THE AMERICAN UNIVERSITY IN CAIRO

الجامعة الأمريكية بالقاهرة

Digital Design 2 Project:
A Simple Simulated-Annealing Cell Placement Tool

Abdulrahman Magdy Abufarag - 900202484
Habiba Mohamed Elsayed - 900213883

Introduction:

The objective of this project is to implement a simulated annealing algorithm for cell placement on a grid to minimize the total wire length (TWL). The simulated annealing technique is used to find an optimal solution by iteratively improving the placement while exploring a wide search space, making it suitable for combinatorial optimization problems like cell placement.

Algorithm:

Simulated annealing is a probabilistic technique inspired by the annealing process in metallurgy. It involves heating and then slowly cooling a material to remove defects and optimize its structure. The key components of the algorithm are:-

1) Initial Temperature: A high starting temperature that allows the system to explore a wide search space.
2) Cooling Schedule: A function that gradually reduces the temperature, typically by multiplying the current temperature by a cooling rate.
3) Acceptance Probability: At each step, the algorithm may accept worse solutions with a probability that decreases with temperature, allowing it to escape local minima.

Half-Perimeter Wire Length (HPWL) is used as a metric to estimate the wire length of a net. It is the half-perimeter of the smallest bounding box containing all pins of a net. This simplifies the computation while providing a good approximation of the actual wire length.

Implementation:

Data Structures used includes:-

1) Grid: A 2D array representing the placement sites.
2) Cell Positions: A dictionary mapping each cell to its coordinates on the grid.
3) Netlist: A list of nets, where each net is a list of cells connected by that net.

Steps:

1) Reading Input: The netlist file is read, and the grid size and nets are extracted.
2) Initial Placement: Cells are randomly placed on the grid.
3) Calculating Initial HPWL: The initial total wire length is computed using HPWL.
4) Simulated Annealing Process:
    - Iteratively swap cells and calculate the new HPWL.
    - Accept the new placement if it reduces the HPWL, or probabilistically accept worse placements based on the temperature.
    - Gradually reduce the temperature according to the cooling schedule.
5) Output: The final placement and the total wire length are printed, and graphs showing the cooling rate vs. final TWL and temperature vs. TWL are generated.

Implementation of each function:

1) **Update_hpwl_for_cells**

This function updates the half-perimeter wire length (HPWL) for specific nets and computes the total wire length. Given a list of net indices (cell_indices), the current HPWL list (hpwl_list), the netlist (nets), and the current positions of the cells (cell_positions), the function recalculates the HPWL for each net in cell_indices. It iterates through each net, computes the x and y coordinates of all cells in the net, and calculates the HPWL as the sum of the horizontal and vertical extents of the bounding box that contains all cells in the net. Finally, it returns the updated HPWL list and the new total wire length.

2) **Initialize_hpwl**

This function initializes the HPWL for all nets in the netlist. It takes the netlist (nets) and the current positions of the cells (cell_positions) as input. For each net, it calculates the x and y coordinates of all cells in the net and computes the HPWL as the sum of the horizontal and vertical extents of the bounding box containing all cells in the net. The function then returns a list of HPWL values for each net and the total wire length.

3) **Print_grid**

This function prints the current grid to the console. It takes the current positions of the cells (cell_positions) and the grid size (grid_size) as input. The function creates a 2D array representing the grid, where each cell is initially marked as '1' (indicating an empty site). It then updates the grid to reflect the positions of the cells by setting the corresponding positions to '0'. Finally, it prints the grid row by row.

4) **Read_netlist**

This function reads the netlist from a file. It takes the filename (filename) as input and returns the grid size and the netlist. The function opens the file and reads the first line to extract the grid size (number of rows and columns). It then reads the remaining lines to build the netlist, where each line represents a net and contains a list of cells connected by that net. The function returns the grid size and the netlist.

5) **Initial_placement**

This function performs an initial random placement of the cells on the grid. It takes the set of cells (cells) and the grid size (grid_size) as input and returns a dictionary mapping each cell to its coordinates on the grid. The function ensures that no two cells occupy the same position by maintaining a set of occupied positions. For each cell, it randomly selects an unoccupied position on the grid and updates the dictionary and the set of occupied positions.

6) **Create_grid**

This function creates a 2D grid representation based on the current positions of the cells. It takes the current positions of the cells (cell_positions) and the grid size (grid_size) as input and returns a 2D array representing the grid. The function initializes the grid with "--" to indicate empty sites and updates it to reflect the positions of the cells based on the cell_positions dictionary.

7) **Swap_cells**

This function swaps the positions of two cells on the grid. It takes the current positions of the cells (cell_positions), the grid (grid), the two cells to be swapped (cell1 and cell2), and their respective positions (pos1 and pos2) as input. If either cell is an empty site (indicated by "--"), the function updates the position of the non-empty cell. Otherwise, it swaps the positions of the two cells.

8) **Simulated_annealing**

This function performs the simulated annealing optimization for cell placement. It takes the filename (filename), the cooling rate (cooling_rate), and the GIF name (gif_name, although not used in this version) as input. The function performs the following steps:

- Reads the netlist and initializes the grid and cell positions.
- Calculates the initial HPWL and sets the initial and final temperatures.
- Iteratively performs cell swaps to optimize the placement, using a probabilistic acceptance criterion based on the current temperature.
- Gradually reduce the temperature according to the cooling schedule.
- Updates the HPWL and records the temperature and wire length at each step.
- Prints the initial and final placements and wire lengths, and returns the final results.

9) **Plot_graphs**

This function plots the temperature vs. total wire length (TWL) graph. It takes the list of temperatures (temps), the list of total wire lengths (hpwls), and the cooling rate (cooling_rate) as input. The function creates a plot showing how the TWL changes with temperature, using a logarithmic scale for both axes. The plot helps visualize the optimization process and the convergence of the simulated annealing algorithm.

10) **Plot_cooling_rate_vs_twl**

This function plots the cooling rate vs. final total wire length (TWL) graph. It takes the filename (filename) as input and evaluates the performance of the simulated annealing algorithm for different cooling rates. The function iterates through a list of cooling rates, runs the simulated annealing algorithm for each rate, and records the final TWL. It then creates a plot showing the

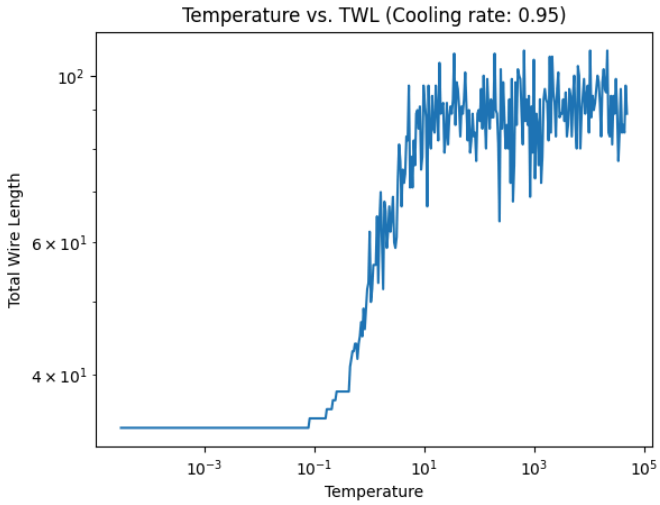
final TWL as a function of the cooling rate. This plot helps evaluate the impact of the cooling rate on the optimization results.

11) Main Execution

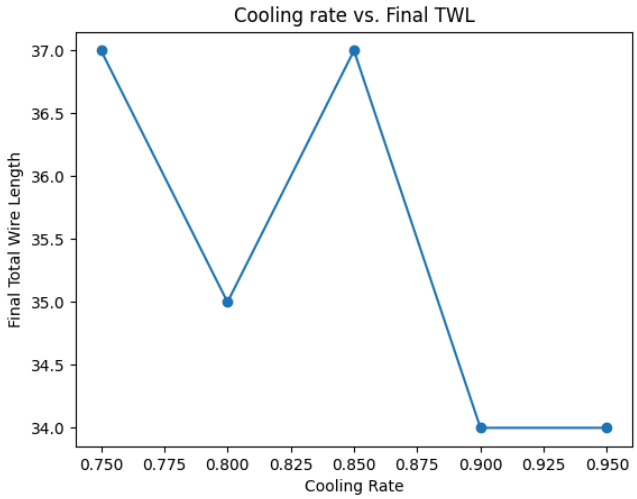
The main execution block sets up the argument parser to accept the filename from the command line, initializes the start time, and runs the simulated annealing algorithm with a specified cooling rate. It then plots the temperature vs. TWL and cooling rate vs. TWL graphs and prints a completion message.

Graphs:

D0.txt
Temperature vs TWL

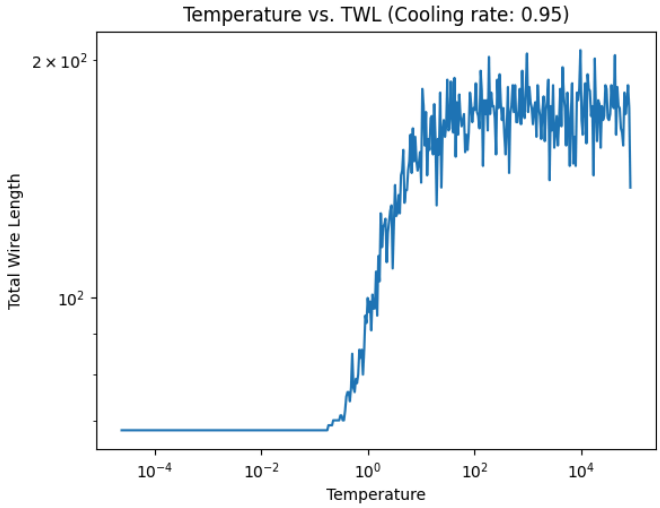


Cooling Rate vs TWL

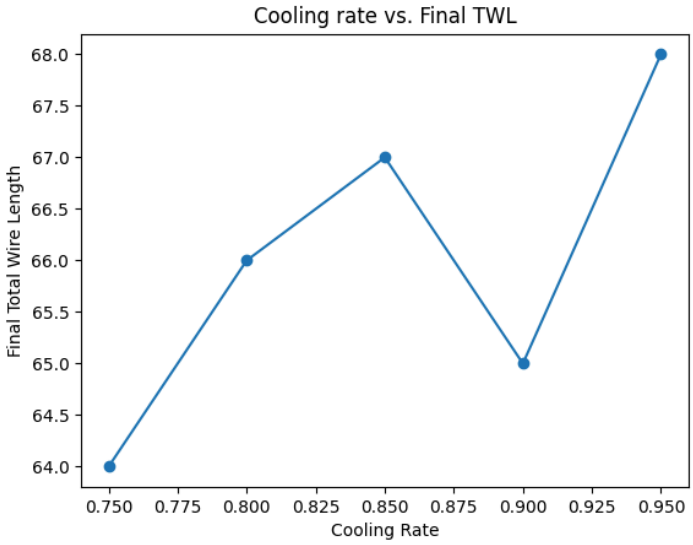


D1.txt

Temperature vs TWL

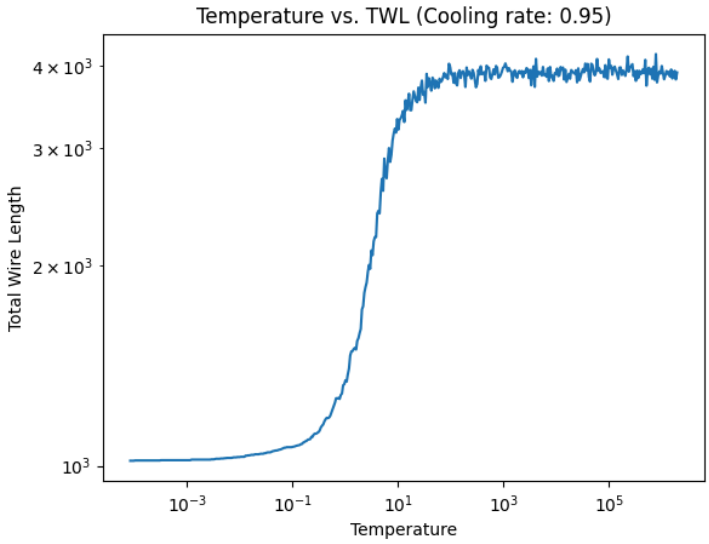


Cooling Rate vs TWL

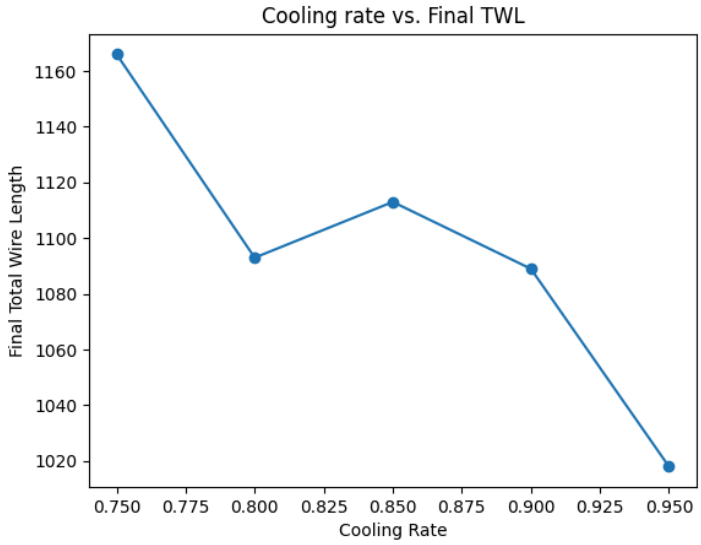


D2.txt

Temperature vs TWL

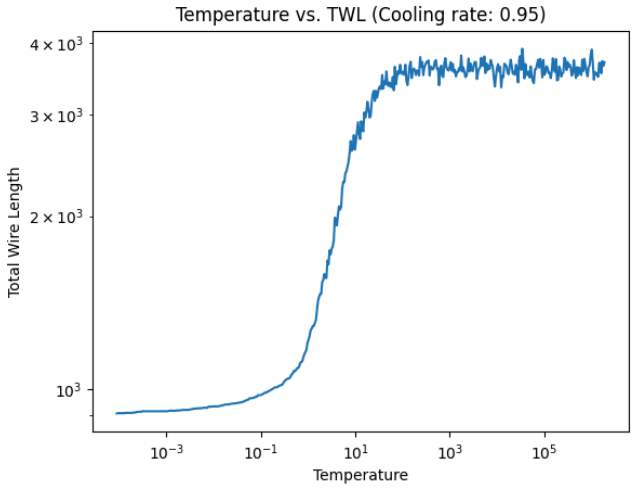


Cooling Rate vs TWL

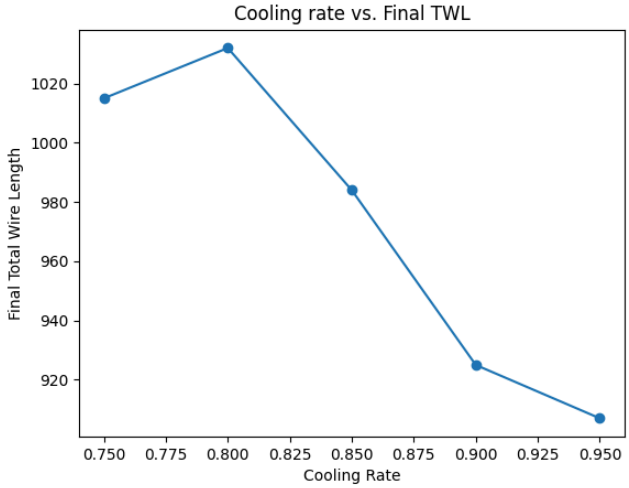


D3.txt

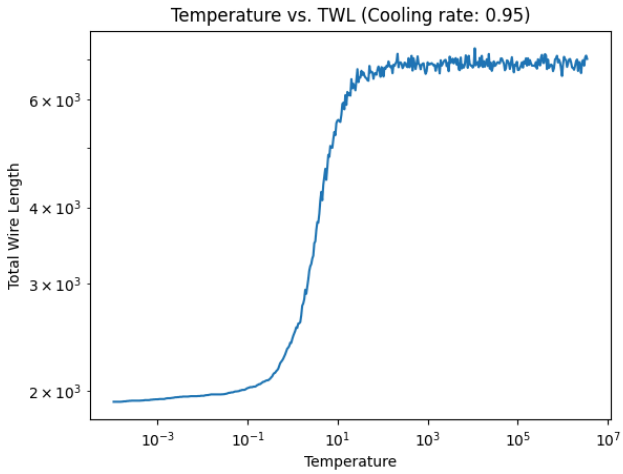
Temperature vs TWL



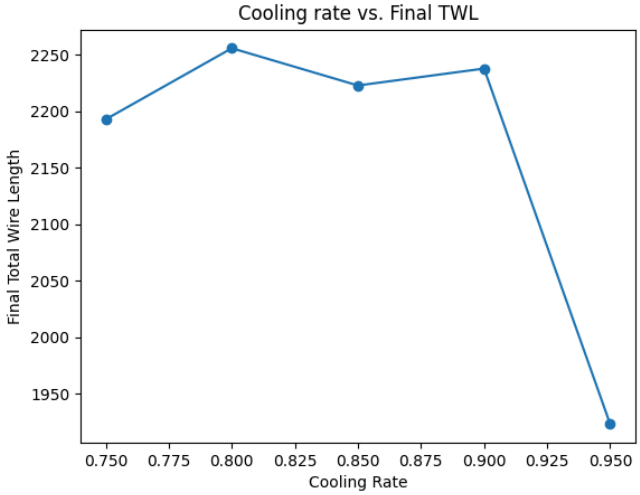
Cooling Rate vs TWL



T1.txt
Temperature vs TWL



Cooling Rate vs TWL



Conclusion:

The simulated annealing algorithm effectively improves cell placement on a grid to reduce total wire length, making it useful for complex optimization problems. By allowing occasional acceptance of worse solutions, it can avoid getting stuck in poor solutions and move towards a better overall arrangement. Our tests with different cooling rates show that higher rates usually result in better final placements, though they require more time to compute. The graphs of temperature versus wire length show how the algorithm gradually finds better placements, with initial large changes that stabilize as the temperature drops. In summary, simulated annealing is a strong method for finding efficient cell placements, balancing between exploring new possibilities and refining existing ones.