Abdelrahman El Askary 900211596

Merna El Saaran 900211654

Malak Zeerban 900211667

DD II Project

Simulated Annealing Placer Report

Algorithm:

The idea we are implementing here in this project is to use simulated annealing to find the closest to optimal as possible placement of the components to minimize the total wire length connecting them together. At first, we started by making a random placement of the components while taking into account which components belong to which netlist. After that, we started with an initially really high temperature and started doing swaps for the locations of the components as long as our current temperature was greater than the final temperature. If the change in wire length was negative, which means that the new wire length was smaller than the previous one, we accepted the swap and continued to the next iteration since we were required to perform 20 * number of component swaps at any given temperature. For every swap, we calculated an acceptance probability that is dependent on the change in total wire length and temperature. For every swap, we generated a random number so that for example if our acceptance probability was 0.6 and the random number generated at that time was 0.4, we would accept the swap and update the current wire length with the newly calculated value. We used the cooling schedule given to update the temperature and do the corresponding swaps until our current temperature becomes smaller than our final temperature which we previously calculated using the given data. The final wire length we have at that point is the almost optimal wire length for the given components and their netlists since the approach we followed is a greedy algorithm that tries its best to find an optimal solution but doesn’t guarantee that it is the best solution.

Implementation:

- We used c++ since it would be faster than using python

parser_function ( ):

- We began by implementing a feature where the user inputs their file name and accordingly this file is opened, and we added an error-checking mechanism to ensure that the file was opened correctly so that the data we read is not corrupted or damaged.
- After that we read row by row our input and stored the read data in a vector of vector of type int as the vector data structure is easy to use due to the availability of multiple libraries and functions that facilitate its usage and the dynamic resizing upon adding or removing elements.

calculate_WL ( ):

- This function begins by looping over all of the netlists we have to calculate the wire length for every netlist and hence calculate the total wire length at the end.
- Inside the netlists for-loop we loop over the number of components for every netlist we are examining to identify the maximum and minimum x and y values so that we can use HPWL to calculate the total wire length.
- We created a vector of vector of int to called components_netlists to store the netlists that each component is present in.
- We also created a netlist_lengths vector of int to store the wire length of every netlist so that later on we can calculate the new wire length of the affected netlists only.

calculate_new_WL ( ):

- We used an unordered set called affected_nets to store the nets that have undergone a change in wire length to be able to only calculate the change for such nets and not recalculate the length for the whole nets from the beginning. We used an unordered set because it is easy to use and fast to insert in.
- We then recalculated the wire length after the change in the wire length of the affected nets.

printArrayAsBinary ( ):

- Prints the binary representation of the placement where 0 represents a component and 1 represents an empty cell.

Inside the main ( ) function:

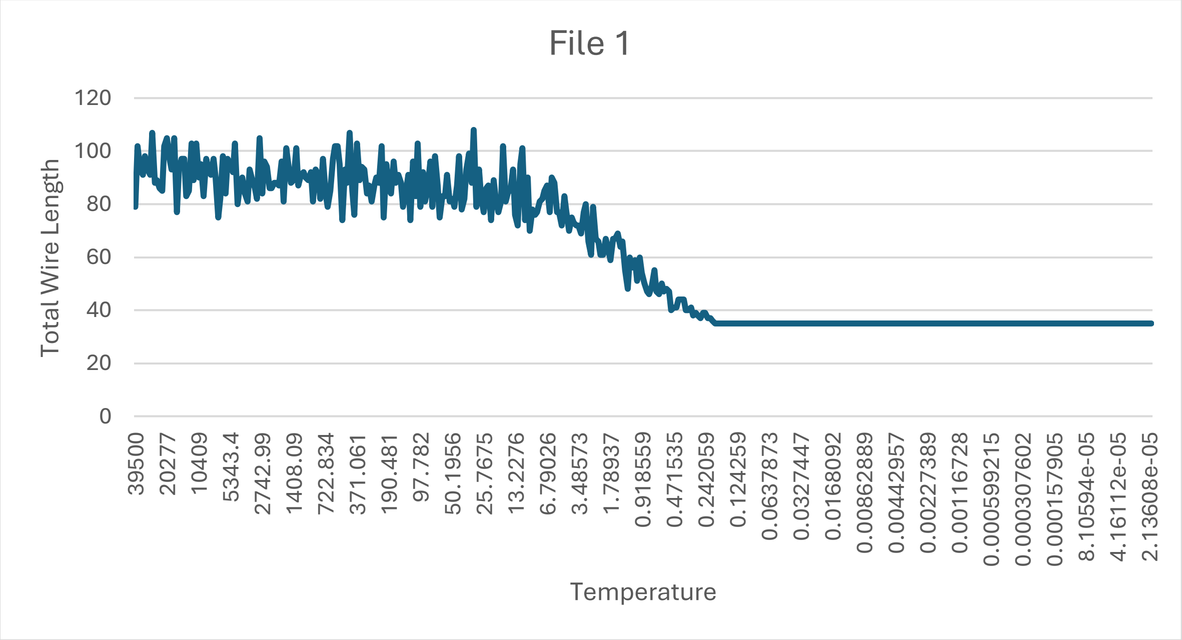
- We initialize important data and create the data structures we will be using.
- We then generate a random placement for every component as long as the components' location in the array we are using to represent our chip is not previously occupied by another component.
- Following that we calculate the wire length for the initial placement and print it as well as print the initial placement of the components after the first random placement both in its normal representation and in its binary representation.
- We then start to initialize some important variables that we will use to implement our cooling schedule.
- After that we started our simulated annealing algorithm by randomly generating a cell ID and then randomly generating a location to swap this cell with what is in the generated

location, and we ensured that the generated location must not be equal to the location of the cell whose ID we generated.

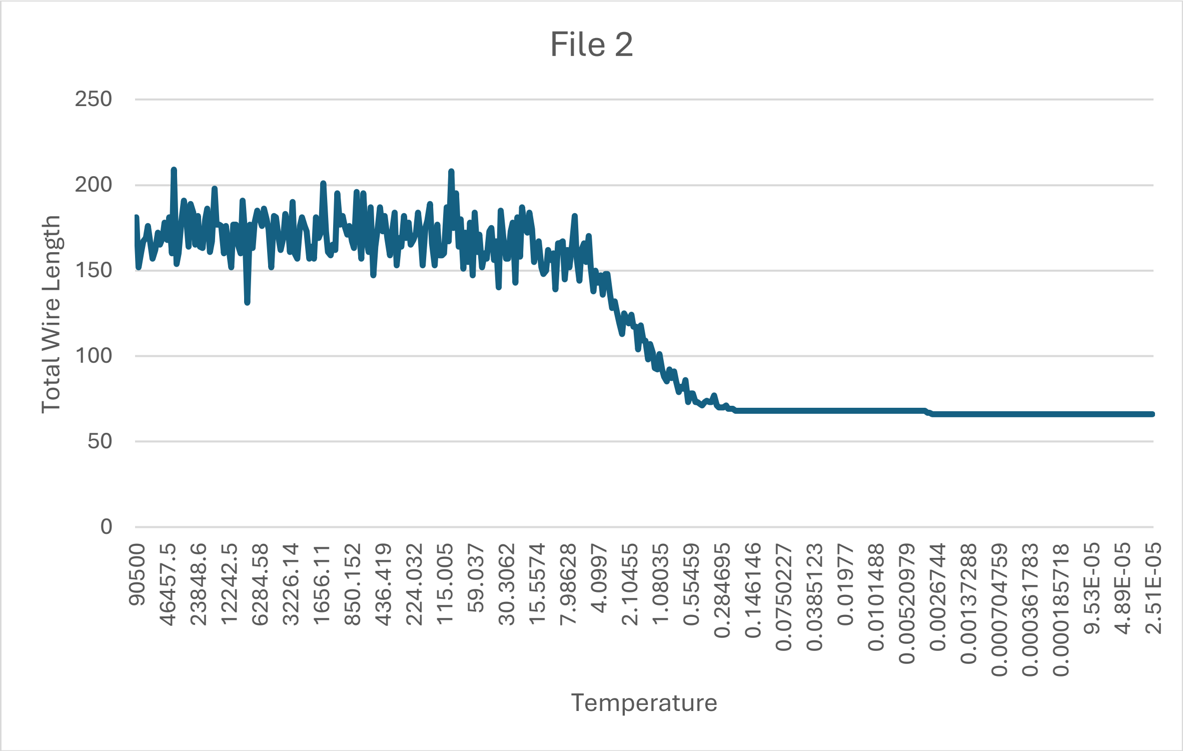
- We added the affected netlists to the affected_netlists set to be used to calculate the new wire length.
- We calculated the new wire length and followed the steps and conditions in the simulated annealing algorithm.
- One modification we made is that we calculated the random number and the probability as a decimal number between 0 to 1 instead of a percentage from 0 to 100 to reduce the time the code needs to run.
- If the swap is accepted the temporary locations are added to the components_locations vector as a way of approving the swap.
- If we reject the swap we revert the swap to its original placement.
- Finally, we printed the solution both in the normal representation and binary representation after the loop finished iterating and the final temperature was reached. We also printed the new wire length and the time taken to run the code.

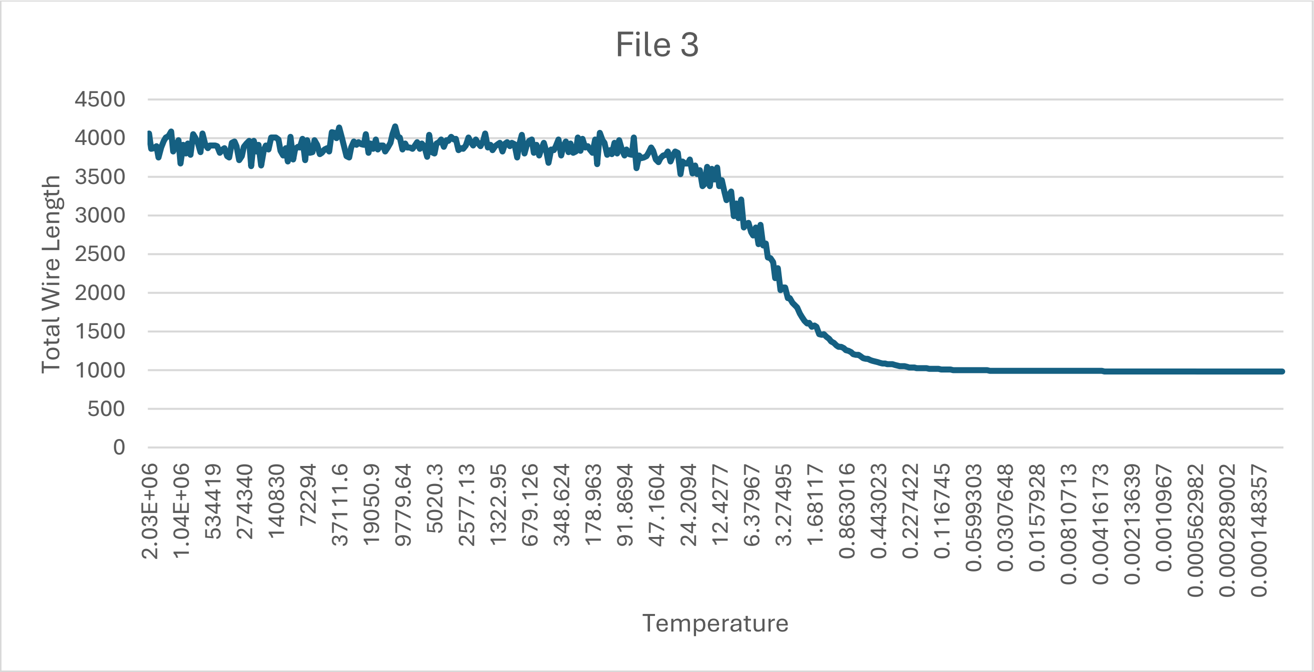
Graphs:

Total Wire Length vs Temperature for the 6 designs:

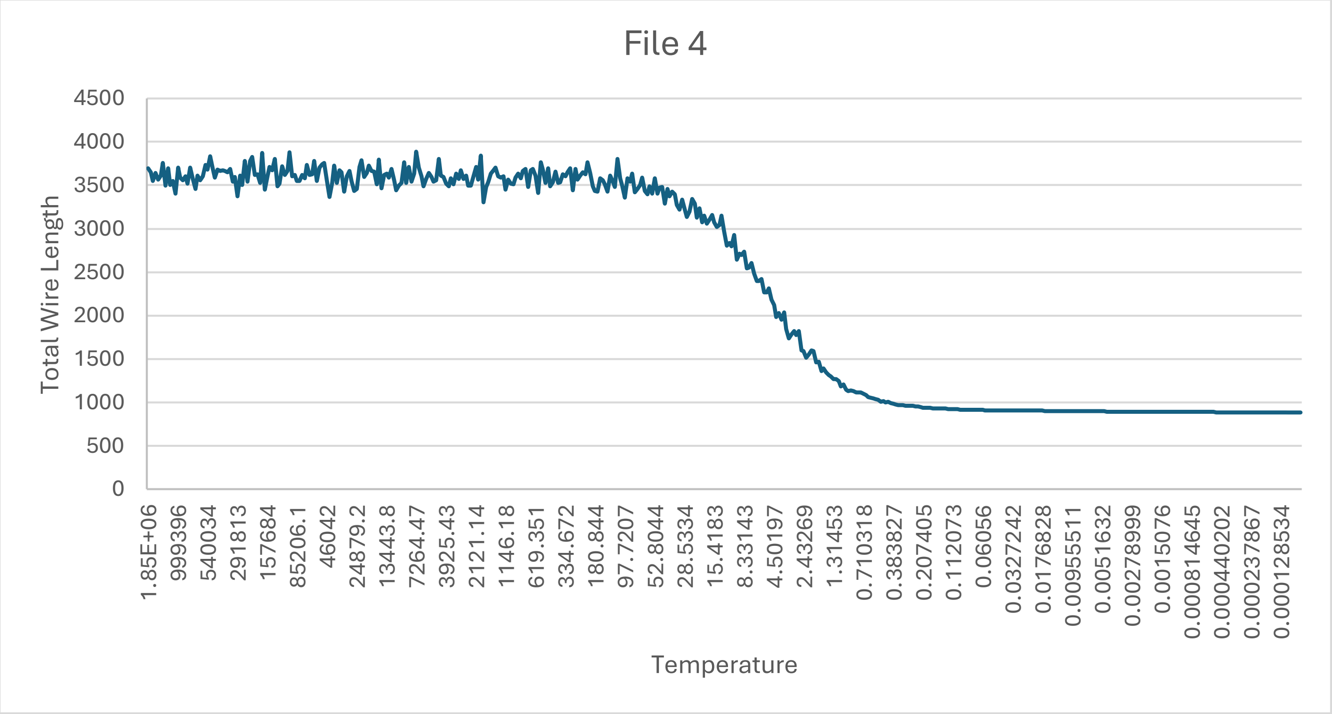


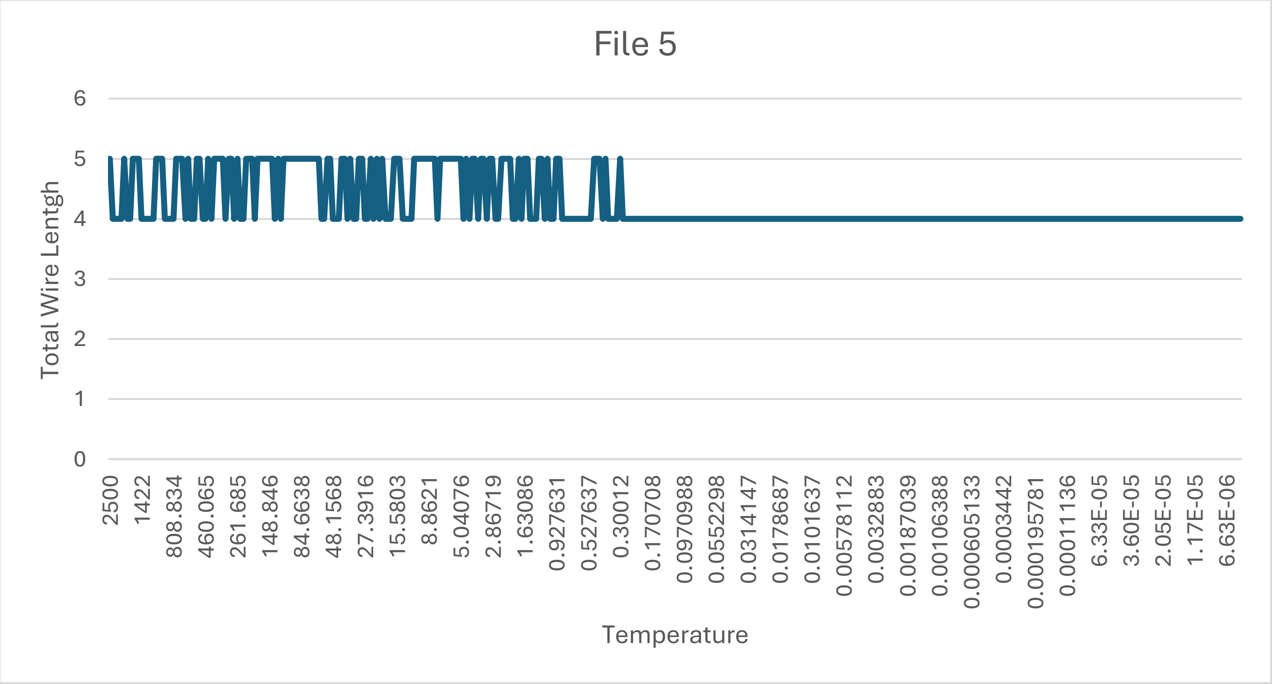
Digital Design 2



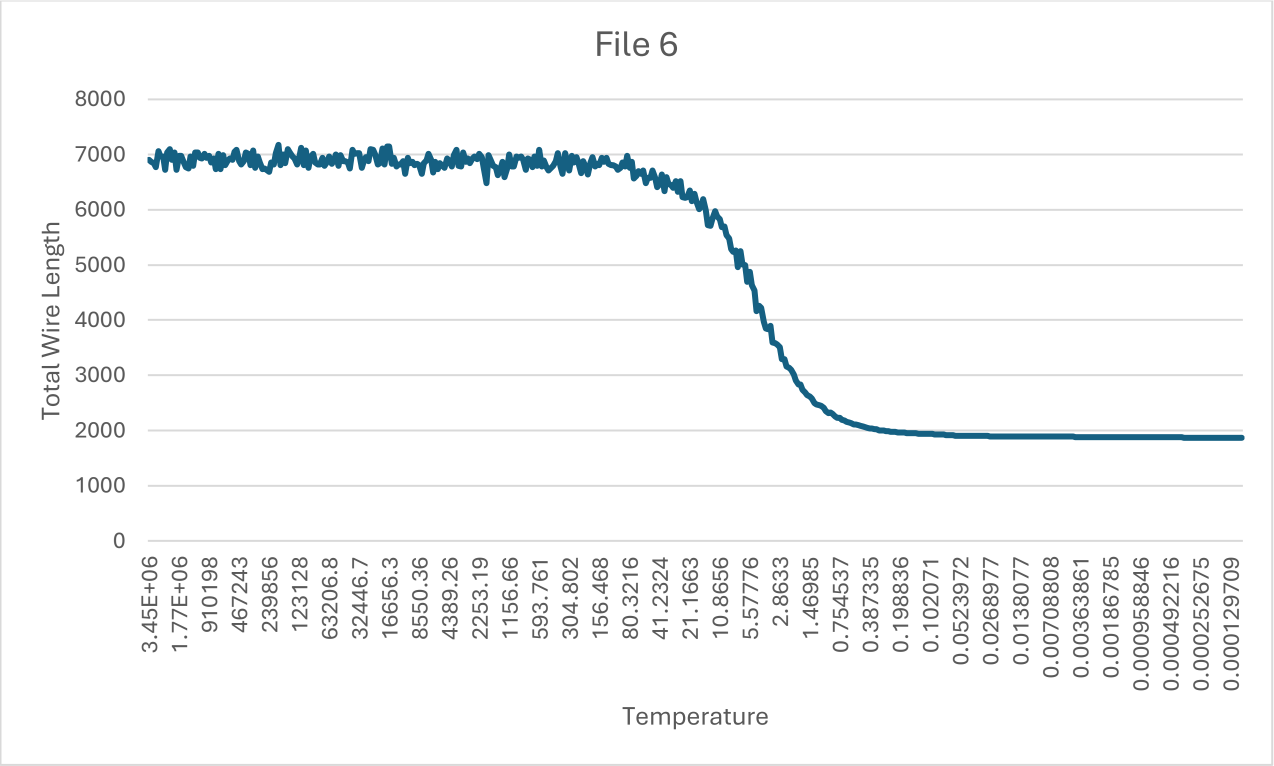


Digital Design 2



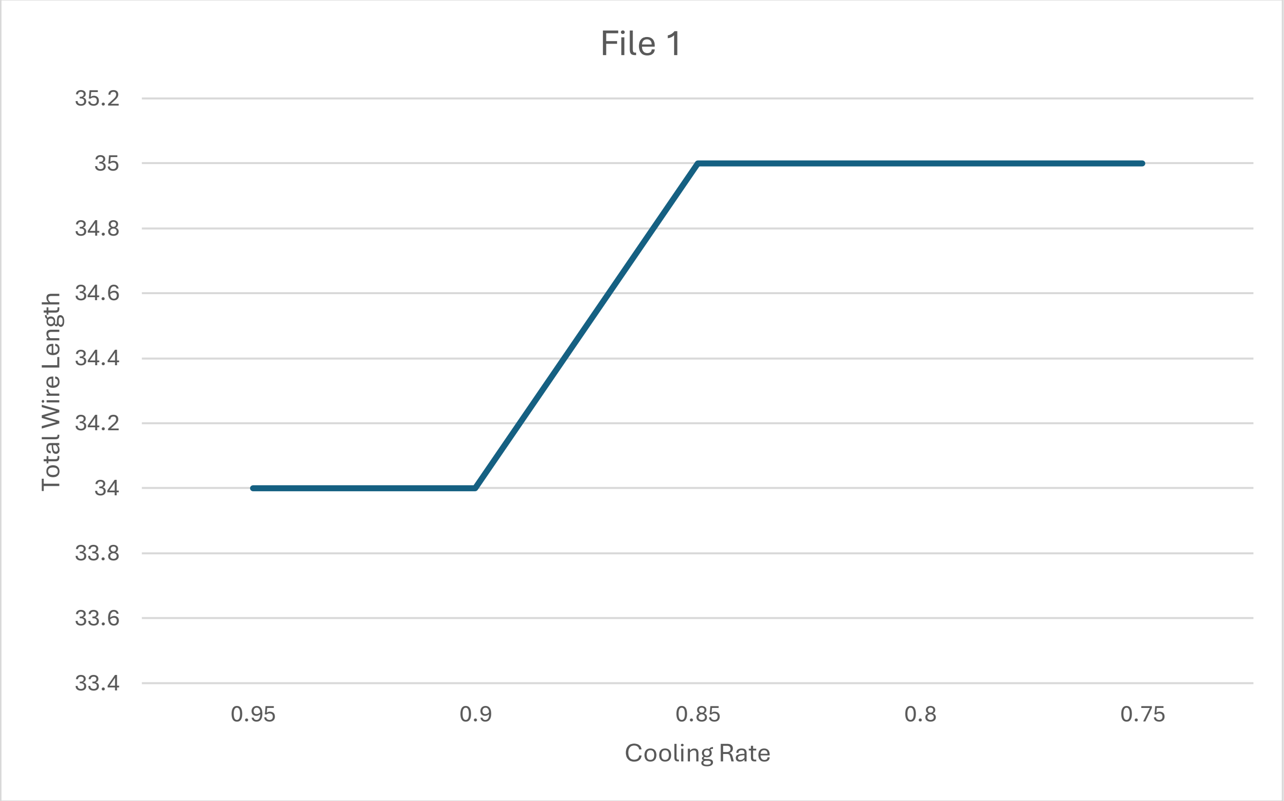


Digital Design 2

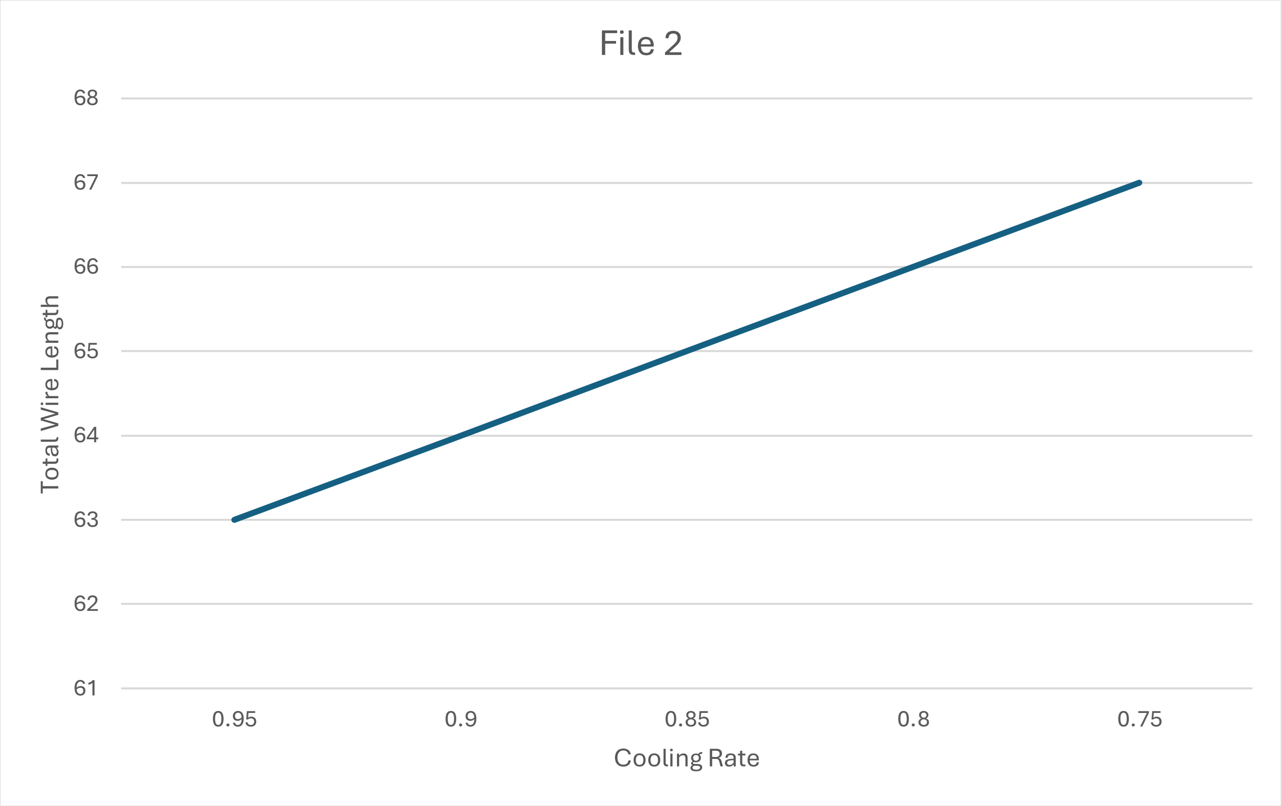


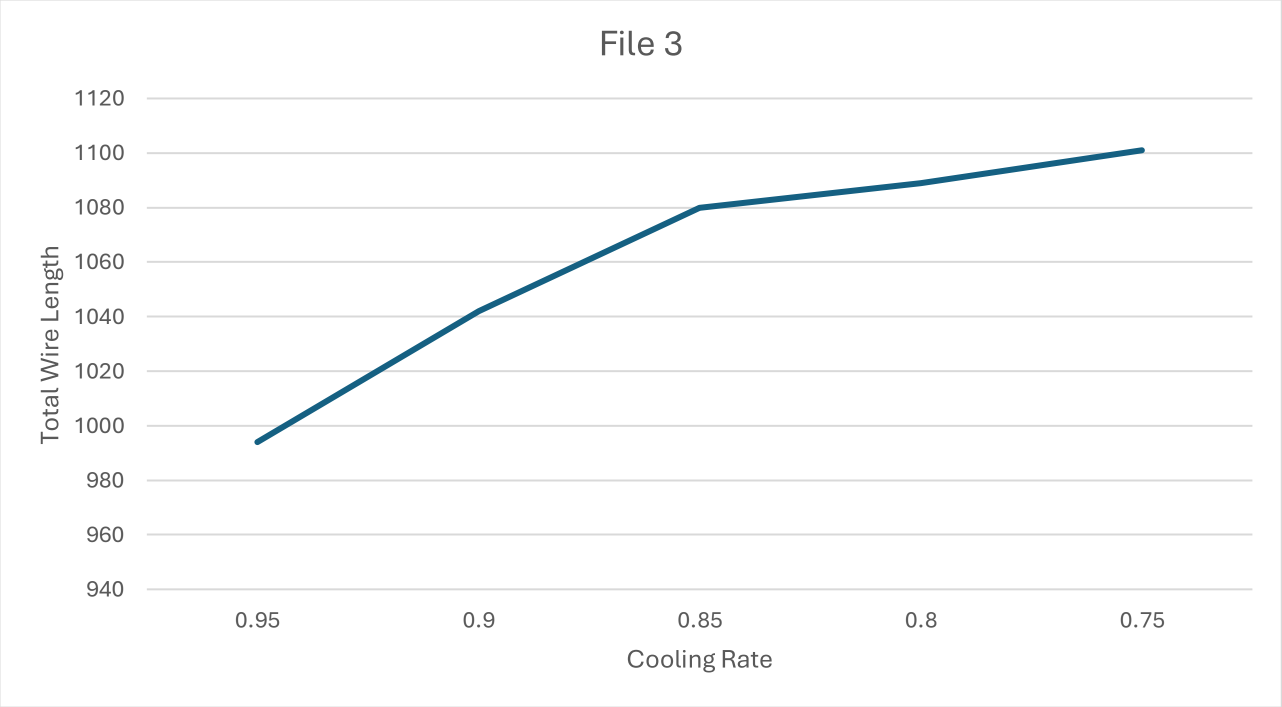
- As temperature decreases, the total wire length tends to decrease. This is expected because lower temperatures in simulated annealing algorithms typically result in fewer accepted swaps that increase wire length, leading to a more optimized placement.
- The graphs show a general downward trend, indicating that the algorithm is effectively minimizing the wire length as the temperature cools.

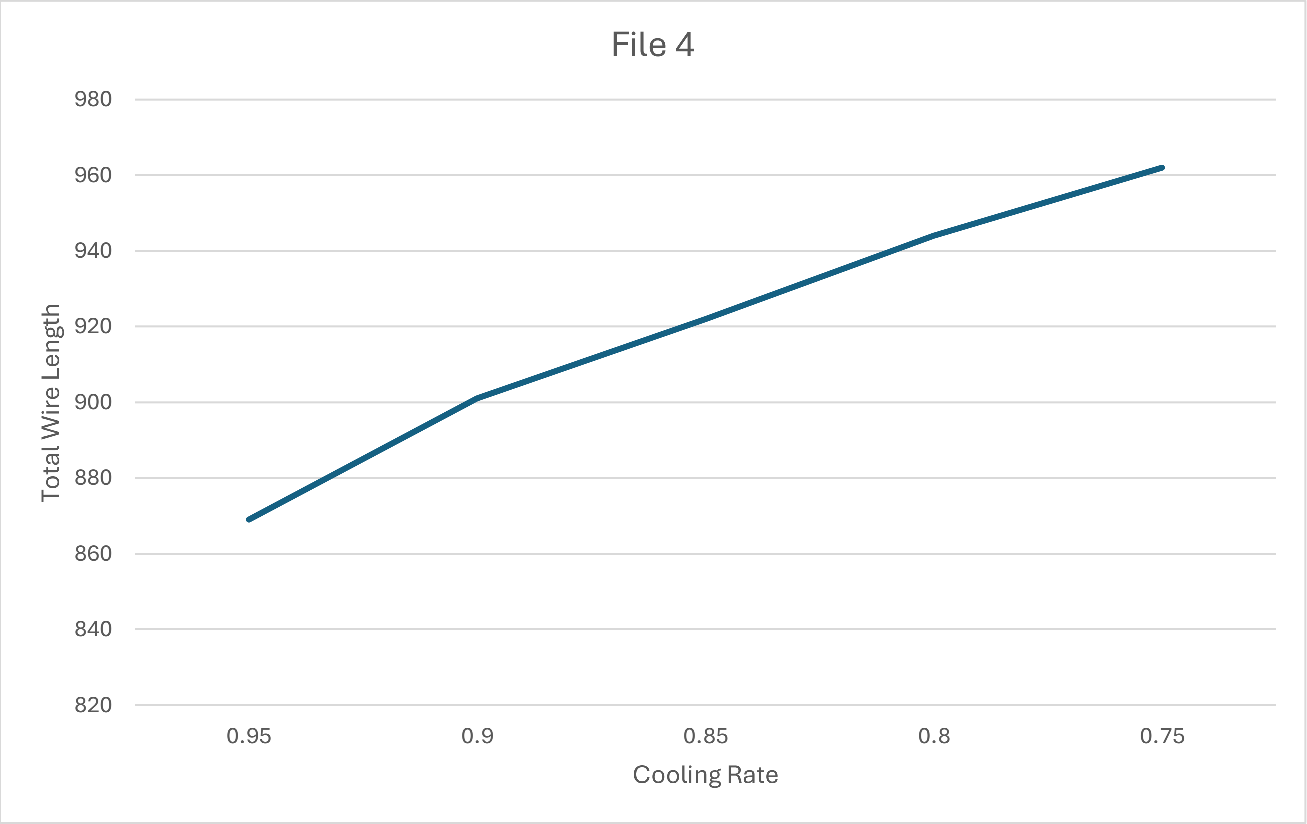
Total Wire Length vs Cooling Rate for the 6 designs:



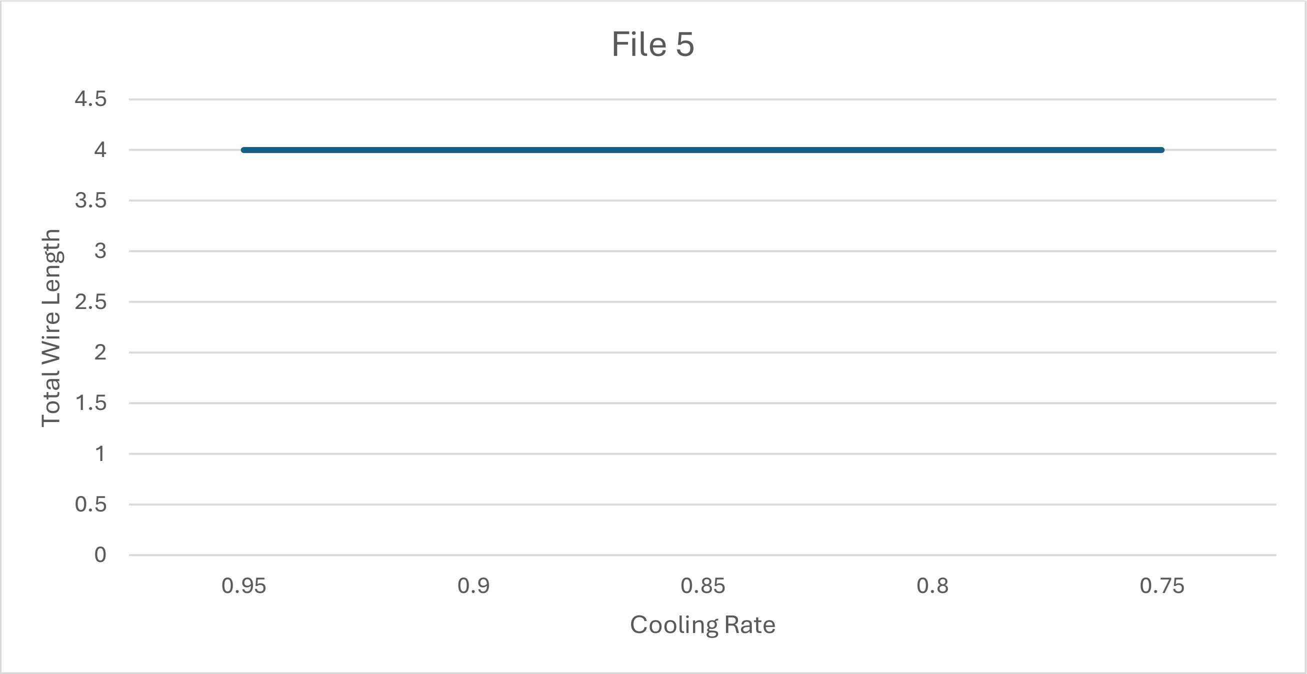
Digital Design 2

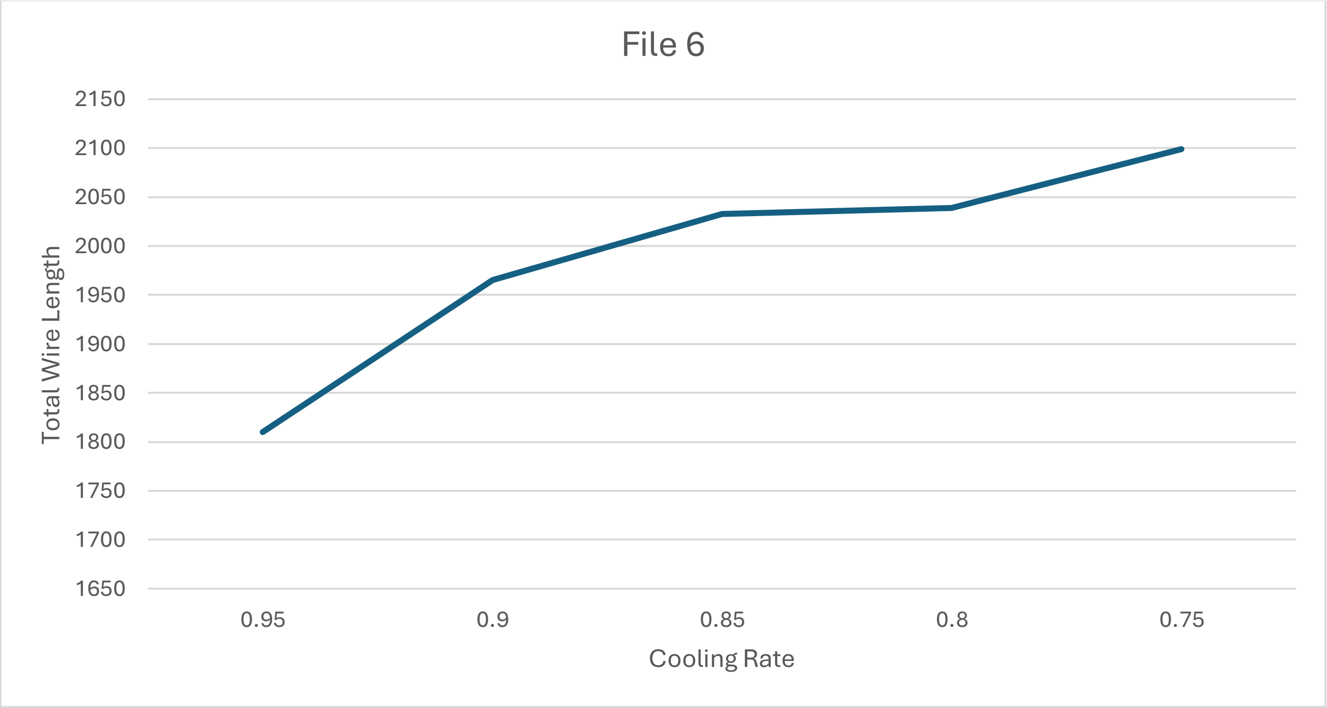





File 4
980
960
940
920
900
880
860
840
820
Total Wire Length
0.95
0.9
0.85
0.8
0.75
Cooling Rate

Digital Design 2





- Different cooling rates result in varying final wire lengths. A slower cooling rate (more gradual decrease in temperature) generally allows the algorithm to explore more configurations, potentially leading to a better (lower) wire length.
- The graphs suggest that there is an optimal cooling rate for each design, where the wire length is minimized. Too fast a cooling rate may not allow sufficient exploration of the solution space, while too slow may result in unnecessary computation without significant improvement.

Conclusion:

This project successfully implemented a simulated annealing algorithm to optimize component placement on a chip, minimizing total wire length. The results showed a significant reduction in wire length as temperature decreased, demonstrating the algorithm's effectiveness. We found that slower cooling rates generally led to better optimization, highlighting the importance of parameter tuning. Overall, this project confirmed simulated annealing as a valuable method for optimizing VLSI design. Future work could explore more advanced cooling schedules and hybrid optimization techniques.