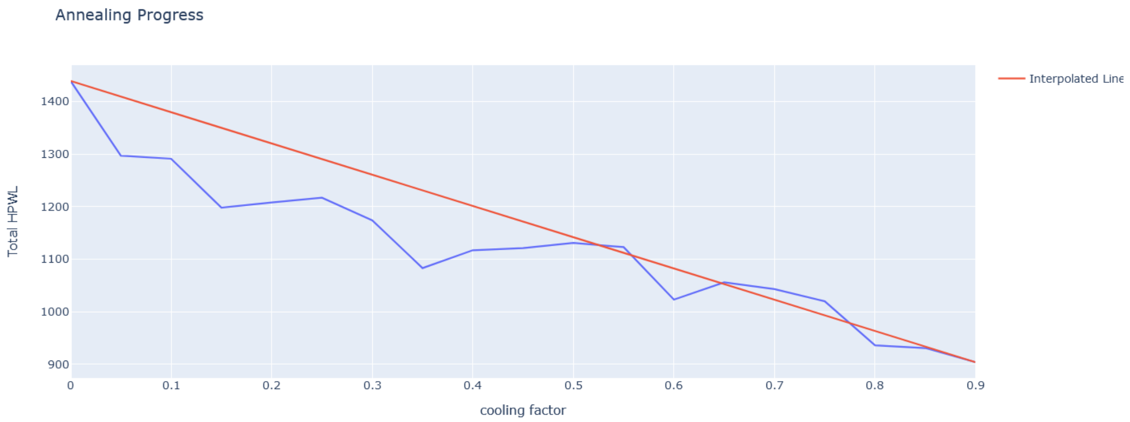
The first graph:





The second graph:

]



4.4.1. Discussion

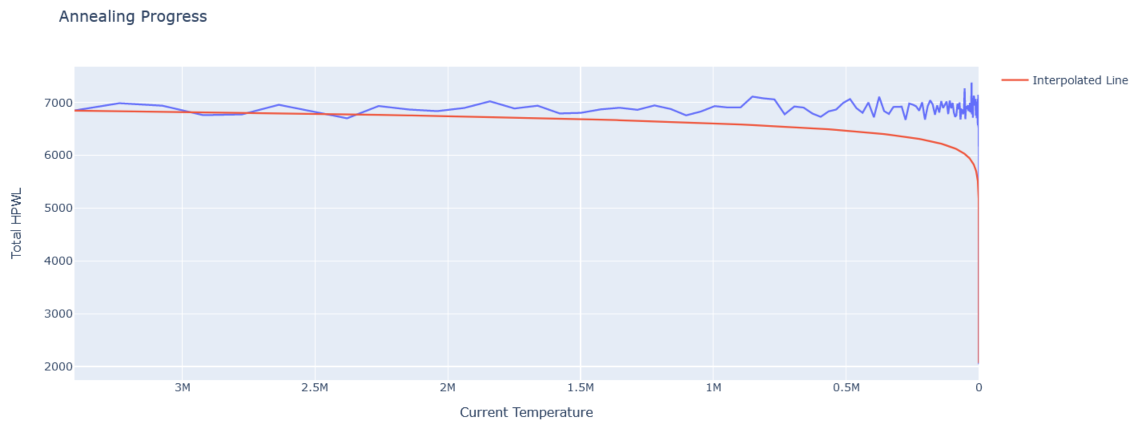
Similar to the previous testcases, the outcomes here as well confirm the efficiency of the annealing algorithm and how it placed the cells effectively decreasing the wire length. The first graph also shows this in both its versions, the normal and log scale ones. The second graph in this test case makes sense as the lowest cost corresponds the highest temperature factor.

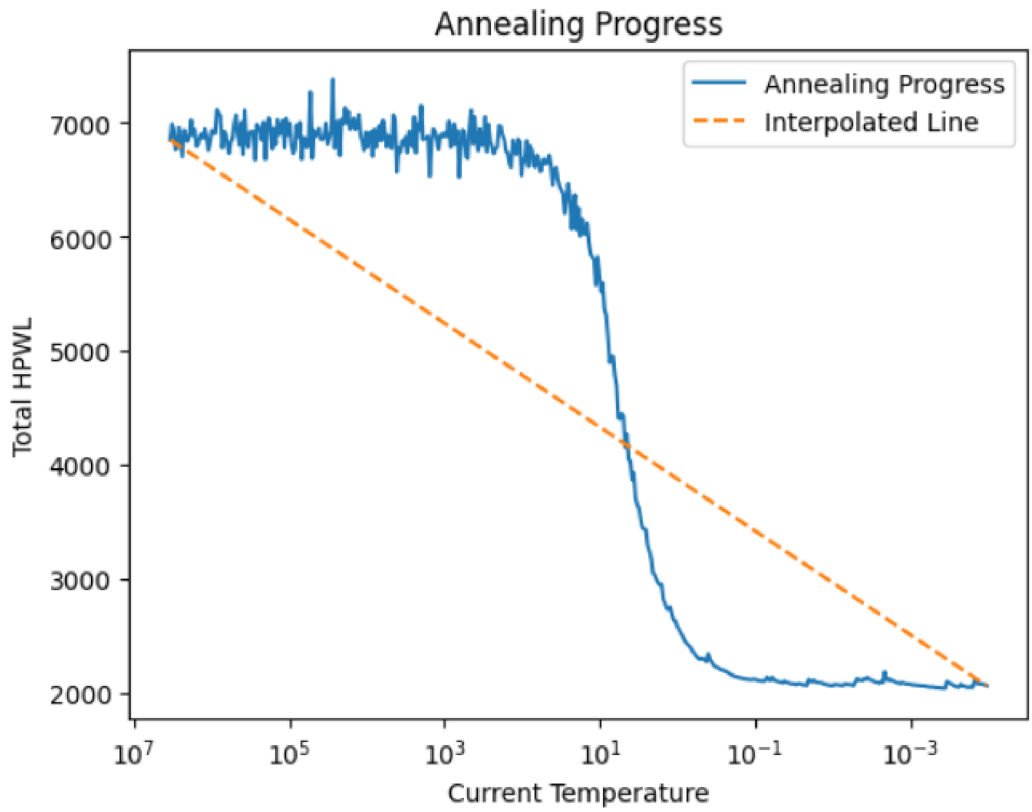
4.5. t1.txt

```
0279    0080    0323    0212    0143    0114    0252    0107    0070    0240    0331    0043    0065    0224    0208    0203    0280    0275    0357    0221    0215    0278    0141    0371    0268    0264
0146    0150    0325    0402    0007    0120    0175    0320    0378    0206    0049    0199    0127    0342    0058    0307    0102    0205    0061    0334    0265    0094    0169    0060    0153    0022
0105    0332    0359    0263    0076    0230    0051    0019    0081    0316    0113    0237    0388    0341    0087    0109    ----    0236    0315    0295    0010    0170    0400    0083    0147    0287
0328    0116    0364    0187    0086    ----    0282    0006    0068    0333    0184    0338    0290    0343    0310    0013    0311    0066    0029    0158    0344    0189    0214    0100    0260    0317
0154    ----    0273    0176    0380    0190    0063    0272    0082    0329    0145    0191    0042    0308    0353    0073    0172    0243    0075    0182    0008    0361    0387    0139    0062    0064
0286    0111    0335    0108    0314    0375    0092    0140    0020    0119    0248    0300    0362    0133    0132    0276    0093    0038    0151    0101    0351    0396    0258    0012    0112    0017
0055    0249    0128    0312    0002    0035    0053    0291    0233    0285    0261    0370    0381    0078    0289    0238    0136    0106    0186    0372    0255    0069    0046    0377    0229    0148
0337    0071    0014    0059    0048    0379    0256    0363    0037    0117    0234    0373    0305    0350    0160    ----    0104    0072    0247    0027    0123    0374    0179    0228    0157    0018
0152    ----    0326    0217    0030    0122    0222    0166    0025    0225    0368    0016    0196    0384    0118    0360    0288    0089    0177    0266    0162    0198    0299    0354    0032    0036
0369    0188    0033    0034    0347    0041    0050    0045    0054    0026    0173    0313    0327    0156    0250    0304    0318    0074    0031    0235    0297    0336    0365    0005    0028    0057
0322    0385    ----    0015    0193    0283    0088    0358    0103    0168    0155    0294    0301    0009    0267    0163    0110    0241    0271    0339    0135    0319    0281    0210    0277    0270
0223    0340    0209    0401    0382    0309    0197    0121    0137    0171    0395    0011    0355    ----    0178    ----    ----    0251    0376    0095    0192    0262    0056    0115    0227    0330
0367    0242    0181    0213    0356    0321    0091    0274    0392    0021    0386    0167    0349    0293    ----    0200    0324    0040    0298    0389    0194    0383    0391    0346    0097    0134
0077    0393    0211    0138    0306    0096    0296    0397    0394    0003    0131    0218    0239    ----    0047    0399    0023    0202    0090    0180    0164    0219    0390    0039    0079    0216
0253    0098    0302    0161    0144    0099    0142    0259    ----    0231    0124    0125    0207    0149    0366    0067    0245    0244    0024    0345    0084    0174    0001    0183    0126    0165
0129    0269    0085    0257    0159    0220    0052    0232    0246    0352    0201    0185    0195    0004    0044    0292    0226    0254    0398    0130    0303    0204    0348    0284    ----    0000
Time taken by function: 7329 microseconds
started annealing
Initial cost: 6973
Final Cost: 2074
Time taken by function: 12208563 microseconds
----    ----    ----    0090    0029    ----    0379    0160    0220    0096    0346    0332    0152    0033    0021    0257    0250    0232    0234    0036    0030    0194    0086    0124    ----    0011
----    0380    0122    0017    0351    0078    0032    0106    0387    0384    0176    0383    0378    0238    0037    0225    0084    0365    0218    0318    0317    0328    0166    0196    0343    0319
0002    0008    0066    0087    0134    0211    0367    0198    0126    0283    0117    0244    0312    0269    0258    0151    0006    0123    0183    0329    0146    0092    0261    0094    0370    0339
0173    0104    0248    0233    0071    0167    0182    0357    0054    0273    0200    0103    0120    0098    0398    0044    0178    0242    0201    0240    0324    0114    0300    0024    0097    0203
0363    0043    0199    0316    0150    0392    0285    0247    0276    0020    0018    0111    0185    0109    0304    0260    0278    0214    0305    0131    0217    0227    0127    0076    0206    0135
0116    0298    0067    0382    0197    0321    0119    0005    0031    0373    0275    0397    0229    0265    0230    0205    0376    0372    0288    0172    0049    0310    0063    0223    0309    0148
0377    0385    0356    0162    0115    0314    0155    0108    0110    0243    0083    0145    0189    0290    0280    0219    0026    0099    0130    0208    0342    0046    0088    0345    0040    0059
0079    0025    0140    0052    0336    0195    0163    0010    0294    0207    0235    0296    0340    0075    0082    0331    0401    0101    0168    0366    0362    0293    0186    0272    0064    0154
0169    0209    0156    0074    0369    0255    0287    0270    0000    0027    0095    0222    0289    0045    0315    0291    0129    0396    0013    0048    0375    0237    0302    0386    0181    ----
0353    0080    0128    0073    0333    0055    0374    0141    0292    0299    0274    0311    0056    0139    0161    0246    0391    0389    0323    0313    0359    0286    0279    0035    0136    0157
0371    0009    0102    0271    0354    0252    0105    0170    0295    0322    0213    0158    0307    0334    0147    0038    0277    0394    0239    0308    0149    0266    0259    0072    0153    0344
0224    0171    0241    0004    0347    0335    0070    0077    0215    0263    0281    0262    0187    0028    0267    0284    0349    0061    0326    0174    0253    0193    0352    0402    0249    0348
0175    0204    0060    0125    0113    0137    0212    0350    0022    0190    0226    0320    0301    0254    0400    0138    0068    0179    0107    0093    0144    0327    0297    0177    0001    0039
0393    0325    0118    0050    0184    0014    0381    0069    0143    0330    0231    0041    0100    0256    0306    0081    0065    0142    0007    0245    0132    0210    0360    0062    0112    ----
----    0053    0188    0089    0057    0303    0264    0251    0042    0368    0361    0364    0395    0202    0012    0003    0051    0121    0047    0164    0337    0390    0221    0388    0355    0016
----    0399    0133    0191    0034    0268    0180    0228    0085    0358    0192    0091    0165    0159    0019    0023    0338    0015    0216    0282    0341    0058    0236    ----    ----    ----
```

```
Binary representation:

00011011111111111111111101
01111111111111111111111111
11111111111111111111111111
11111111111111111111111111
11111111111111111111111111
11111111111111111111111111
11111111111111111111111111
11111111111111111111111111
11111111111111111111111110
11111111111111111111111111
11111111111111111111111111
11111111111111111111111111
11111111111111111111111111
11111111111111111111111110
01111111111111111111111111
01111111111111111111111000
```

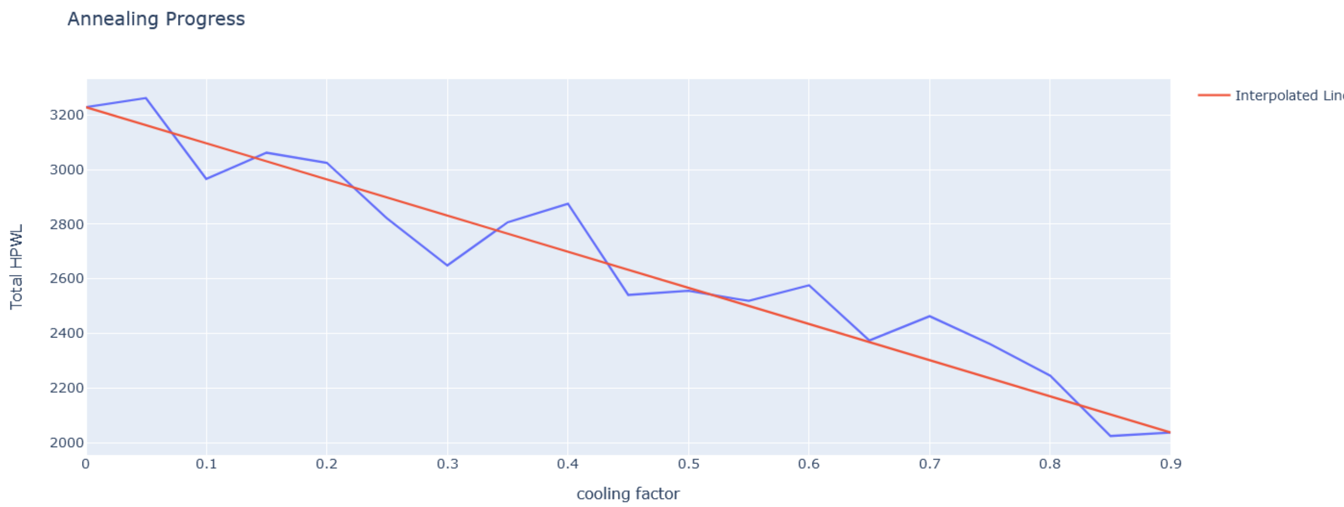
The first graph:





The second graph:

[46]



4.5.1. Discussion

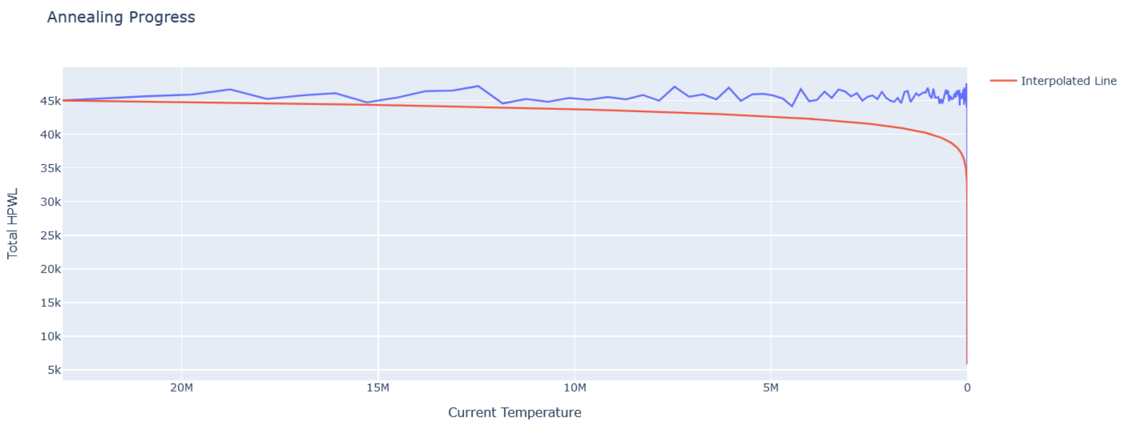
As a further confirmation, the outcomes here as well confirm the efficiency of the annealing algorithm and how it placed the cells effectively decreasing the wire length. The first graph also shows this in both its versions, the normal and log scale ones. The second graph in this test case makes more sense as the lowest cost corresponds the highest temperature factor. Additionally, the variations appear more when the design gets larger.
For the remaining two testcases, we will be showing the binary representation, the final cost and initial cost only as the design is huge

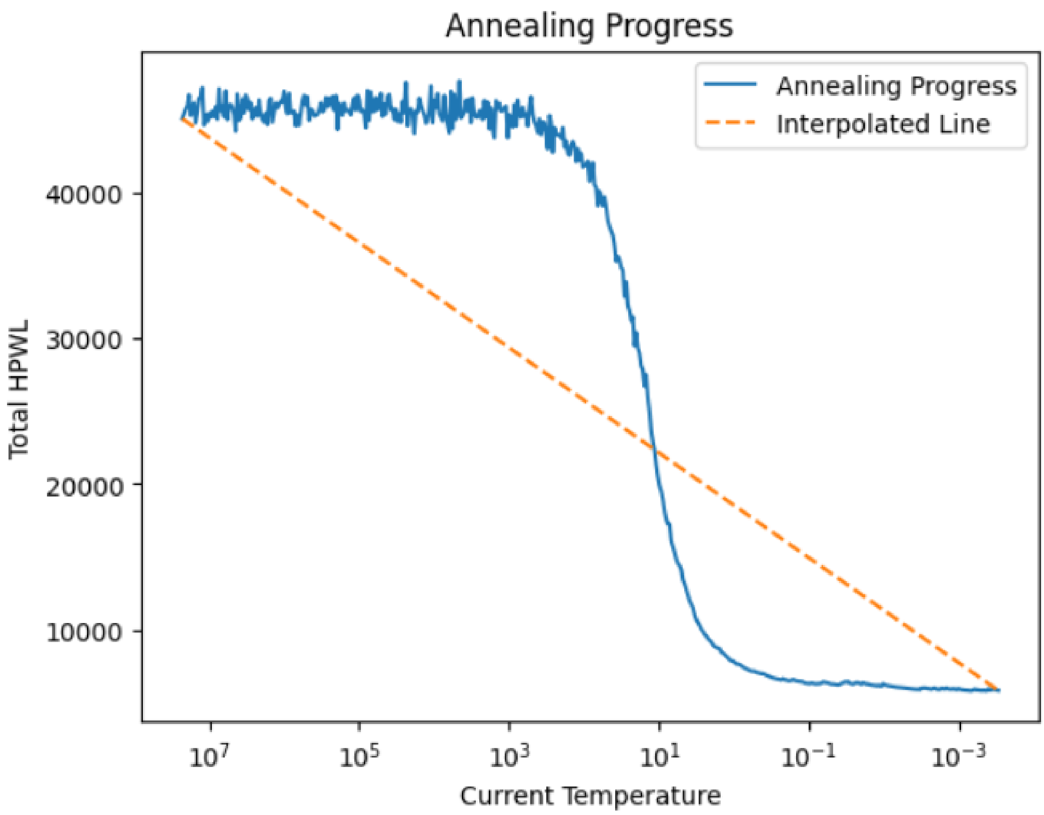
4.6. t2.txt

```
started annealing
Initial cost: 45587
Final Cost: 5048
```

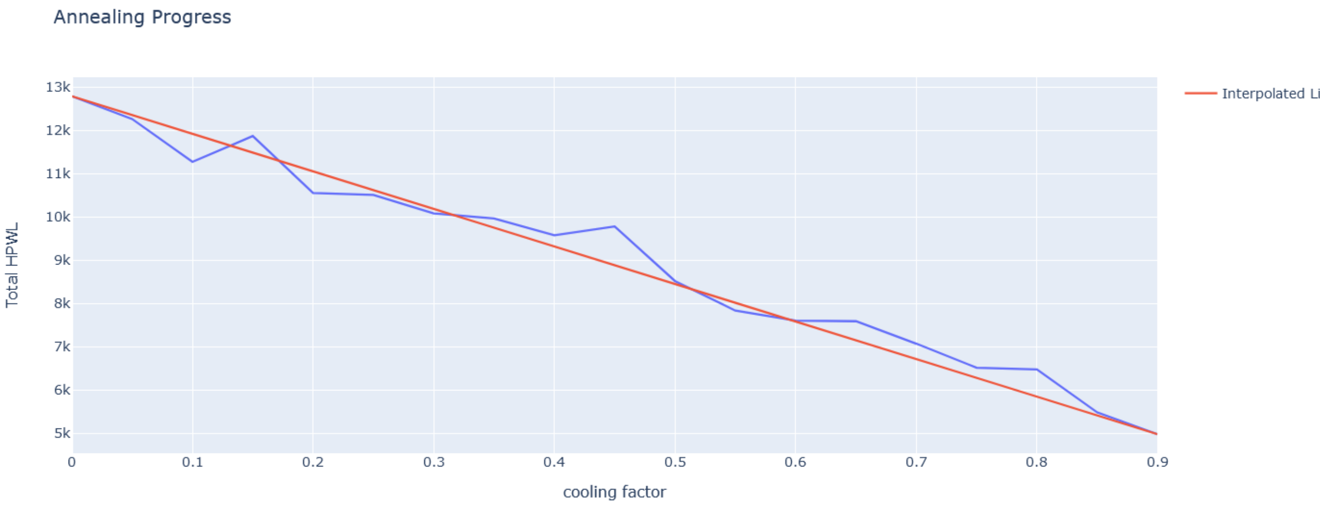
```
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000001100000000000000000000000000000000000000000000000
0000000000000000000100100000000000000000000000000000000000000000000000
0000000000000000000010000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000001000010100000000000000000000000000000000000000000000000
0000000000000000001110100000000000000000000000000100000000000000000000
0000001001110111101111110000010000000000000010000000000000000000000000
0000000110111111101111011110010000000000000111111101110000000000000000
0000000011111111111111110001110000000000000111111111110000000000000000
0000000111111111111111111111110000000011010111111111101000000000000000
0000000111111111111111111111111000000000011111011111111100000000000000
0000000001111111111111111111111010011111101111111111110000001000000000
0000000001111111111111011111000000000011011111110111111000000000000000
0000000001111111111111111111011000000000111111111111111110000000000000
0000000111111011111111111111111000000000011110111111111111110000110000000
0000000001111101011111111111001000000000001111111111110111101101100000000
0000000000000011111110111111110000000001110011111111111111110110110000000
0000000000000110011111111111111000000000111011111111111111111111111000000
0000000000001100111111111111111111000000011101111111111111111110111000000
0000000000000011101111111111111000000000001110110111111111111101100100000
0000000000000011111111111111111000000000000001011111111111111111111100000000
0000000000000001101111111111111000000000110001111111111111111111111000000
0000000000000011111111111111011000000000000011111111111111111111111110000000
0000000000000001101111111111111000000000101110111111111111111111111110000000
0000000000000001111111111111111010000000100101011111111111111111111111000000
0000000000000001011111011111011000000011000011111111111111111010100000000
0000000000000011111111111110000000000111001111111110011011110010000000000
0000000000000011111011111110100000001111001111111111110110000000000000
0000000000000011111111111101010000001110000101111011111110110000000000
0000000000000001111101111111100000000000000101011111111111011000000000
0000000000000011111111011111000000000100000000111111111111110000000000
0000000000000110101111111110000000000000000000011011101111101000000000000
0000000000000010111111111100000000000000000000101011111111010000000000000
0000000000000011111111101110000000000000000000100001011110101000000000000
0000000000001001011111110001000000000000000000000100001110000000000000
0000000000000000011100000000000000000000000000000000001000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
```

The first graph:





The second graph:

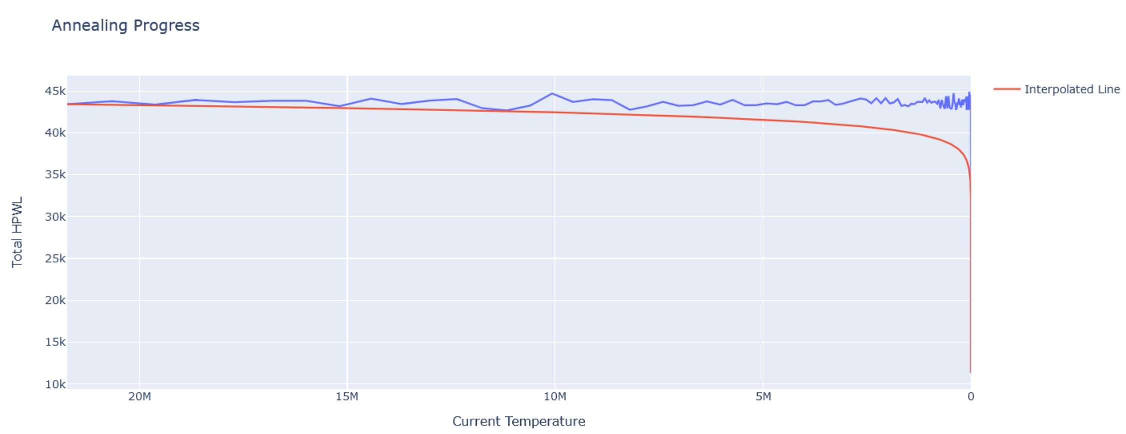


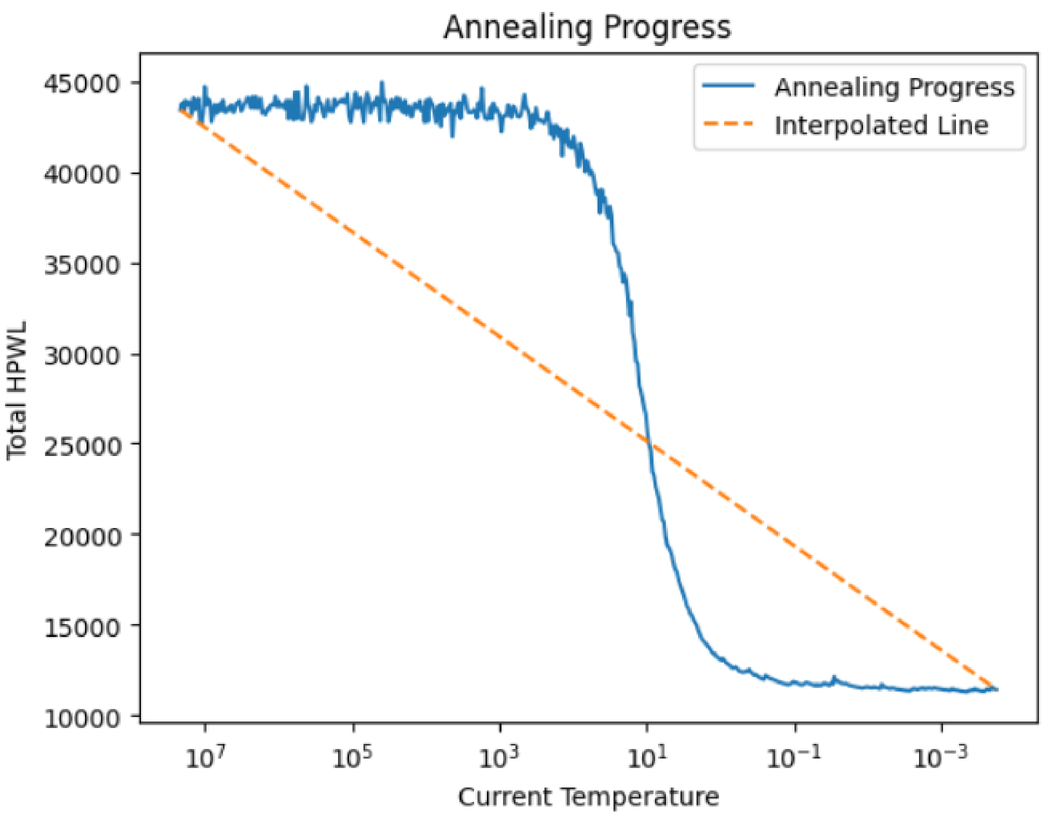
4.7. t3.txt

```
Initial cost: 43979
Final Cost: 11332
```

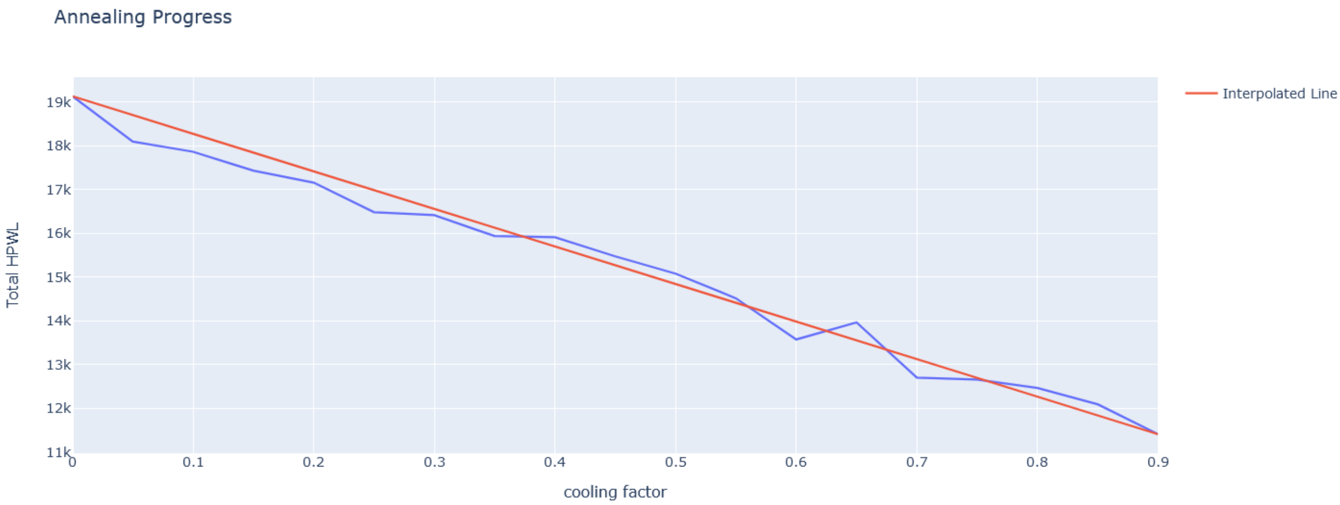
```
Binary representation:

00100010110111111111101101111011111111011111011000
00011111111111111111111101001111111111111111111100
00011111111111111111111111111111111111111111110000
01111111111111111111111111111111111111111111111100
01111111111111111111111111111111111111111111111110
01111111111111111111111111111111111111111111111110
11111111111111111111111111111111111111111111111111
01111111111111111111111111111111111111111111111110
01111111111111111111111111111111111111111111111110
11011111111111111111111111111111111111111111111110
01111111111111111111111111111111111111111111111110
00011111111111111111111111111111111111111101111110
01111111111111111111111111111111111111111111111110
01111111111111111111111111111111111111111111111111
11111111111111111111111101111111111111111111111110
11111111111111111111111111111111111111111111111111
01111111111111111111111111111111111111111111111100
00111111111111111111111111111111111111111111111101
00111111111111111111111111111111111111111111111110
01111111111111111111111111111111111111111111110111
00111111111111111111111111111111111111111111111110
11111111111111111111111111111111111111111111111110
00111111111111111111111111111111111111111111111100
00111111111111111111111111111111111111111111111000
00011111111111111111111111111111111111111111111100
01111111111111011111111111111111111111111111110101
01111111111111111110111111101111111111111111110100
00100000101111111111110111111111111101011110110000
```

The first graph:





The second graph:



4.7.1. Discussion for t2 and t3

As we saw what happened in the previous testcases, the cost considerably decreased for such a big design which shows how effective the annealing algorithm is. Additionally, the graph shows how the HPWL decreased as we progressed with time. Finally, the second graph for each one makes even more sense and even more than the other testcases as these were the largest two testcases and we show how the temperature factor affects the final cost and we saw how this varied with different cooling rates.

5

Conclusion

The simulated annealing experiments conducted in this project have provided valuable insights into the optimization of wire length in electronic circuit layouts. The primary objective was to analyze the impact of changing the cooling factor on the wire length during the annealing process.

The results clearly demonstrate that the total wire length consistently decreases throughout the annealing process until an acceptable solution is reached. This behavior aligns with the fundamental principles of simulated annealing, where the algorithm explores a wide solution space, initially accepting worse solutions and gradually converging towards optimal configurations.

Interestingly, the final wire length was observed to increase as the cooling factor decreases. This phenomenon suggests a delicate balance between exploration and exploitation in the algorithm. A higher cooling factor allows for greater exploration of the solution space, potentially leading to the discovery of more optimal configurations. On the other hand, a lower cooling factor emphasizes exploitation, focusing on fine-tuning the solution. The trade-off between exploration and exploitation becomes evident in the observed relationship between the cooling factor and final wire length.

It is worth noting that the algorithm exhibits efficient performance, completing the optimization process in minimal time for the worst available test case. This efficiency is crucial for practical applications where quick turnaround times are essential in electronic design.

Additionally, the project has provided means of visualizing the annealing process. The generation of a GIF animation allows for a dynamic representation of how the algorithm explores and refines the solution space over time. This visualization aids in understanding the algorithm's behavior and can be a valuable tool for both analysis and communication of results.

In conclusion, the findings of this project contribute to the broader understanding of simulated annealing in the context of wire length optimization. The observed trade-off between cooling factor and final wire length, coupled with efficient algorithmic performance and visualization capabilities, opens avenues for further research and application in electronic design automation.