Project Report

Department of Computer Science and Engineering

School of Sciences and Engineering

The American University in Cairo

CSCE 3304: Digital Design II

Professor Mohamed Shalan

May 20, 2024

Project Report

Algorithm

Hill climbing is about maximizing the objective function. The move that will add an improvement is taken. It is a greedy algorithm because it searches for immediate gain. All moves are being driven by the fact that it is greedy. However, greedy algorithms give non-optimal solutions. The greedy algorithm will always take us down the hill until we reach the local or global minimum. Then, the greedy algorithm stops because any move after this will increase the Total Wire Length (TWL). There are both local minima and the global minimum. If the greedy algorithm is lucky, it reaches the global minimum. Starting anywhere will probably lead to a local minimum.

Simulated annealing is less greedy and accepts bad moves. However, it does not accept any bad moves. Simulated annealing is a very well-known optimization algorithm. Alloys are mixtures of metals that are made using annealing. Metals are first converted from solid to liquid. Every metal has a melting point which is associated with the bonds between the atoms. More energy is required to break these bonds. Then, atoms start to vibrate when heat is injected. These oscillations will be larger as more heat is injected. Next, the metals are mixed. There must be a cooling profile for cooling the metals down. These cooling profiles do not involve rapid cooling, so the cooling is very slow. The cooling profile is associated with how the temperate changes with time. Every metal alloy has its own profile, and this depends on the metals being mixed to create the alloy. The idea is to give atoms a chance to figure out their optimal positions. A system tends to be in the lowest energy state. Entropy measures randomness, and natural systems tend to be random. The atoms start to organize themselves to create lattices and crystals. For the problem of cell placement, simulated annealing is performed. We start with random placement. Then, we

pick two cells and swap them. Next, we measure the change in the wire length. The decision to discard the swap does not depend only on the fact that the total wire length increases or decreases. It also depends on some probability or randomness. This probability is associated with time. Initially, the probability is high, and it starts to go down with time. This probability is used for accepting bad moves. Initially, all the good moves and many bad moves are accepted. The acceptance of bad moves decays with time because the probability goes down with time. Starting with a random placement and accepting only good moves will lead to a local minimum. Accepting bad moves it away from the local minimum to some random place not far away to reach an even better local minimum. Initially, bad moves are done with high probability. Then, they are taken with a probability of almost zero. The temperature is the time for running the simulated annealing algorithm. Timber-Wolf is a very well-known algorithm that starts with a temperature of 4 million degrees. It keeps running until it reaches 0.1 degrees. The initial, final temperatures, and cooling profiles influence how long the simulated annealing algorithm will run. In classical simulated annealing, rapid cooling is done at the beginning and the cooling starts to slow down later. The probability is associated with the cooling.

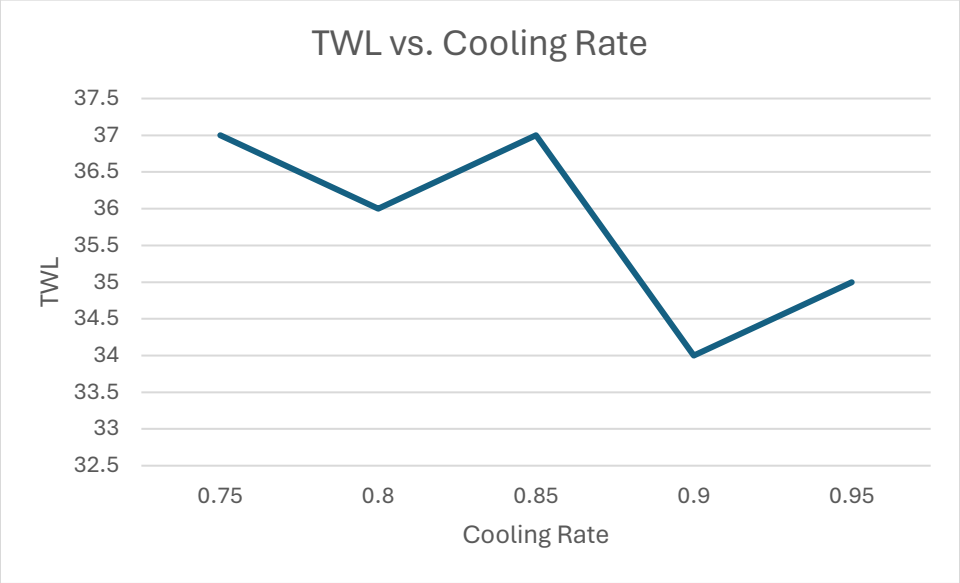
Implementation

The simulated annealing placer was developed using C++. C++ was used to implement the algorithm described above. C++ was chosen because of its efficiency and optimization. The data structures that were used in the implementation were vectors, vectors of vectors, and unordered set of unordered maps. *numbers* is a vector of vectors that is used to store the data obtained from the parse function. *cells_nets* is an unordered map of unordered sets which contains the nets mapped to each cell. *connections* is an unordered map of unordered sets which contains maps the cell to the different nets connected to it. This data structure was used for optimizing the simulated annealing algorithm.

parse is a function which was used for reading the text files representing the netlists. This function takes the file path as a string and a vector of vectors to store the parsed data and returns a boolean flag indicating whether the text file was read without issues. *initialize* is another function which takes the numbers, *xcoords*, and *ycoords* and returns a vector of vectors called *grid*. The vector of vectors returned by the function is the initial random placement of the cells on the grid. *print* is a void function which displays the grid in normal format. *print_bin* is another void function which displays the grid in binary format. *calculate_hpwl* is a function that takes the *xcoords*, *ycoords*, *numbers*, and *net* to extract the minimum and maximum x and y coordinates and calculate the HPWL by getting the difference between the maximum and minimum x coordinates and adding the value obtained to the difference between the maximum and minimum y coordinates. *set_union* is a function which does the union operation on sets when both cells that are to be swapped are not empty.

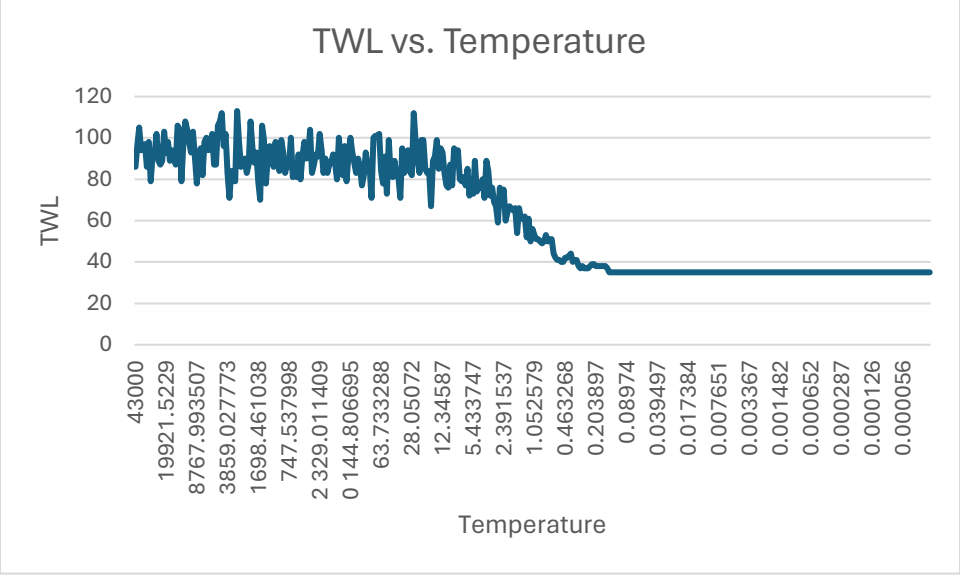
Graphs

The TWL vs. cooling rate graph for d0 is shown in the figure below:

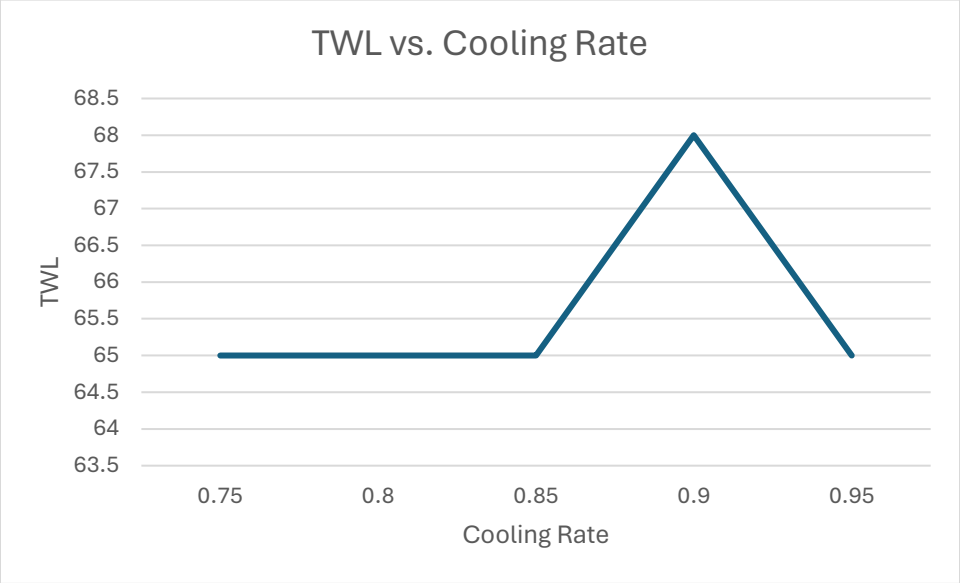


The TWL for d0 peaked at a cooling rate of 0.75 and 0.85 and was lowest at a cooling rate of 0.9.

The TWL vs. temperature graph for d0 is shown in the figure below:

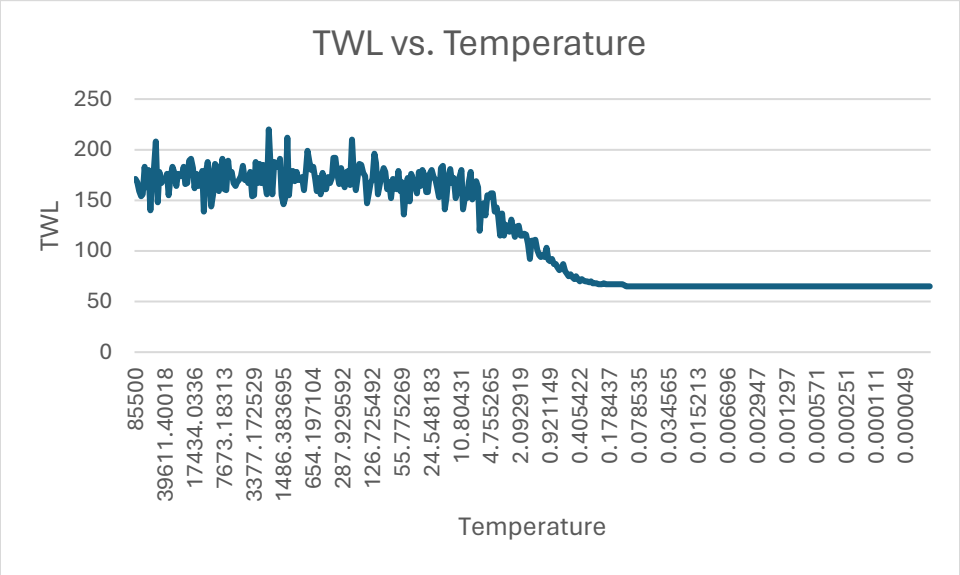


The TWL vs. cooling rate graph for d1 is shown in the figure below:

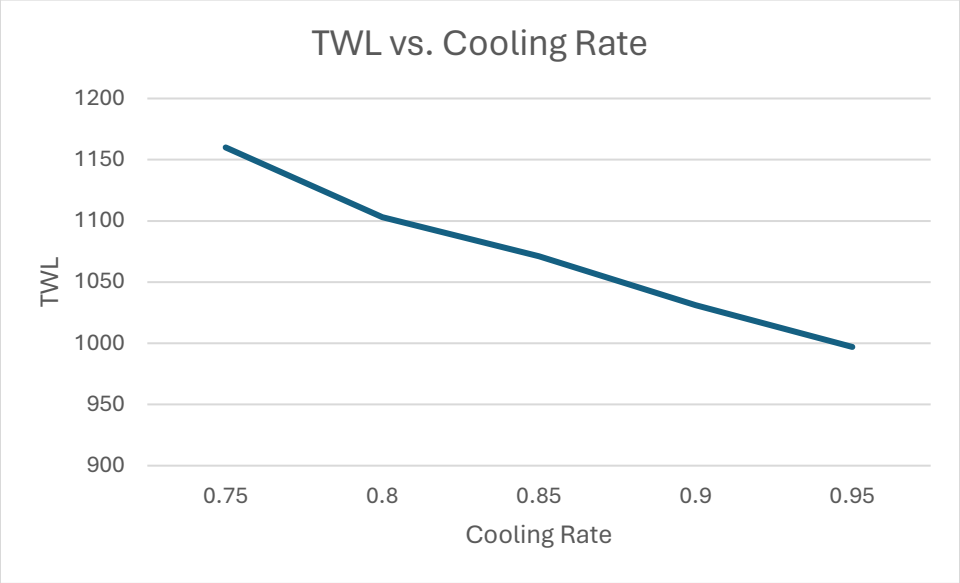


The TWL for d1 reached a maximum at a cooling rate of 0.9 and had the lowest values at the cooling rates of 0.75, 0.8, 0.85, and 0.95.

The TWL vs. temperature graph for d1 is shown in the figure below:

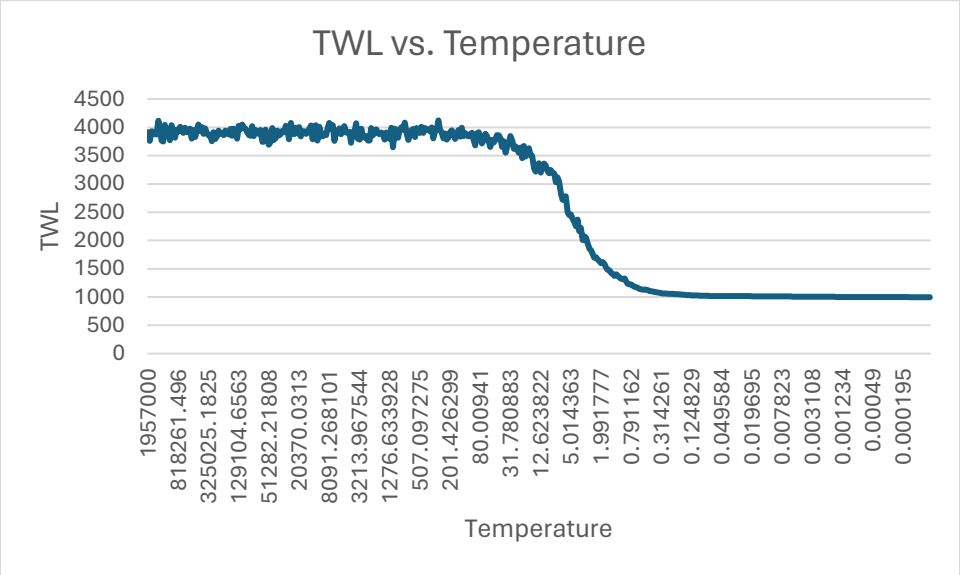


The TWL vs. cooling rate graph for d2 is shown in the figure below:

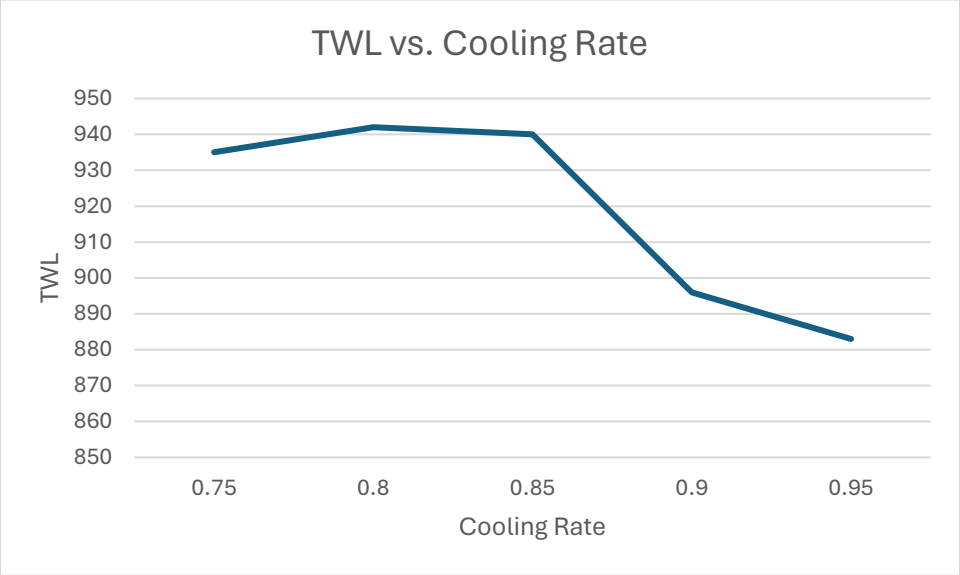


The TWL for d2 reached a minimum at a cooling rate of 0.95 and a maximum at a cooling rate of 0.75.

The TWL vs. temperature graph for d2 is shown in the figure below:

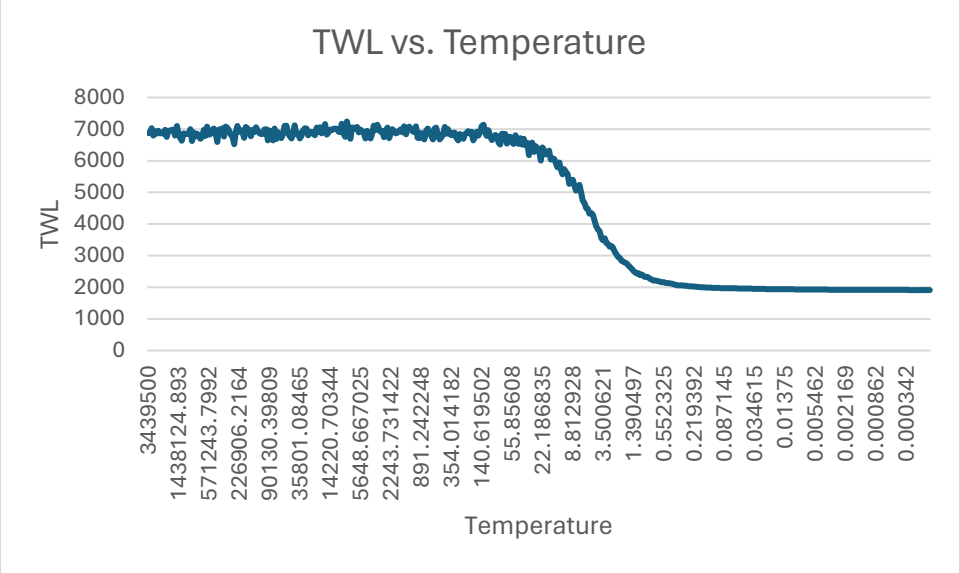


The TWL vs. cooling rate graph for d3 is shown in the figure below:

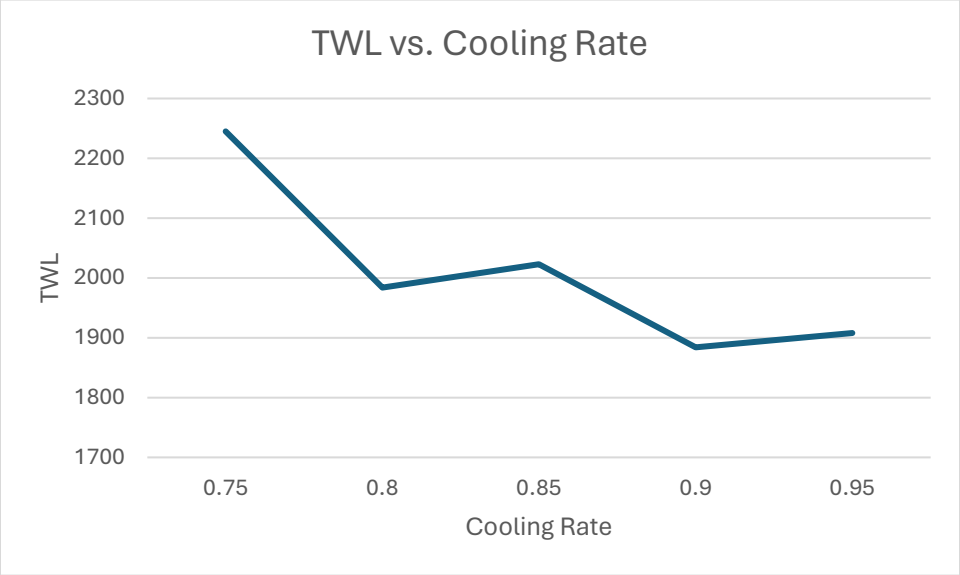


The TWL reached a maximum at a cooling rate of 0.8 and a minimum at a cooling rate of 0.95.

The TWL vs. temperature graph for d3 is shown in the figure below:

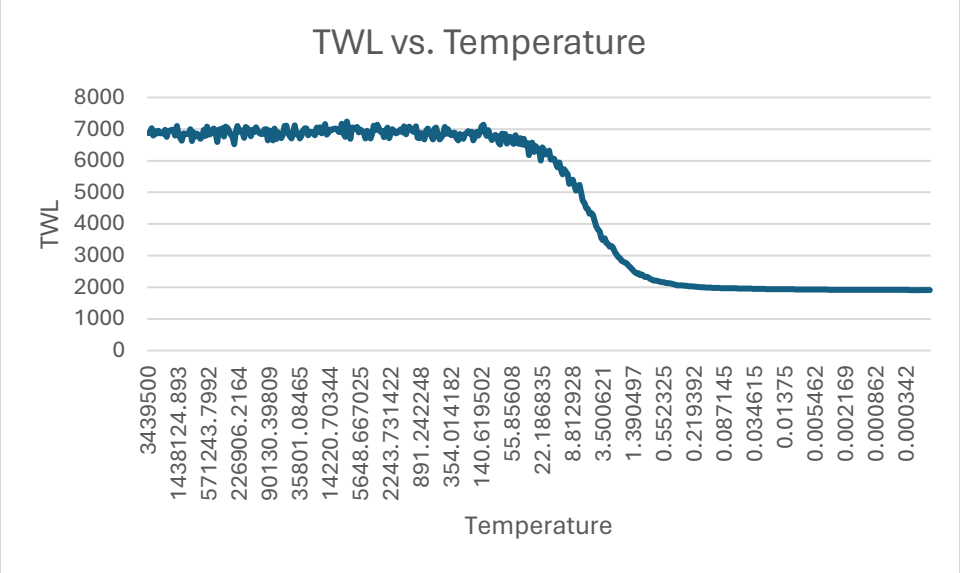


The cooling rate graph for t1 is shown in the figure below:

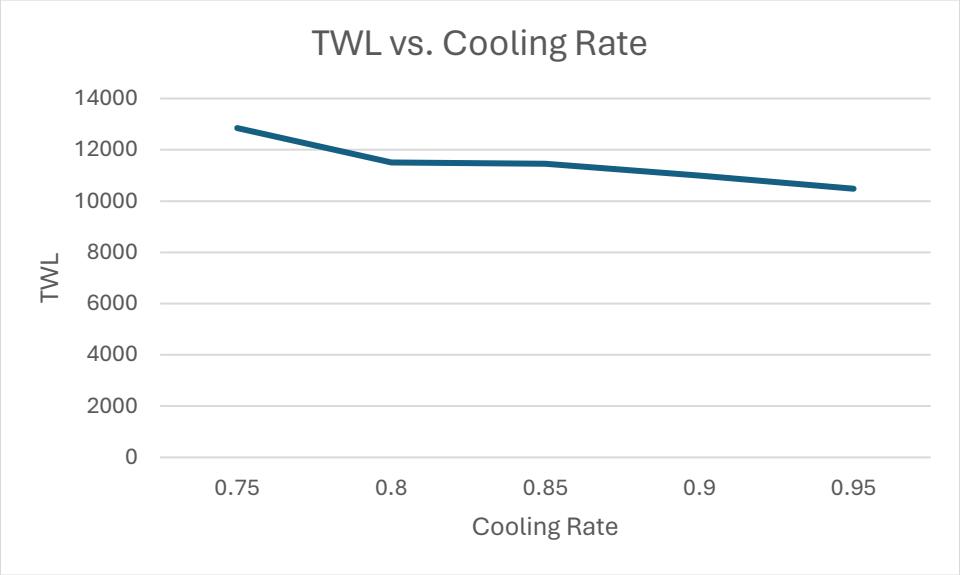


The TWL reached a maximum at a cooling rate of 0.75 and a minimum at a cooling rate of 0.9.

The TWL vs. temperature graph for t1 is shown in the figure below:

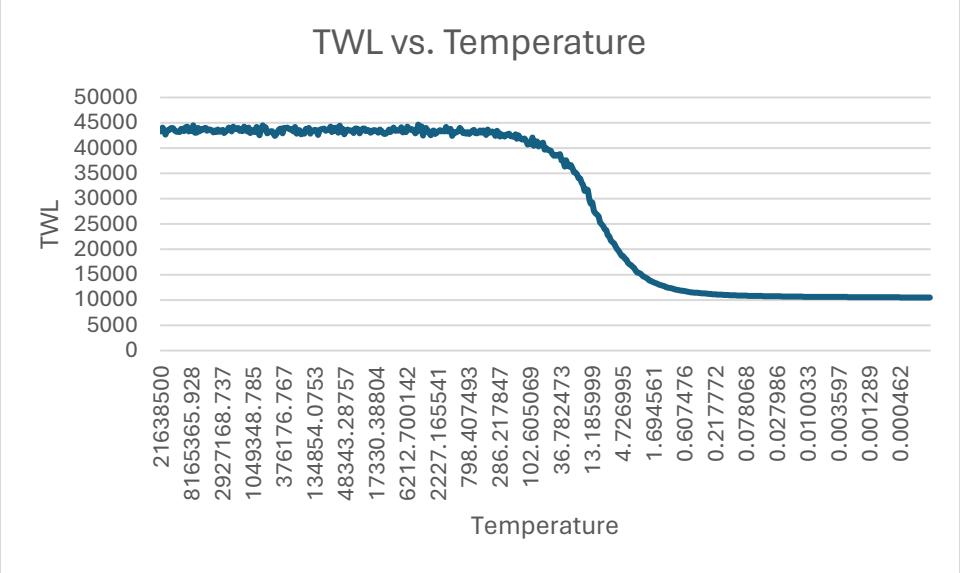


The cooling rate graph for t3 is shown in the figure below:



The TWL reached a maximum at a cooling rate of 0.75 and a minimum at a cooling rate of 0.95.

The TWL vs. temperature graph for t3 is shown in the figure below:



Conclusions

The simulated annealing algorithm has proved to be efficient at getting the optimal cell placements by minimizing the TWL. The stochastic nature of simulated annealing allows it to get out of local minima and makes it the perfect choice for optimization problems with numerous local minima. The capability of simulated annealing to exit local optima is clear by the slow enhancements in TWL seen over several iterations. The cooling rate is essential to the performance of simulated annealing. In general, lower cooling rates result in enhanced optimization results. This is because it enables a more extensive exploration of the solution space. On the other hand, the time complexity increases when the cooling rate is decreased. The program successfully gives precise outputs via its execution. It displays the positions of the different cells both before and after execution of the simulated annealing algorithm. Additionally, it displays the TWL both before and after the execution of the simulated annealing algorithm. This helps with the evaluation of the algorithm. Comprehensive testing was conducted, and the code passed the tests. For the biggest test case, the program finished execution in approximately 26 seconds on an Ubuntu 20.04 Virtual Machine (VM) running on an AMD Ryzen 4800H processor with 8 GB of primary memory.