Digital Design II

Project II Report

Submitted by:

Omar Shaalan - Mostafa Sharkawy - Hend Makram

900193887-900184043-900191686

1. Abstract: -

Simulated Annealing is a popular optimization algorithm used to solve global optimization problems in various fields such as engineering, computer science, and machine learning. In our project we chose Python for the implementation of Simulated Annealing that involves generating an initial solution and iteratively finding new solutions by randomly perturbing the current solution and accepting or rejecting the new solution based on a probability determined by the temperature parameter. As the temperature decreases, the algorithm becomes more selective and converges towards the global minimum. The choice of Python provided several libraries such as tkinter, matplotlib, and random. This allowed us to implement the Simulated Annealing and visualize the output and the graphs, we also created a GIF that is generated to show the change in the iterations to the initial random solution. Unfortunately the choice of Python has one main drawback, it takes more time to run the program and the algorithm than it would have taken in other languages, such as C++. However, this is heavily related to the hardware of the machine running the program, so we tried to do such a program with C++, but we did not have the GIF part running, so we preferred to have more facilities to visualize the outputs as a GIF for easier data analysis. The flexibility of Simulated Annealing and its ability to handle non-differentiable and non-convex functions make it a powerful optimization technique that can be used to solve complex optimization problems.

2. Introduction: -

Simulated Annealing is a global optimization algorithm that is used to find the global minimum of a given function. It is a metaheuristic algorithm that is inspired by the physical process of annealing in metallurgy, where a material is heated and then slowly cooled to reduce its defects and increase its strength. In the context of optimization, the algorithm starts by generating an initial solution and then iteratively modifies it by randomly perturbing it in some way. The algorithm accepts the new solution if it improves the objective function value, but also has a probability of accepting solutions that are worse than the current solution. This probability is determined by the temperature parameter, which is gradually decreased during the annealing process. As the temperature decreases, the algorithm becomes increasingly selective and converges towards the global minimum. The annealing process can be run for a fixed number of iterations or until a certain stopping criterion is met. Simulated Annealing is a powerful optimization algorithm that can be used to solve a wide range of problems, including combinatorial optimization, machine learning, and engineering design, and in this project a detailed explanation will be given for our implementation. In this paper we will discuss the implementation of the code and how it works. In other words we will divide the code into sections, each section will contain one of the functions in the project itself. This will make it easier to understand the behavior of each and every function, and how they work. Moreover, data for the test cases provided will be offered and discussed too.

3. The Annealing Algorithm: -

In this section we will explain how the Annealing algorithm works. How it changes the solutions in each iteration and what it needs in order to do such a thing. Moreover, we will then start explaining how our implementation to the algorithm works, explaining the functions and the flow of the code.

1. How it works: -

We will start by discussing the temperature role in the algorithm. In the Simulated Annealing algorithm, the temperature parameter plays a crucial role in determining the probability of accepting a new solution that is worse than the current solution. The temperature parameter is gradually decreased during the annealing process, and as it decreases, the algorithm becomes more selective in accepting new solutions.

At the start of the annealing process, the temperature is set to a high value, and the algorithm accepts new solutions that are worse than the current solution with a relatively high probability. This allows the algorithm to explore the solution space more widely and avoid getting trapped in local minima. As the temperature decreases, the algorithm becomes increasingly selective and converges towards the global minimum.

The temperature controls the amount of randomness in the algorithm and determines how likely it is to accept a new solution that is worse than the current solution. When the temperature is high, the algorithm is more likely to accept worse solutions because it is exploring the solution space more widely. As the temperature decreases, the algorithm becomes more selective and tends to accept only better solutions.

Choosing an appropriate cooling schedule for the temperature is important for the success of the Simulated Annealing algorithm. If the temperature is cooled too quickly, the algorithm may converge too quickly to a suboptimal solution. If the temperature is cooled too slowly, the algorithm may take too long to converge to a solution.

Now as we understand how crucial the temperature parameter is we resume to understand how the rest of the algorithm works. The Simulated Annealing algorithm follows a simple iterative process that typically involves the following steps:

Perturbation: A new solution is generated by perturbing the current solution. The perturbation can be done in various ways, such as flipping a bit in a binary string, changing a value in a vector, or swapping two elements in a permutation.

Evaluation: The objective function is evaluated for the new solution to determine its quality. The objective function can be any function that we want to optimize, such as a cost function that we want to minimize or a reward function that we want to maximize.

Acceptance: The new solution is either accepted or rejected based on a probability distribution that is a function of the temperature and the magnitude of the difference between the objective function values of the current and proposed solutions. If the new solution is better than the current solution, it is always accepted. If the new solution is worse than the current solution, it is accepted with a probability that depends on the temperature and the magnitude of the difference between the objective function values.

Update: If the new solution is accepted, it becomes the current solution, and the temperature is updated according to a cooling schedule. The cooling schedule determines how quickly the temperature decreases over time, and it is critical for the success of the algorithm.

2.How we did it: -

random_initialize(): -

This function, called "random_initialize()" , takes a filename as an input. The function opens the file in read mode, reads the contents of the file, and initializes a grid with the specified number of rows and columns, with -1 indicating empty cells. The function then creates a dictionary of sites to store the locations of the components on the grid. Next, the function randomly assigns a site ID to each component, ensuring that no two components occupy the same grid cell. The function prints the resulting grid and closes the input file. Finally, the function returns the grid, the dictionary of sites, and the number of components and nets. This function is needed, as it will be a part of the larger wrapper module/function called "annealing" discussed below.

ConnectionsGetter: -

This function is called "ConnectionsGetter()" that takes a file, connections, and cells as inputs. The function initializes variables to store the total number of nets and the connections between the cells. The function then reads the connections from the file and builds a dictionary called `All_connections` that stores the connections between the cells. The function reads the connections from the file one by one and stores them in a list called “total_nets”.
For each connection, the function extracts the cell numbers and adds the connection to the “All_connections” dictionary. The “All_connections” dictionary is a nested dictionary where each key represents a cell and its corresponding value is another dictionary that stores the connections between the cell and the nets. Finally, the function returns the “total_nets” list and the “All_connections” dictionary.

HPWLL: -

The "HPWLL" function takes two input arguments: "nets" and "dict". The function calculates the half-perimeter wire length (HPWL) of a given set of nets, where each net is a collection of cells represented by their x and y coordinates stored in the "dict" dictionary.

The function initializes an empty dictionary called "HPWL_dict" and a variable "hpwl" with a value of 0. It then uses a "for" loop to iterate over each element in the "nets" list, where each element represents a net. For each net, the function extracts the x and y coordinates of each cell in the net from the "dict" dictionary. The function calculates the width and height of the net by finding the difference between the maximum and minimum x and y coordinates, respectively. The function calculates the HPWL of the net by adding the net's width and height. It stores the HPWL of the net in the "HPWL_dict" dictionary, where the key is the index of the net in the "nets" list. The function adds the HPWL of the net to the "hpwl" variable. After iterating over all nets, the function returns two values: the total HPWL of all nets ("hpwl") and a dictionary ("HPWL_dict") that maps each net's index to its HPWL.

In summary, the "HPWLL" function is a useful tool for calculating the HPWL of a set of nets. It takes the nets and their corresponding cell coordinates as input, and computes the HPWL for each net by finding the width and height of the net, and adding them together. The function then returns the total HPWL for all nets, as well as a dictionary that maps each net's index to its HPWL.

HPWL: -

The "HPWL" function takes six input arguments: "size", "nets", "dict", "cf", "selected", "cL", and "HPWL_dict". The function initializes a set called "finished" to keep track of which nets have been processed. It also creates a copy of the input "HPWL_dict" dictionary called "HPWL_dict_new" and initializes a variable called "hpwl" to the value of "cL", which is a constraint on the total wire length.

The function then uses a nested "for" loop to iterate over each cell in the "selected" list and each net index in the "cf" dictionary that corresponds to that cell. For each net, the function checks if it has already been processed by checking if its index is in the "finished" set. If the net has not been processed, it adds its index to the set and updates the "hpwl" variable.

The function then calculates the width and height of the net by extracting the x and y coordinates of each cell in the net from the "dict" dictionary, finding the difference between the maximum and minimum x and y coordinates, and adding them together. The function then updates the HPWL of the net in the "HPWL_dict_new" dictionary and updates the "hpwl" variable by adding the new HPWL value.

After processing all cells and nets, the function returns the updated "hpwl" value and the updated "HPWL_dict_new" dictionary.

SWAP: -

The "swap" function takes four input arguments: "placement", "i1", "i2", and "d". The function is designed to swap the positions of two cells in a placement array and their corresponding positions in a dictionary of cell-to-coordinate mappings. The function creates a new list called "new_placement" by creating a copy of the "placement" list using a nested list comprehension. This is done to avoid modifying the original "placement" list. The function then extracts the values of the cells at the positions "i1" and "i2" in the "placement" list using indexing and assigns them to variables "cell1" and "cell2", respectively.

The function creates a new list called "selected" containing the non-empty cells in "cell1" and "cell2". It also creates a copy of the "d" dictionary called "new_dict". If both "cell1" and "cell2" are not empty, the function swaps their positions in the "new_dict" dictionary. If only "cell1" is not empty, the function updates its position in "new_dict" to "i2". If only "cell2" is not empty, the function updates its position in "new_dict" to "i1".

The function then swaps the positions of "cell1" and "cell2" in the "new_placement" list. Finally, the function returns the updated "new_placement" list,the updated "new_dict" dictionary, and the list of selected cells.

Annealing: -

The "annealing" function takes in a file name as an argument and performs simulated annealing optimization on the circuit design described in the file. The function begins by initializing the placement array randomly and extracting the number of cells and connections from the file. It then sets up the range of cooling rates given and initializes variables to store the temporary and final x and y-axis values for the plot.

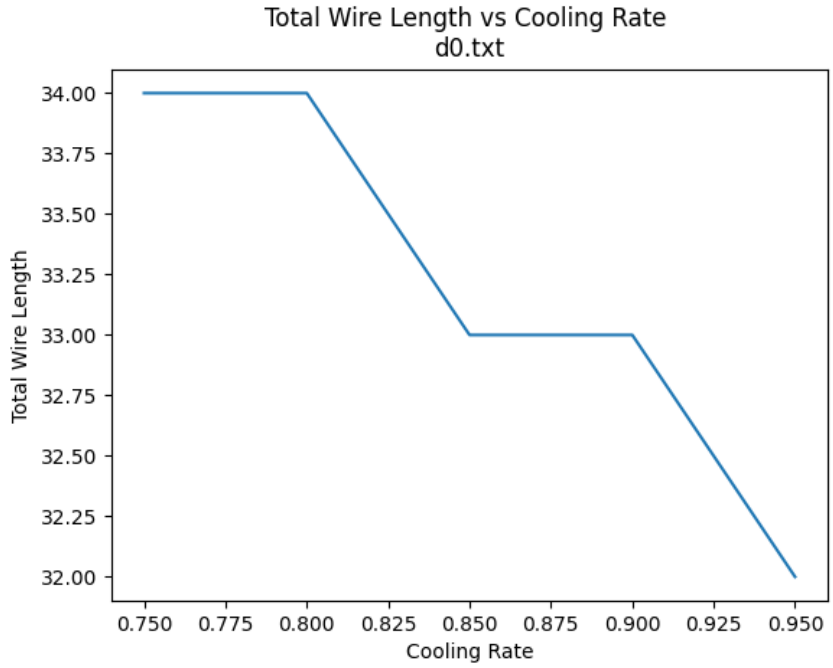
It then begins the simulated annealing optimization process by calculating the initial HPWL and temperature based on the size of the grid and the number of connections. It then uses a loop to keep swapping the positions of two randomly selected cells in the placement array until the temperature reaches a certain threshold. During each iteration of the loop, the function calculates the HPWL of the new placement array and compares it with the previous HPWL. If the new HPWL is lower than the previous one, the new placement array is accepted. If it is higher, the function calculates the probability of accepting the change based on the new and previous HPWL and a random number. If the change is accepted, the new placement array is kept; otherwise, the previous placement array is kept.

The function also updates the temperature and stores the x and y-axis values for the plot. Once the optimization process is complete, the function plots the graph of total wire length vs. cooling rate and total wire length vs. temperature using the stored values and saves the plots to files.

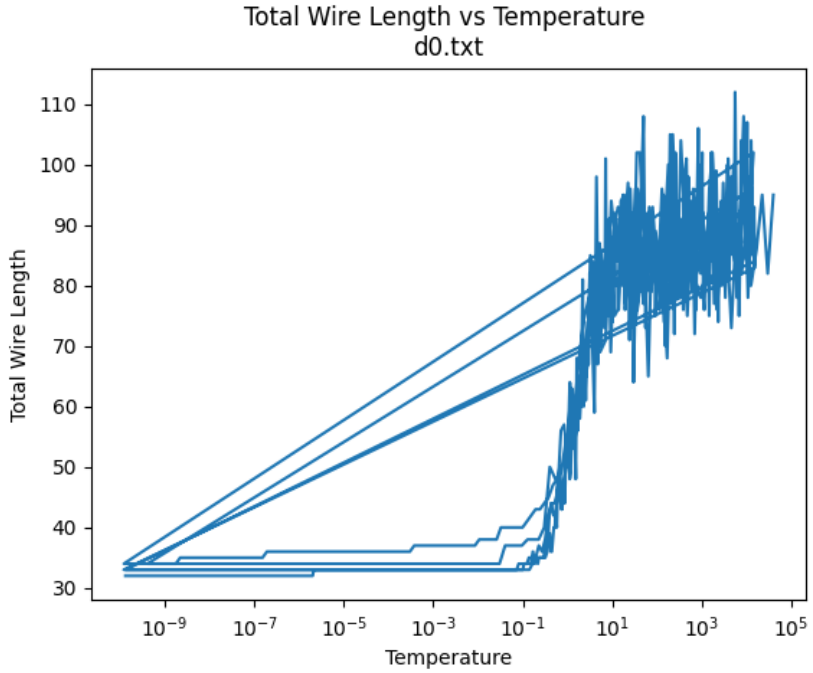
4. Graphs:

D0:

Cooling vs TWL:

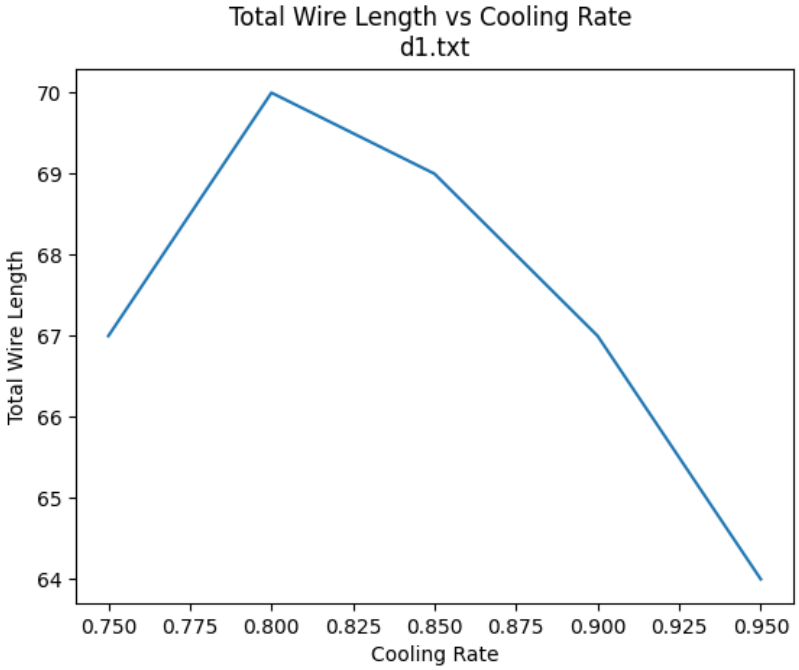


Temperature vs TWL:

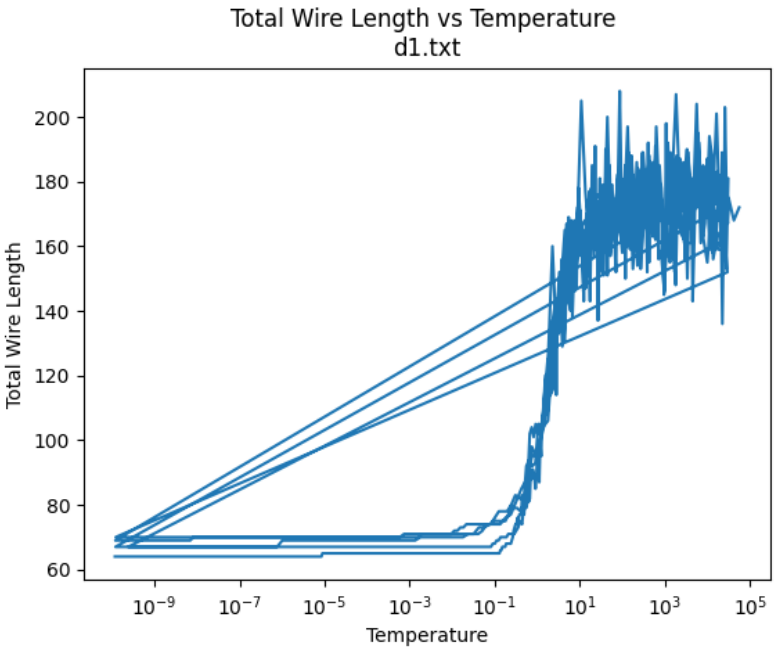


D1:

Cooling Rate vs TWL:

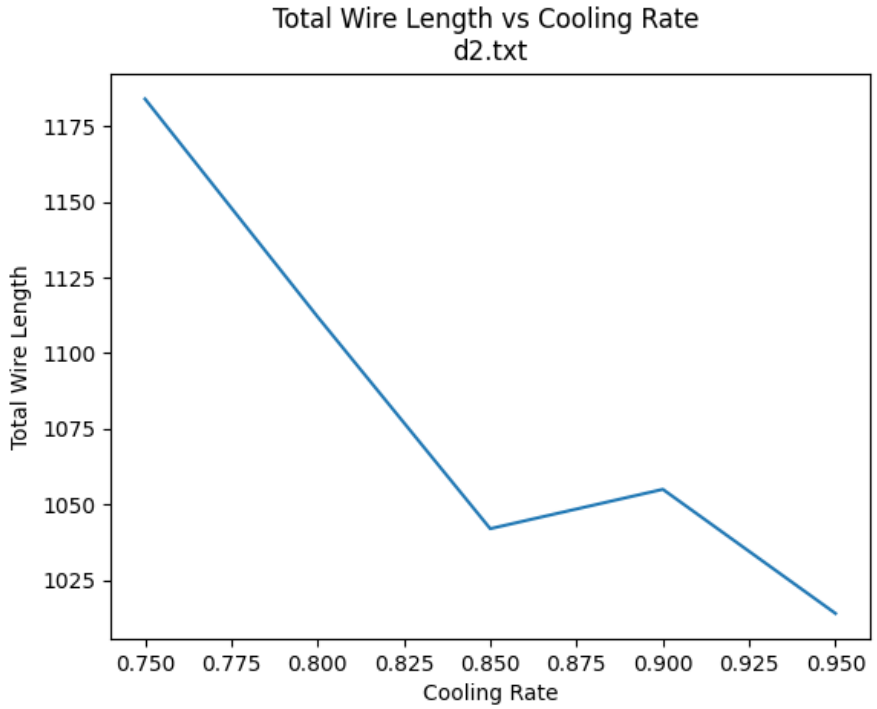


Temperature vs TWL:

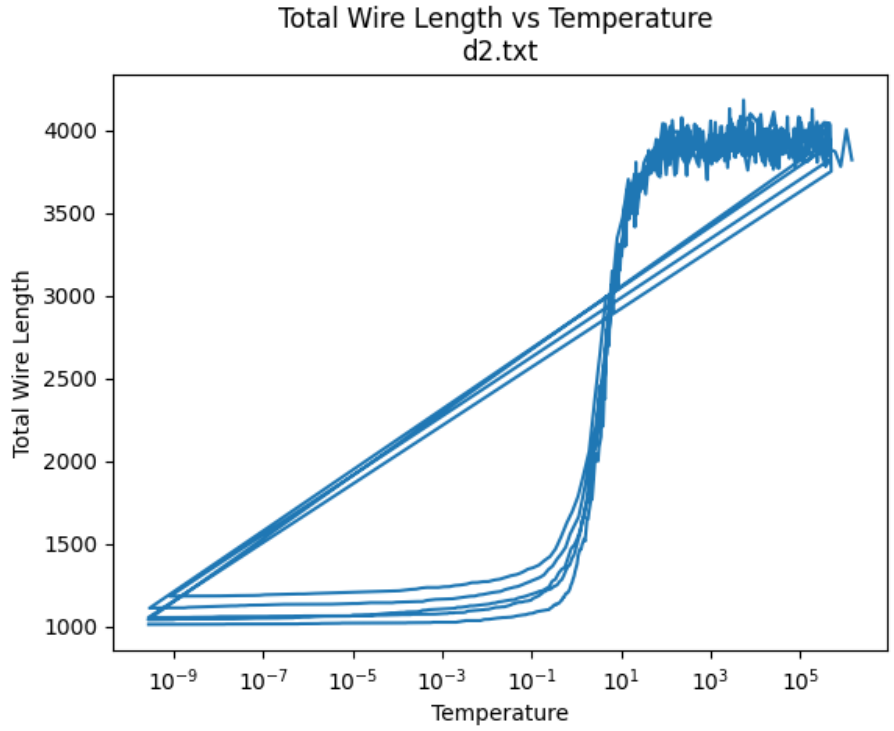


D2:

Cooling Rate vs TWL:

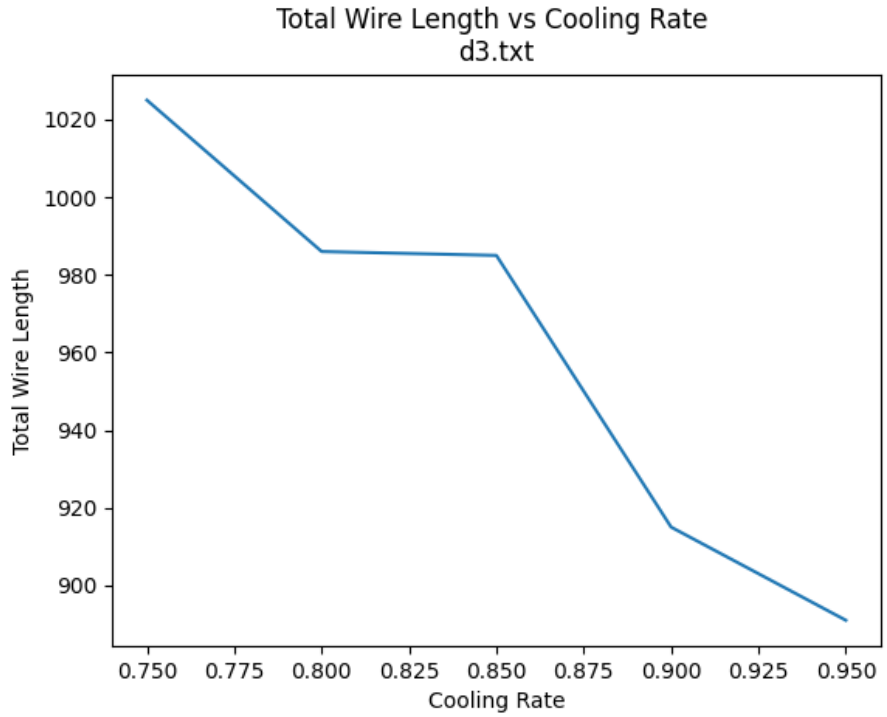


Temperature vs TWL:

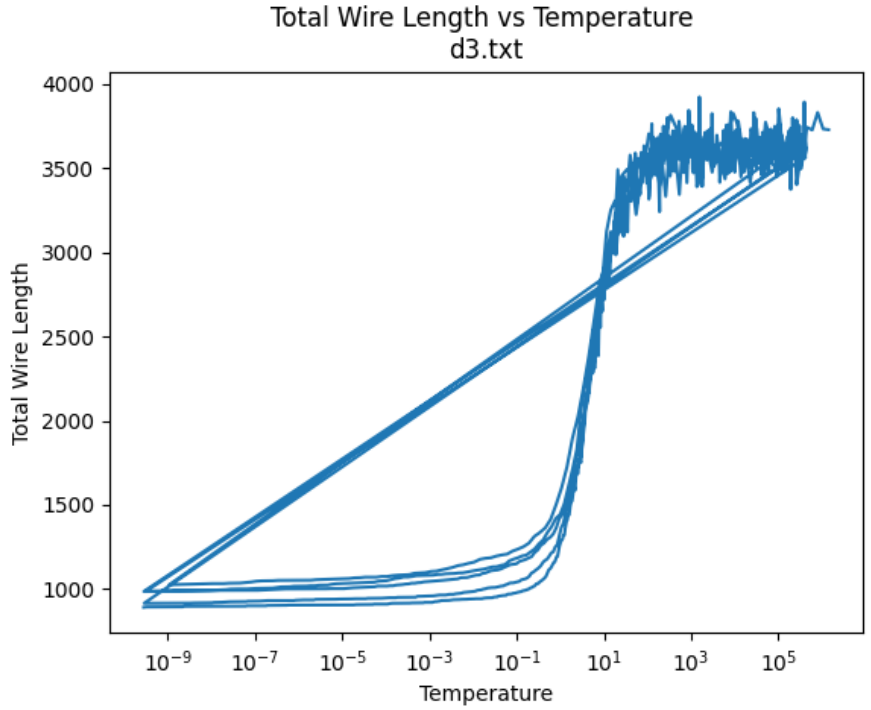


D3:

Cooling Rate vs TWL:

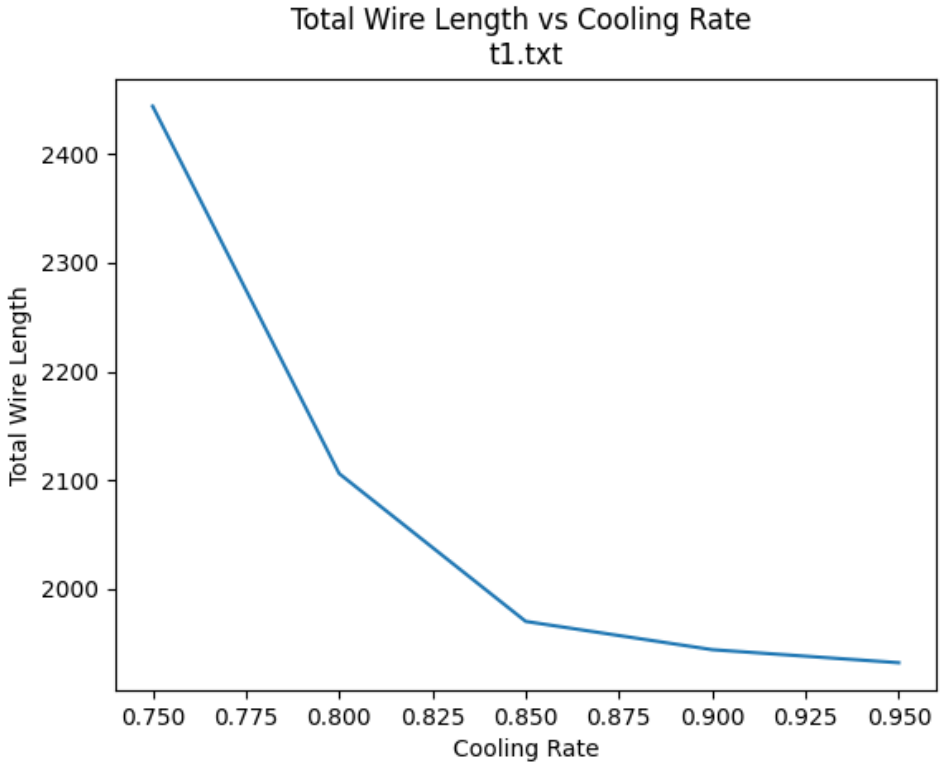


Temperature vs TWL:

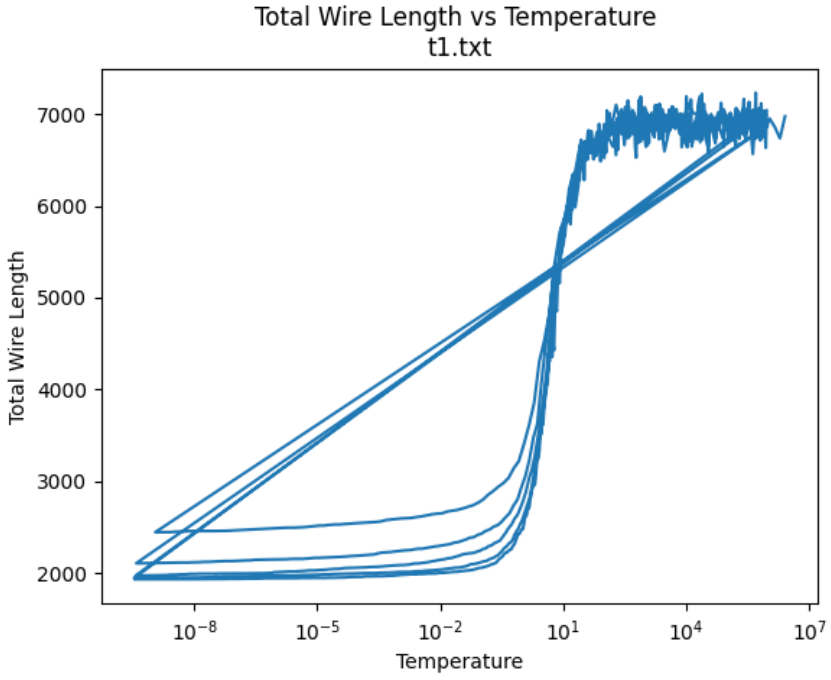


T1:

Cooling Rate vs TWL:



Temperature vs TWL:



T2:

Cooling Rate vs TWL:

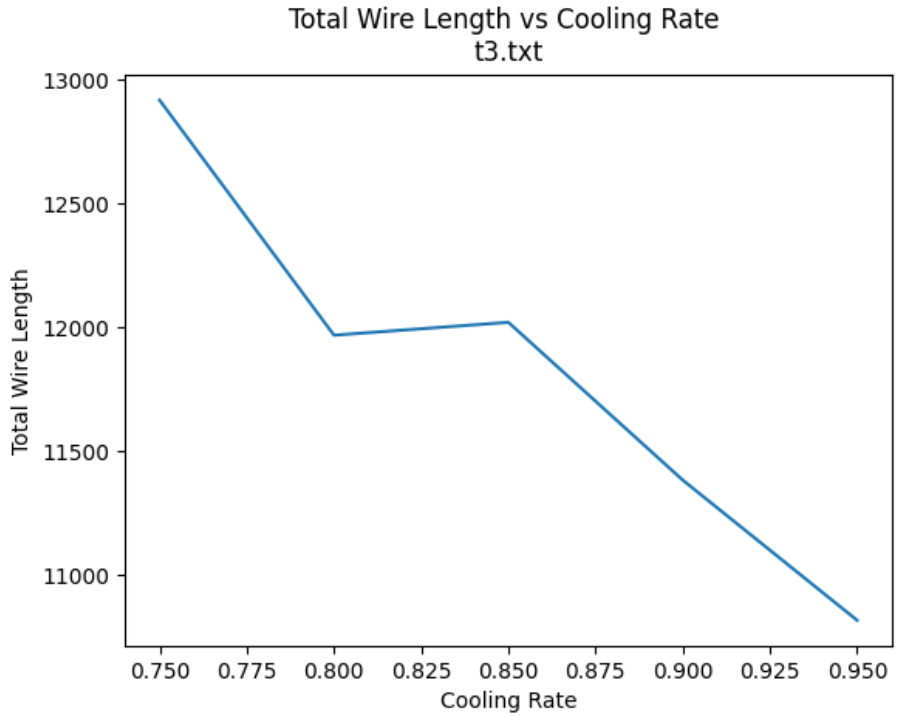
Graph is missing as T2 kept crashing before completion

Temperature vs TWL:

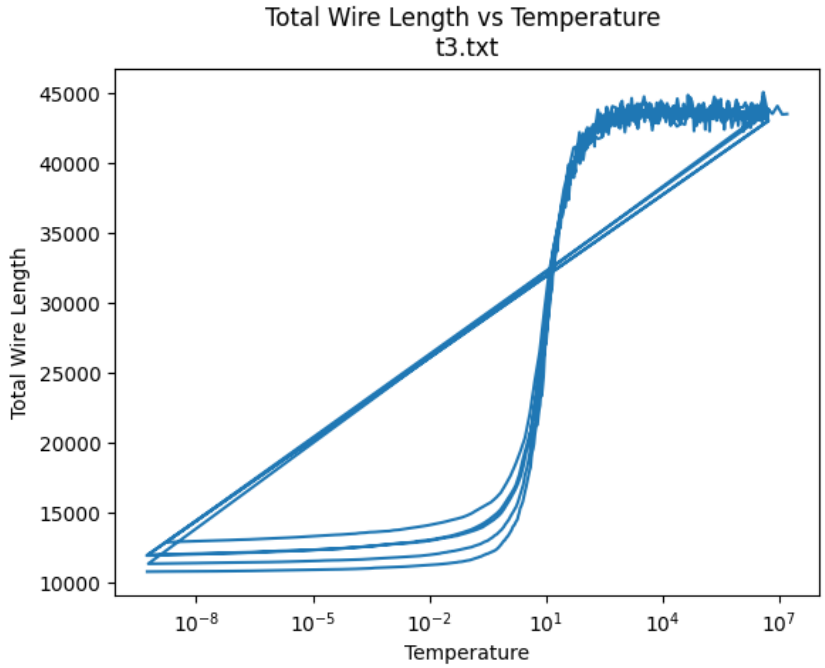
Graph is missing as T2 kept crashing before completion

T3:

Cooling Rate vs TWL:



Temperature vs TWL:



5. Conclusions:

1. Temperature's Conclusion:-

From the graphs, as illustrated above, we can observe that as the temperature is high the wire length increases, this means that in terms of mathematics and statistics they are both directly proportional. However, in our graphs we plot the temperature on the x-axis from low to high, and this does not happen in real life as the annealing algorithm starts with the high temperature and then cools down using the cooling schedule which is given and used to calculate the new temperature [Refer to the issues faced section for the plotting of temperature problem]. Moreover, the obvious conclusion is that as the temperature decreases the wire length also decreases.

2. Cooling Rate Conclusion

From the graphs shown above, we can conclude that the wire length and cooling rate are inversely proportional so as the wire cooling rate increases the length increases. This happens because I start swapping more as it will take more time for the temperature to decrease hence giving more time resulting in more iterations for the swapping of the cells.

6. Issues Faced:

We faced some issues in our project, mainly 2. The t2.txt test case kept crashing whenever we ran it, even though it was smaller than t3.txt, which ran normally. The Second was the Temperature graphs. There was an issue when we plotted the temperature graphs, they came out reversed. The graph should have started from a high and go to low values. This was due to an issue in our list generation for the values to be plotted. The graphs do not contain wrong values, they were just supposed to be plotted in a different way.

7. Bonus:

We achieved the GIF animation bonus. We did it using the Imageio library in python. We took screenshots of our GUI for every update of it. This was done through tkcap. After taking the screenshots, using Imageio we merged them together into a GIF format. The GIF will be available to see during the demo of the project.