The American University in Cairo
School of Sciences and Engineering

Project: Simulated Annealing Cell Placer

By:

Basant Abdelaal ID: 900192802

Shahd Elmahallawy ID: 900194441

CSCE331301 –Digital Design II

Fall 2023

Table of Contents

Introduction

Overview

The efficient placement of cells in electronic circuits is a vital aspect of electronic design automation (EDA), significantly impacting the performance of electronic devices. This report explores the application of simulated annealing, a probabilistic optimization technique, to address the cell placement challenge by minimizing the total wire length in a grid layout.

Project Objective

The primary goal of this project is to investigate how different cooling rates in the simulated annealing algorithm affect cell placement optimization. By experimenting with various cooling schedules, the project aims to uncover insights into the algorithm's effectiveness and adaptability in solving complex optimization problems.

Simulated Annealing Algorithm

2.1 Algorithm Overview

Simulated Annealing (SA) is a stochastic optimization technique that draws inspiration from the physical process of annealing in metallurgical practices. It is particularly effective in finding near-optimal solutions in complex optimization tasks, such as cell placement in electronic circuits.

The algorithm mimics the process of heating a material and then gradually cooling it, allowing atoms to rearrange to form a stable crystal structure. In optimization terms, this translates to exploring a solution space at high "temperatures" and gradually reducing the "temperature" to settle into a state that is hopefully close to the global optimum.

In the project, SA is applied to optimize cell placement, focusing on studying the effects of varying the cooling rate. By systematically altering this rate and analyzing its impact on the quality of placement through temperature versus total wire length graphs and final TWL versus cooling rate plots, the project explores the delicate balance between exploration and exploitation inherent in the algorithm.

Procedure:

Initial Setup: Start with a randomly generated solution and a high initial temperature.

Iteration Loop:

- Randomly modify the current solution.
- Evaluate the new solution and compare it with the current one.
- If the new solution is better, accept it.
- If the new solution is worse, accept it with a certain probability that decreases as the temperature lowers and as the quality difference between the solutions increases.

Cooling Schedule: After each iteration, reduce the temperature according to a predefined schedule until it reaches a final low temperature.

2.2 Key Components and Parameters

For the specific application in cell placement optimization, the SA algorithm is characterized by several tailored components and parameters:

Initial and Final Temperature:

- Initial Temperature: Set as 500 times the initial cost, which represents the total wire length of the initial random cell placement. This high temperature allows extensive exploration of the solution space at the beginning.
- Final Temperature: Defined as 5x10^-6 times the initial cost divided by the number of nets, signifying the point at which the algorithm ceases execution.

Temperature Update Mechanism:

After each iteration, the temperature is updated by multiplying with a cooling rate of 0.95. This gradual reduction in temperature helps the algorithm transition from exploration to exploitation, increasingly favoring only those solutions that improve the objective.

Objective Function:

The total wire length (Half-Perimeter Wire Length - HPWL) serves as the objective function to be minimized. HPWL is a standard metric in electronic design automation for estimating the wiring complexity of a net.

Modification Approach:

A pair of cells is randomly chosen at each step, and their positions are swapped. This perturbation method allows the algorithm to traverse the solution space and find different configurations.

Acceptance Criterion:

The probability of accepting a worse solution depends on the difference in total wire length (ΔL) and the current temperature. This acceptance of inferior solutions at higher temperatures enables the algorithm to avoid premature convergence to local optima.

Moves per Temperature Level:

A predefined number of moves (swaps) are performed at each temperature level, precisely 10 times the number of cells, ensuring thorough exploration at each stage of the annealing process.

Implementation Details

3.1 Development Environment

The simulated annealing cell placement tool was developed using both C++ and Python. C++ was used to implement the core algorithm. It was chosen for its performance efficiency and

optimization. Python, known for its simplicity and powerful data visualization libraries, was used for generating graphical representations of the algorithm's output. We used libraries such as Matplotlib for plotting and PIL (Python Imaging Library) for creating animations. This setup was chosen for its ease of use in handling and displaying complex data sets.

3.2 C++ Implementation - Simulated Annealing

Data Structures

The data structures used in the implementation are:

- **grid:** `vector<vector<int>>`

  This data structure carries the cells in each grid square.

- **cells:** `vector<pair<int, int>>`

  This data structure save the current place of the cell where:

  cells[i].first is the row of the ith cell

  cells[i].second is the column of the ith cell

- **nets:** `vector<vector<int>>`

  This data structure carries the information for each net. Nets[i] is a vector of cells in the ith net.

- **nets_of_cells:** `vector<vector<int>>` – (needed for optimization)

  This data structure saves the nets of each cell. Nets_of_cells[i] has a vector of all nets that contain the ith cell.

- **wire_lengths:** `vector<vector<int>>` – (needed for optimization)

  This data structure saves the current length for each net

Functions

The following functions are used:

- **calculateWireLength**

  This is a helper function to calculate the total wire length using HPWL for a specific net based on the current cell locations saved in cells. The implementation for HPWL is

simple as it passes over all cells and calculate min_x, min_y, max_x, and max_y for thee bounding box

- **random_placement**

  This function initializes the grid with random placement for the cells.
- **print_grid and print_binary**

  These two functions are for printing the grid. In binary it prints 1 for each cell in the grid and 0 for empty sites
- **save_grid_state**

  This function is used in GIF creation as it saves a snapshot of the current grid.

Simulated Annealing Algorithm

The implementation for the algorithm is as follows:

- The parameters of the algorithm are initialized as described in the cooling schedule:
  - Initial Temperature = 500 x Initial Cost
  - Final Temperature = $5x10^{-6}$ x (Initial Cost) / (Number of Nets)
  - Next Temperature = cooling rate x Current Temperature
  - cooling rate = 0.95
  - Moves/Temperature = 10 x (Number of cells)
- When selecting the two cells to swap, to allow for selecting empty sites, the following happens:
  - Cell A is selected at random from the existing cells
  - Cell B is initialized as -1 (no cell) and instead Location of cell B is selected at random from the sites in the grid
  - The cells and grid data structures are updated in the swap
- In calculating the new cost, to optimize the performance, we did the following:
  - We noticed that the new cost = the current cost by taking into account the changed nets
  - The only changed nets are the ones that include cells A and B.
  - Therefore, we initialize the new_cost with the current_cost
  - Then, loop over nets_of_cells[a] and nets_of_cells[b]

    - For each net update the new_cost in by subtracting old_wire_length[net] and adding the new cost of the net calculated by the helper function
    - If the net is common between cells A and B, then old_wire_length is already filled and we accounted for that net. Therefore, we continue
- If the swap was successful, either because the new cost or because the probability of rejection is less than 0.5:
    - Update current_cost with new_cost
- If swap is not successful:
    - Revert all the changes in cells, grid data structures, and wire lengths

3.3 Python Implementation - Scripts

Graph Plotting

I. Data Parsing and Preparation:

Python scripts read the output files generated by the C++ code, which contain temperature and total wire length (TWL) data. The data is parsed using Python's file-handling capabilities and stored in arrays for plotting.

II. Matplotlib Integration:

- Temperature vs. TWL Plot: For each cooling rate, a line plot is created showing how the TWL changes as the temperature decreases during the annealing process. This involves plotting temperature values on the x-axis against TWL on the y-axis.
- Final TWL vs. Cooling Rate Plot: This plot compares the final TWL achieved for different cooling rates, using cooling rates as the x-axis and the final TWL values as the y-axis.

III. Plot Customization:

- Customization options like labeling axes, adding titles, adjusting line colors and markers, and implementing a grid layout are used for better readability and comprehension.
- The plt.xlim function is used to reverse the x-axis in the temperature vs. TWL plot, aligning with the cooling nature of the algorithm.

Animation Creation (GIF)

I. Data Parsing and Preparation:

It first reads the state of the grid at different iterations from the output file. Then, Each grid state is converted into an image using PIL's Image and ImageDraw modules.

II. Image Manipulation:

A rectangle is drawn on the image for each cell in the grid. The color of the rectangle varies depending on whether the cell is occupied or empty. Text annotations (cell IDs) are added to each occupied cell for clarity. The font size and color are adjusted to ensure visibility against the cell's color. We used blue for nonempty cells and black for empty cells.

III. Animation Compilation:

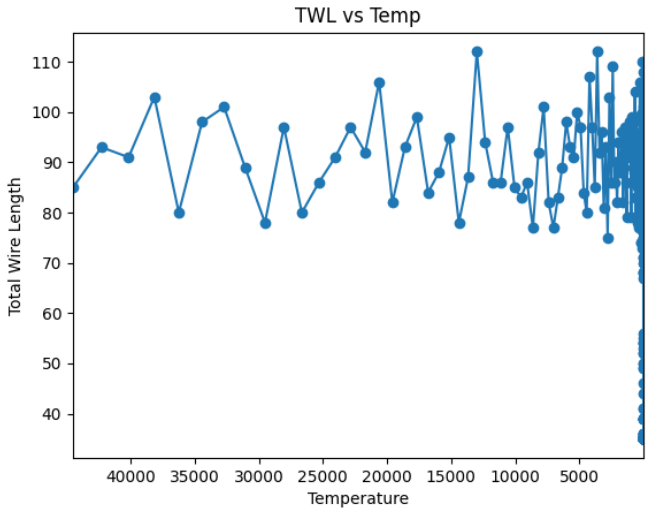
The individual frames are compiled into a GIF animation. This is achieved using the same method from PIL's Image module, where each frame is appended to create a continuous animation. Parameters like frame duration and loop settings are configured to control the animation's speed and repetition behavior.

Results and Analysis

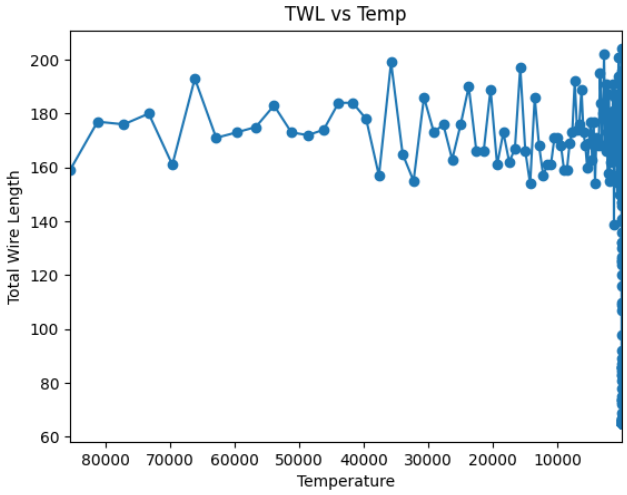
4.1 Temperature vs. Total Wire Length

Results

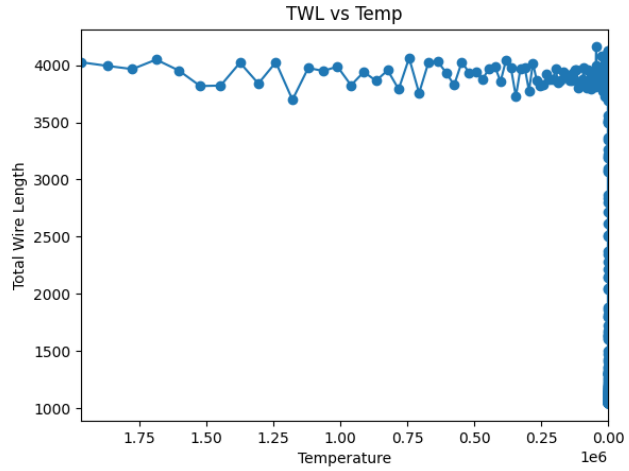
d0 Result



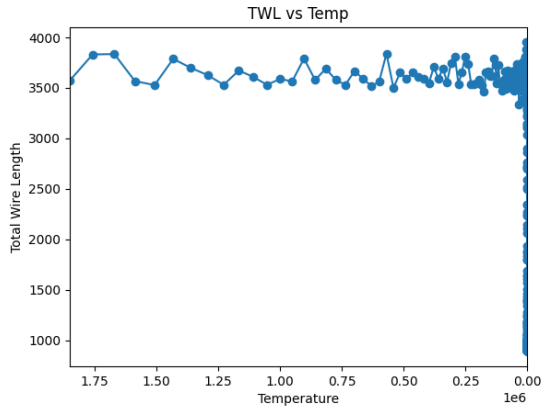
d1 Result



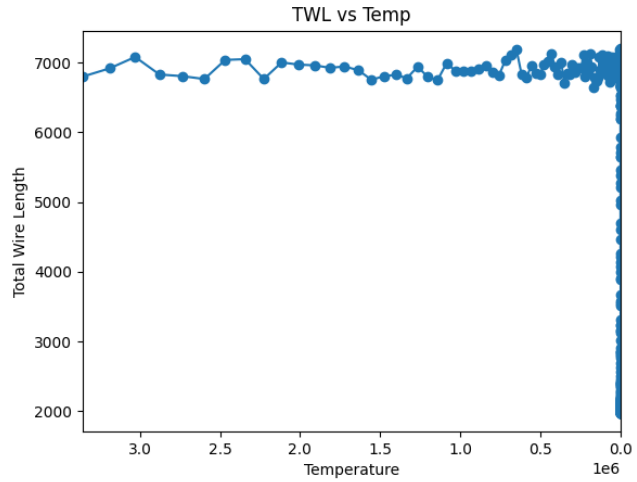
d2 Result



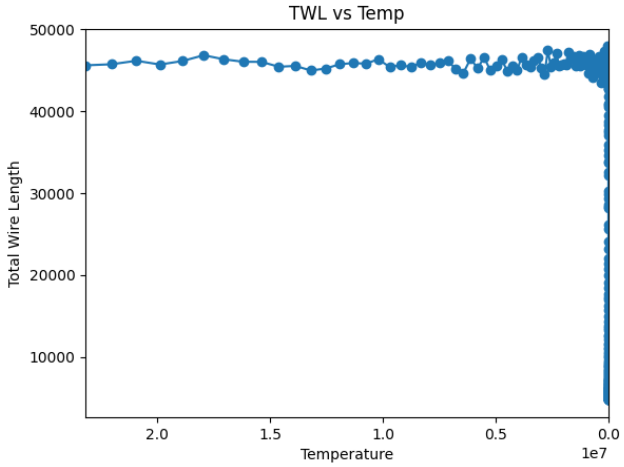
d3 Result



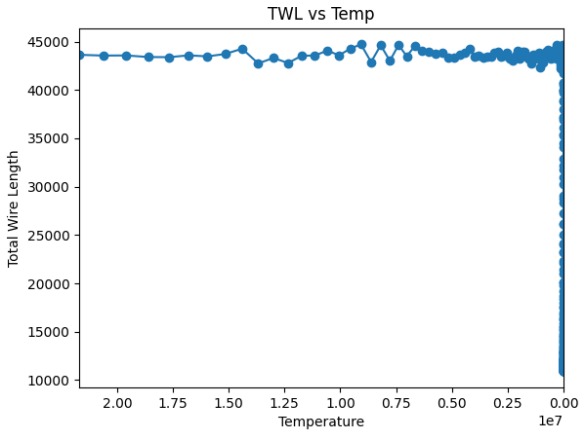
t1 Result



t2 Result



t3 Result



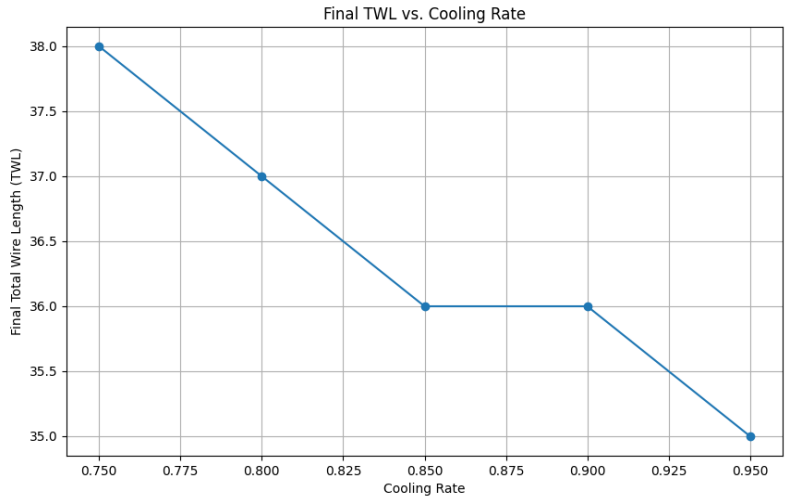
Analysis

At higher temperatures, more fluctuations in TWL are observed. This is due to the algorithm's willingness to explore the solution space more freely, accepting both good and bad swaps. As the temperature approaches the final value, which means it cools down, it only accepts better changes, so the fluctuations in TWL reduce.

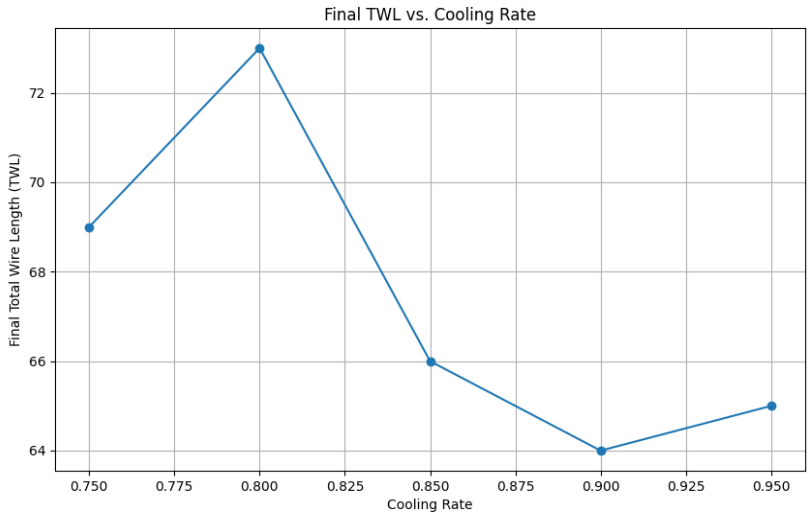
4.2 Final Total Wire Length vs. Cooling Rate

Results

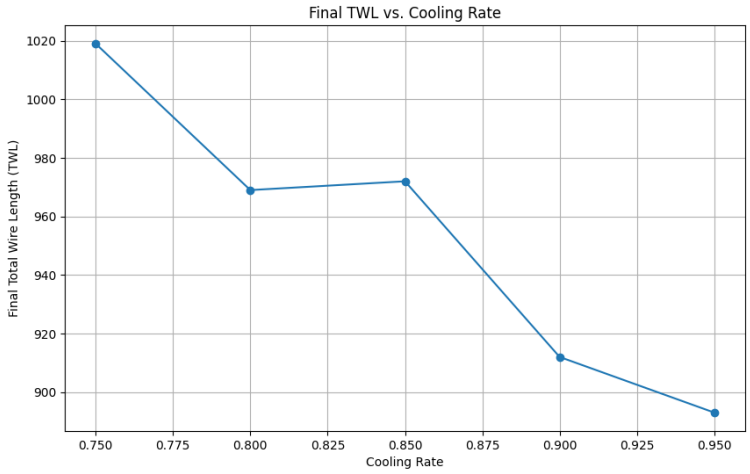
d0 Result



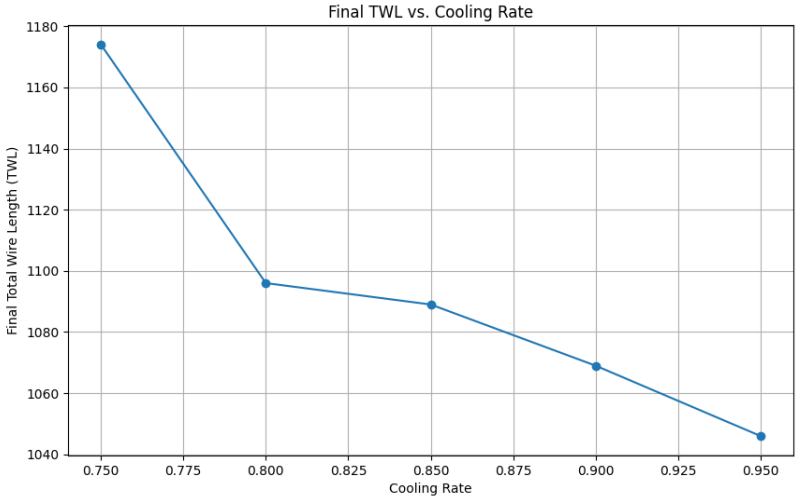
d1 Result



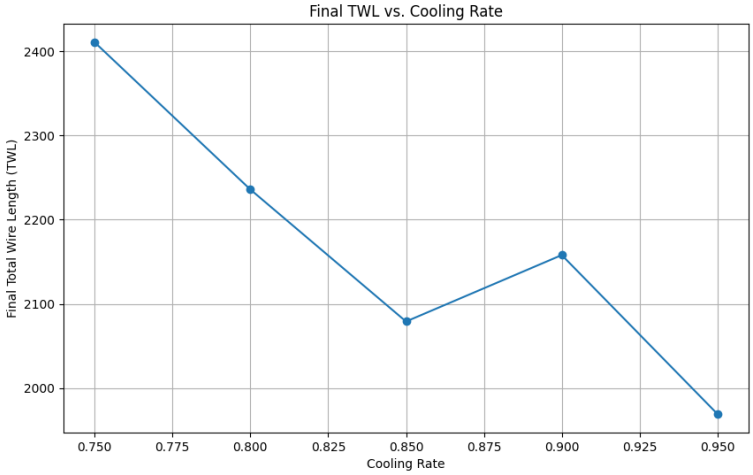
d2 Result



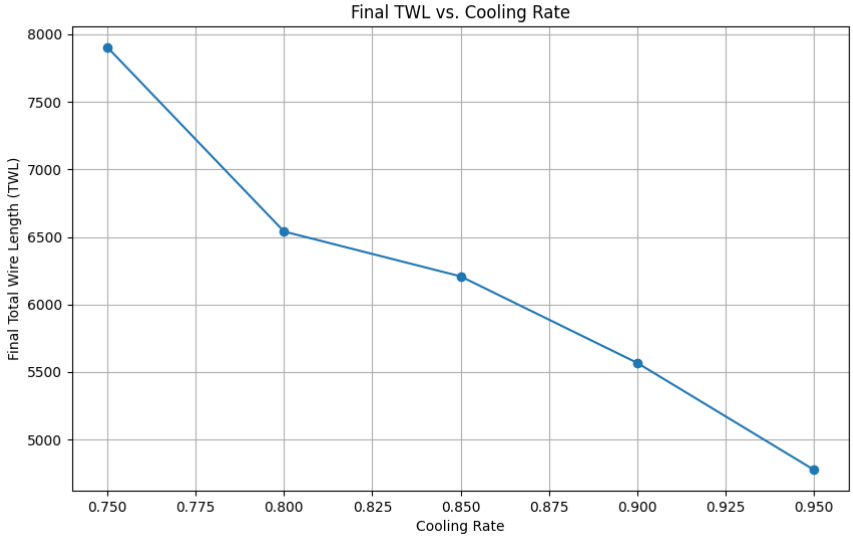
t0 Result



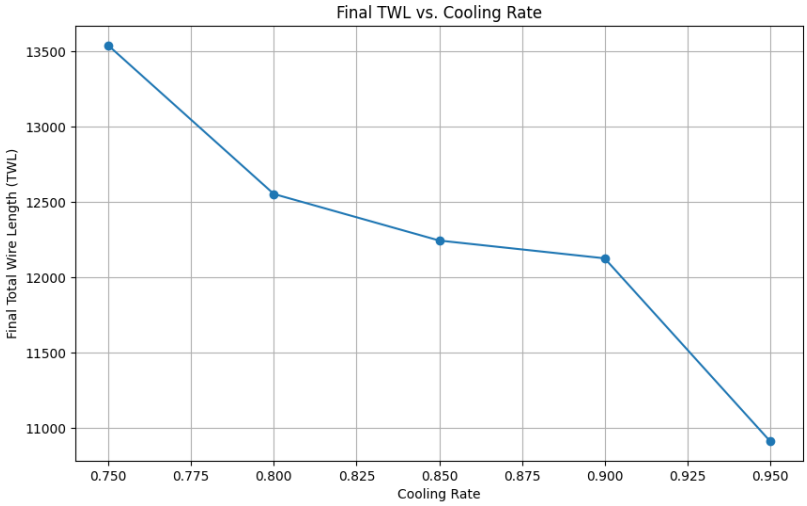
t1 Result



t2 Result



t3 Result



Analysis

TWL decreases while increasing the cooling rate, as it allows for more iterations in the simulated annealing to find the optimal wire length, potentially leading to a better-optimized solution but at the cost of increased computational time.

4.3 GIF Animation of Cell Placement Process (Bonus Feature)

We have done the bonus feature to illustrate the cell placement adjustments better. At the beginning of the animation, cells are likely to be more randomly distributed across the grid. This reflects the initial random placement of cells, where no optimization has yet been applied. As the animation progresses, especially in the intermediate stages, you might observe cells starting to move closer together. This behavior is indicative of the algorithm's attempts to reduce the total wire length by minimizing the distance between connected cells.

Conclusion

5.1 Effectiveness of Simulated Annealing Algorithm

The Simulated Annealing (SA) algorithm has demonstrated its effectiveness in optimizing cell placements to minimize total wire length. Unlike greedy algorithms, SA's

probabilistic nature enables it to escape local optima, making it a valuable choice for complex optimization problems with multiple local minima. The algorithm's ability to escape local optima is evident through the gradual improvement in wire length observed over multiple iterations.

5.2 Impact of Cooling Rate

The cooling rate parameter has proven to be critical in the algorithm's performance. Slower cooling rates generally lead to better optimization outcomes by allowing more extensive solution space exploration. However, there exists a trade-off with computational time when opting for slower cooling rates.

5.3 Program Outputs and Performance

Our program has been successful in providing valuable outputs throughout its execution. It outputs cell positions both before and after placements, along with their binary representations. The program also displays wire lengths before and after executing the algorithm, aiding in result assessment. For further analysis and visualization, it saves both the Total Wire Length (TWL) and temperature data to text files, facilitating visualization in Python code. Additionally, it preserves the grid status after each temperature iteration, which can be utilized for generating GIFs. All generated graphs and GIFs are saved for reference and presentation purposes.

5.4 Performance

Our code has undergone comprehensive testing and has successfully passed all test cases. In the largest test case, "t3.txt," the program performed efficiently, completing in just under 4 seconds.