ALGORITHMS FOR VLSI

INTEGRATED CIRCUIT CELL PLACEMENT ALGORITHM

Kymry Burwell

January, 2019

INTRODUCTION

The goal of this project is to implement an algorithm for placing cells on an integrated circuit with an objective of minimizing total wire length.

Cell placement is an essential step in the design of electronic circuits. The end goal is to assign exact locations on the chip for various circuit components. Unfortunately this problem is known to be NP-complete, therefore heuristics are required. The problem can be defined as an optimization problem with one or more varying objectives. The most common are the following:

- Minimizing the total wire-length.
- Avoiding local congestion - groups of cells placed closed together that may lead to excessive routing detours.
- Minimizing total power consumption.
- Ensuring adequate timing - meeting the requirements of clock cycle.

The algorithm implemented for this report will focus solely on minimizing total wire length.

The effects of a cell placement can be dramatic later on in the design process - bad placement can lead to chips that cannot be built because of excessive wire-length and inadequate routing resources. Therefore, it is essential to design and build quality cell placement algorithms. There are a multitude of techniques used in such algorithms, however we will be focusing on following two: linear ordering of cells using eigenvectors for the initial placement and simulated annealing to improve the placement.

The report is divide into four sections as follows:

- **Algorithm overview** - provides an outline of the main algorithm, along with comments about decisions made during the building process.
- **Execution test results** - provides the results of two different tests run on the algorithm.
- **C++ program manual** - provides a more detailed view into the implementation by providing a description of each .cpp file.
- **Algorithm output** - provides the final cost and placement of two small netlists.

ALGORITHM OVERVIEW

The algorithm can be separated into two main parts: obtaining the eigenvectors from the netlist and using the result for an initial placement and using simulated annealing to improve the placement by swapping and moving individual cells. The following pseudo-code named **Algorithm 1** provides an outline of the main steps in the cell placement algorithm.

Algorithm 1 Generate Exact Cell Placement

```
1: procedure CELL_PLACEMENT(num_nodes, net_list, chip_size)
2:     if net_list is NULL then
3:         net_list ← Generate_Random_Graph(num_nodes)
4:     laplacian_matrix ← Convert_Laplacian(net_list)
5:     eigenvectors ← Compute_Eigenvalues(laplacian_matrix)
6:     initial_chip ← Initial_Placement(eigenvectors, chip_size)
7:     final_chip ← Sim_Annealing(initial_chip, net_list)
8:     return chip
```

In lines 2 and 3 either a netlist is provided or a random graph is generated. The random graph may be a k-regular, geometric random, or Erdos Renyi. The parameters of each graph are fully customizable. I decided to add functionality for multiple random graphs to enable thorough testing

Next, in line 4, a Laplacian matrix is created from the netlist or random graph. A Laplacian matrix is used for the computation of Eigenvalues and their associated Eigenvectors. A Laplican matrix is of the form $L = D - A$, where D is a degree matrix and A is an adjacency matrix of the graph. That is, L is defined as:

$$L(i,j) = \begin{cases} \text{degree(i)} & \text{if i = j} \\ -1 & \text{if i} \neq \text{j and i, j are connected} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

In the case of a graph with varying edge weights, the diagonal is the sum of the weights of adjacent edges for all vertices.

In line 5, the 2nd and 3rd smallest eigenvectors are obtained from the Laplacian matrix. The C++ Library *Spectra*, which is built on top of the library *Eigen* is used. Both are free to

use open-source libraries. In order to compute the eigenvectors, the Laplacian matrix is converted into a specialized data type defined by $Eigen$. I decided to use Spectra because the API manual was thorough and easy to use. Additionally, Eigen is well-known and seems to provide good performance.

We use the eigenvectors of the second and third smallest eigenvalues for the x and y coordinates of the cells, respectively. The goal here is to minimize the squared wire-length. Let x_i and x_j be the coordinates of cells i and j, and E all pairs of connected cells, then our goal is to minimize the following equation:

$$\sum_{i,j \in E} (x_i - x_j)^2 \tag{2}$$

It has been shown in [3] that this is equivalent to:

$$\min_x X^T L X$$
$$\text{subject to: } X^T X = 1$$

Where $X^T = [x_1, ...x_n]$ is the row vector of x-coordinates. The constraint $X^T X = 1$ is necessary otherwise $x_i = 0$ for all i. Then, X can be found by solving $LX = \lambda_2 X$, where λ_2 is the second smallest eigenvalue. The same method is use for obtaining the y-coordinates using the third smallest eigenvalue.

In line 6, the eigenvectors are used to make the initial cell placement. The coordinates provided by the eigenvectors are in the range $[-1, 1]$ which means the results must be adapted to the chip size. In this algorithm, I decided to place all values in the range $[-1, 0)$ on the left side of the chip (lower half for y-coordinates) and all values in the range $[0, 1]$ on the right side (upper half for y-coordinates). The coordinates are scaled according to the chip size.

During the initial placement, it may be the case that a location on the chip is already occupied and thus a cell cannot be placed there. If this happens, the cell currently being placed is placed in the next available location. This location is found by looking at all locations within a distance of 1, then at a distance of 2, etc. until an open space is found. I chose to implement this method because it ensures the cell is placed near it's optimal starting position. An overview of this step is displayed in **Algorithm 2** below.

In line 7, the initial cell placement is improved upon using simulated annealing. Simulated annealing is a probabilistic adaptation of a greedy algorithm that uses a technique found in the natural world known as annealing. At each step of the annealing process, two small changes in the chip placement are attempted: *SWAP* and *MOVE*. *SWAP* takes two cells at random and swaps their positions in the chip. *MOVE* takes one cell at random and places it in an open position, also at random. If the overall cost is improved (i.e. wire-length is decreased) after each of the changes, the changes are immediately accepted. If the cost does not improve, the changes are accepted with some probability as a function of the current temperature. A more detailed overview of the the annealing process can be found in **Algorithm 3** below.

Finally in line 8, the final cell placement is returned.

Algorithm 2 Set Initial Cell Placement

```
1: procedure INITIAL_PLACEMENT(eigenvectors, chip_size)
2:     for i ← 1 to eigenvectors.length() do
3:         current_cell ← i
4:         x, y ← Get_Coordinates(eigenvectors[i])
5:         if chip[x][y] is not NULL then
6:             x, y ← find_closest_open_location(x, y)
           chip[x][y] ← i
```

Algorithm 3 Simulated Annealing

```
1:  procedure SIM_ANNEAL(Temp, ϵ, α, netlist, cost)
2:      while Temp > ϵ do
3:          reject = 0
4:          while iter < netlist.size()*4 and reject < netlist.size() do
5:              iter ← iter + 1
6:              a, b ← select_cell_pair()                          ▷ random pair
7:              new_cost ← try_swap(a,b)
8:              if new_cost < cost then
9:                  make_swap(a,b)
10:                 cost ← new_cost
11:             else if rand_num() < (e^(-(new_cost - cost)/Temp)) then   ▷ chosen uniformly at random
12:                 make_swap(a,b)
13:                 cost ← new_cost
14:             else
15:                 reject ← reject + 1
16:             a ← select_cell()                                  ▷ random cell
17:             new_cost ← try_move(a)
18:             if new_cost < cost then
19:                 make_move(a,b)
20:                 cost ← new_cost
21:             else if rand_num() < (e^(-(new_cost - cost)/Temp)) then
22:                 make_move(a,b)
23:                 cost ← new_cost
24:             else
25:                 reject ← reject + 1
26:         Temp ← Temp * α
27:     return (cost)
```

EXECUTION TEST RESULTS

Computing Eigenvectors and simulated annealing are both computationally expensive operations, so I was curious as to how the algorithm would perform with an increasing number of nodes. I was also curious as to how well the simulated annealing performed, that is, by how

much the overall wire-length was reduced.

Test 1: Algorithm Computation Time - In order to test the computational performance, the algorithm was run on graphs with a varying number of nodes and the execution time (in seconds) was recorded for each run. The graphs started small with 100 nodes and increased in size to 5000 nodes (note: I did not test any large graphs as my computer is nearly a decade old and can't handle large computations.) I tested with three different types of graphs, including a 3-regular graph, Erdos Renyi with a probability of 0.2, and geometric with a radius of 0.2. The results are displayed in figure 1 below.

We can see that with a very small number of nodes the computation times don't differ by much, but as the number of nodes increases the computation times vary quite dramatically. Much, of this variation is likely due the the sparsity of the graphs.

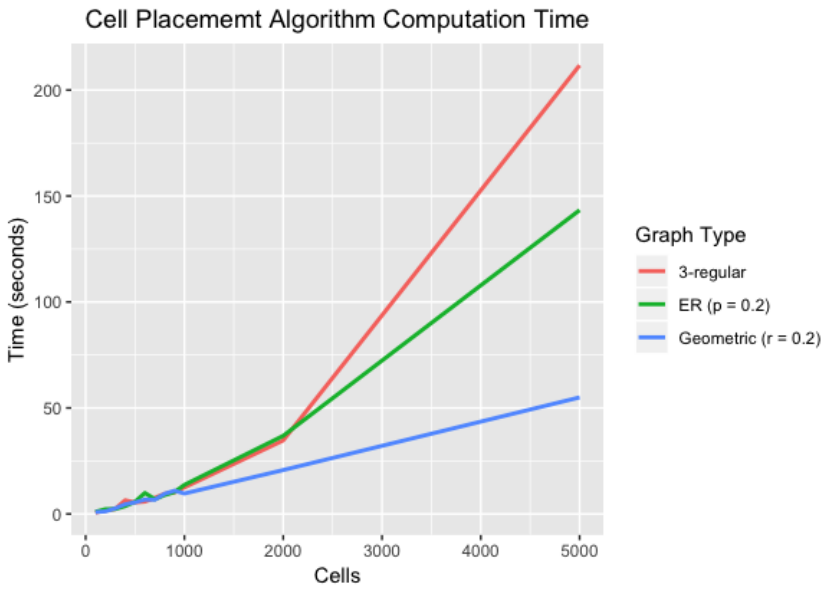

Figure 1

Test 2: Simulated Annealing Performance - In order to test the overall performance of the simulated annealing procedure I ran the algorithm with 5000 nodes and recorded the total wire-length at each temperature during the annealing process. I used a geometric random graph with a radius of 0.2. The results are displayed in figure 2 below.

dThe x-axis of figure 2 is in log10 scale to better show the true annealing process, as there are

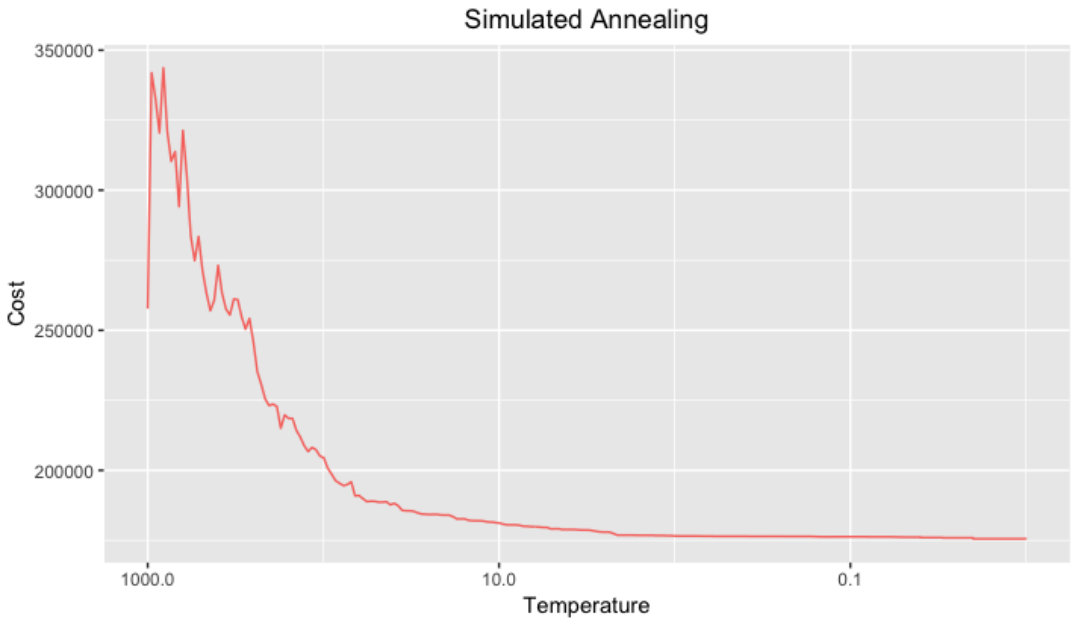

Figure 2

more iterations with lower temperature. The y-axis is the total cost (wire-length). We can see that indeed, simulated annealing seems to work quite well. It's curious to see the cost jump a lot during the first step of the annealing process while almost all changes are recorded (due to the high temperature), but cost increases become less and less frequent as the temperature decreases.

C++ Program Manual

This section provides an overview of each *cpp* file included in the program. It details what the file accomplishs, along with the main functions and the the most important parameters. For more detailed information, please see each corresponding file in the attached *src* folder. The code is commented well and the the header files provide a good API into the program. Additionally, names (e.g. function, parameter, variable) were chosen in an attempt to make their usage intuitive.

generate_graphs.cpp

This file has three functions: generating random graphs, reading a graph from a file, and printing a graph to the console or file.

The *Create_Random_Graph()* function generates random graphs using the $igraph$ library. The graph represents a netlist, with the nodes representing logic cells and the edges the wires between them. The graphs are undirected and edges have no weight. There are three types of random graphs that can be generated:

- K-Regular - All vertices have same degree, k.
- Geometric - Nodes are place randomly in the plane. An edge connects nodes within a specified range.
- Erdos Renyi - Any two nodes are connected with a specified probability, p.

The *read_edgelist()* function reads a netlist from a file. The netlist must be in the form of an edge list.
The *print_igraph_edgelist()* prints the input $igraph$ edge list to the console or file for later use.
All graphs are returned in the adjacency matrix representation.

convert_laplacian.cpp

This file simply converts the input netlist (adjacency matrix) to a laplacian matrix. The function is generic and can convert any symmetric matrix to a laplacian matrix.

get_eigenvectors.cpp

This file has the single function of finding the second and third smallest eigenvalues and their associated eigenvectors. It utilizes the $Spectra$ library for finding the eigenvectors which itself is built on top on the $Eigen$ library.

cell_placement.cpp

This file has two functions: adapting and scaling the eigenvectors to the chip size and finding alternate cell placement if a chip location is already occupied.

The *initial_placement()* function takes as input the chip and the two eigenvectors. The eigenvector values are in the range $[-1,1]$ and therefore must be adapted to find the chip coordinates. The function iterates through each of the eigenvectors simultaneously, adapts the values to the size of the chip by converting the values to x/y-coordinates, and places them

in the corresponding cell location.

If the cell location is already occupied, the *find_alternate_placement()* function is called. This function finds the nearest available cell location by searching in a sort of spiral fashion. It searches all cells within a radius of 1, then all cells within a radius of 2, etc. until an open space is found. This process is usually fairly quick.

simulated_annealing.cpp

This is the most complex file in the program and has three main functions: obtaining the cell coordinates from the adjacency matrix, computing the solution cost vector (vector containing the total wire-length attached to each cell), and performing simulated annealing.

The *get_cell_coordinates()* function obtains the x/y-coordinates of each cell after the initial eigenvector placement. It simply uses the matrix row and column ids for the coordinates.

The *compute_solution_cost_vector()* function computes the total wire-length associated with each cell using the Manhattan distance between itself and all of the adjacent cells.

The *sim_anneal()* function is the main function to compute the simulated annealing. This function calls a number of sub-functions during its execution, the two most important being *swap()* and *move()*, which swap 2 cells and move 1 cell to an open position, respectively.

generic_graph_operations.cpp

As the name implies, this file performs generic graph operations that are used throughout the program. There were several instances where I needed to convert an adjacency list to an adjacency matrix, print a vector, etc. and decided to create functions for them. The following functions are available.

- *coordlist_to_matrix()* - Converts a coordinate list to an adjacency matrix.
- *print_matrix()* - Prints an adjacency matrix to the console.
- *print_adj_list()* - Prints an adjacency list to the console.
- *print_solution_cost_vector()* - Prints the cost of each cell in the netlist to the console.
- *matrix_to_list* - Converts and adjacency matrix to an adjacency list.

ALGORITHM OUTPUT

In order to provide sample output, two small netlists were created from a geometric random graph with a radius of 0.2. The first netlist ((a) in figure 3 below) is of size 20 with a 5x5 chip. The second netlist ((b) in figure 3 below) is of size 30 with an 8x8 chip. The netlists were then fed to the algorithm and the initial and final placements, along with the associated costs, are displayed. The initial placement is the chip after the second and third eigenvectors were used as coordinates. The final placement is the chip after simulated annealing was also applied. The cost is the total wire-length using Manhattan distance.

Looking at the "Initial Placement" we can see that it's actually quite decent. In general, cells that have a wire between them are placed close together, although this isn't always the case. Looking next at the final placement, we can see that the cost decreased quite drastically, greatly improving the cell placement. We can also observe that the cells in the final placement are more centralized on the chip, which makes logical sense. Finally, we see that connected cells are more closely grouped on the chip than in the initial placement, decreasing the wire-length.

FINAL COMMENTS

Overall, this was a very interesting and fun project. It really shed light into the complexity of designing an integrated circuit; cell placement is one small phase of the process but is itself extremely complex and error prone. I'm still learning C++ so it provided a great opportunity to increase my programming skills.

```
----- Netlist -----
1: 2 3
2: 1 3
3: 1 2
4: 5
5: 4 6
6: 5
7: 8
8: 7
9:
10:
11: 12 13
12: 11 13
13: 11 12 15
14: 15 16 19 20
15: 13 14 16 19 20
16: 14 15 19 20
17:
18:
19: 14 15 16 20
20: 14 15 16 19

Initial Cost: 112
----- Initial Placement ------
3 5 1 18 20
4 2 11 14 0
7 12 6 16 0
10 8 9 19 0
13 15 17 0 0

Final Cost: 58
----- Final Placement ------
1 2 5 9 8
3 17 6 4 7
0 18 0 20 0
0 11 16 19 14
10 12 13 15 0
```

(a) 20 cells with 5x5 grid

```
----- Netlist -----
1: 2 3 9
2: 1 3
3: 1 2 9 10
4: 7 8
5: 9 11
6: 12
7: 4 8
8: 4 7
9: 1 3 5 10 11 14
10: 3 9 11 14 16
11: 5 9 10 14 15 16 17
12: 6 13 18
13: 12 18 19
14: 9 10 11 15 16 17
15: 11 14 16 17
16: 10 11 14 15
17: 11 14 15 20
18: 12 13 19
19: 13 18
20: 17 22 25
21: 24
22: 20 25 27
23: 26
24: 21 28 30
25: 20 22 26 27
26: 23 25
27: 22 25 29
28: 24 30
29: 27
30: 24 28

Initial Cost: 412
----- Initial Placement ------
2 11 0 27 3 0 0 0
16 21 9 20 26 14 0 0
24 28 30 0 0 0 0 0
23 25 0 8 12 13 0 0
17 29 0 18 4 5 0 0
15 22 0 6 1 7 0 0
0 0 19 10 0 0 0 0
0 0 0 0 0 0 0 0

Final Cost: 146
----- Final Placement ------
0 0 0 7 0 0 0 0
0 30 8 4 0 0 0 0
0 28 24 0 0 23 26 25
0 21 0 0 0 29 20 27
0 12 0 0 0 0 17 22
19 18 13 0 0 10 14 16
0 6 0 0 9 11 15 0
0 0 0 2 3 1 5 0
```

(b) 30 cells with 8x8 grid

Figure 3: algorithm output

REFERENCES

[1] Kahng, Andrew B., et al. VLSI physical design: from graph partitioning to timing closure. Springer Science Business Media, 2011.

[2] Lim, Sung Kyu. Practical problems in VLSI physical design automation. Springer Science Business Media, 2008.

[3] Shahookar, Khushro, and Pinaki Mazumder. "VLSI cell placement techniques." ACM Computing Surveys (CSUR) 23.2 (1991): 143-220.

[4] Alpert, Charles J., and So-Zen Yao. "Spectral partitioning: the more eigenvectors, the better." Proceedings of the 32nd annual ACM/IEEE Design Automation Conference. ACM, 1995.