Probabilistic Placement for Integrated Circuit Design: ECE1387

Author: Nicholas V. Giamblanco
SN: 1000 324 534
University of Toronto

03/11/2017

Contents

Chapter 1

Introduction

Placement is of large concern for many areas of research and work. The task of constructing a new condominium in downtown Toronto or synthesizing an integrated circuit requires a concept of placement. A condominium cannot simply be *placed* anywhere during construction: it must analyzed to be *optimal* for many constraints (i.e. closeness to public transit, restaurants, amusement and attractions, etc.). As the size of the constraints increase, an optimal *placement* of a new condominium increases. These general ideas apply to the placement of elements within an integrated circuit. Specifically, we are concerned about an automatic approach to placement of elements within an integrated circuit while adhering to a set of constraints.

In this experiment, we implement an automated placement algorithm. This algorithm used the *Half Perimeter Wire Length* (HPWL) as a constraint, where the HPWL must be minimal. To minimize the HPWL, we formulated a system of quadratic equations to be minimized. This placement algorithm is similar to other analytical placement algorithms [5]. In this report we investigate a probabilistic spreading technique which aims to help reduce the HPWL.

Chapter 2

Design Considerations

In order to explore the ideas behind analytical placement algorithms, we must familiarize ourselves with structures required by the algorithms. We will then briefly describe some mathematical notation and formulations required for discussion[1].

This experiment requires us to be familiar with high level views of integrated circuits. In the realm of integrated circuits, we will be concerned with the placement Circuit Elements, which we will refer to as Blocks. These blocks can be viewed as a placeholder for any type of logical or mathematical operation (i.e. a DSP unit, a look-up table unit, an Input/Output device). These blocks live on an integrated circuit. However, the exact placement of the blocks within the integrated circuit are not known at time of placement. We wish to calculate the exact positions of the blocks. However, we do not want the blocks to be placed just anywhere. We want to optimally place the blocks, such that it satisfies some criteria. To derive the criteria for optimal placement, we must first revisit the synthesis flow of an integrated circuit. After placing the blocks on the integrated circuit, we then route communication between the user specified netlist. Typically, the user wants the integrated circuit to operate in a short amount of time, and not consume a large amount silicon. Hence, by reducing the area existing between blocks, we can (1) optimize for the operating frequency of the circuit (reducing the time duration of execution), and (2) optimize for monetary costs (less silicon will require less money). The reduction of area can be inferred through the minimization of wire length[2]. Therefore, we need another structure to place blocks on the integrated circuit according to this constraint. We name this structure 'The Placer'.

2.1 Data Structures

Therefore, the prominent structures that we wish to model for this experiment are listed:

- An Integrated Circuit
- A Block
- The Placer

[1] The interested reader can refer to the bibliography for a more formal discourse on this subject. [3] [2]
[2] We specifically use the HPWL as our constraint.

These structures were implemented in software. We used an object orientated language C++ to model these structures as classes. Specifically, we developed the following software representation: blck.cpp, ic.cpp, placer.cpp

2.1.1 blck.cpp

To represent the idea of a block in software, the block required the ability to store a relative position to an integrated circuit, and what network[3] the block would be connected to. The block was not made aware of the other blocks within the network.

To represent the position of the block, we had defined instance variables of this class:

```
float x;
float y;
```

Since we are not aware of the initial placement of the blocks, we do not initialize the position variables.

To represent the present connections to the block, we had instantiated a map structure:

```
map<int, vector<float> >net_w_expansion;
```

These blocks exist on the integrated circuit ic.cpp. Each block was able to get and set their respective position components (get_x(), get_y()), as well as add new networks to their network mapping. Various other capabilities were provided for the block structure to use, but are not of high importance to this discussion.

2.1.2 ic.cpp

The integrated circuit maintained a list of all blocks to be placed. It is intuitive to think of using a list structure to hold this list of blocks. However, a high reference count to blocks which reside at different positions of the list can be costly. Typically, if one needs to search the list for the required element of n elements, searching can take $O(n)$ time to complete. To reduce the complexity, this class utilized a map structure for its $O(1)$ time for item retrieval. Each block to be placed had an identity. This identity was used as the key for the map structure.

```
map<int, Blck> blck_map;
```

This also simplified the next design consideration: the storage of networks and the blocks that are in that network. Another map structure was used to hold this information:

```
map<int, vector<int> >nbs_map;
```

It is important to note that the amount of blocks to be placed on the integrated circuit, as well as how the blocks connect to each other are *user-defined*. These definition were stored in another structure, configholder.cpp. This held the relevant information for instantiating the blocks, and their respective behaviors.

The integrated circuit object we instantiated (which we will refer to as IC can now be passed to the placer to now place the blcks.

[3] A network is a set of blocks which are connected to each other. Here we represented the network as a clique model [4]

2.1.3 `placer.cpp`

We will now describe the main routines of the placer. We are only concerned with its routines for this discussion since it's main purpose is to modify the positions of the blocks which live on the `IC`. Specifically the placer has three main routines:

1. `place`(): In an analytical placement algorithm, we "place" blocks by solving a set of linear equations, which are quadratic representations of the half perimeter wire length (HPWL). It is important to note that the blocks specified by the user have some fixed blocks (normally Input/Output blocks). These fixed blocks are **fixed** in the integrated circuit, and cannot be moved. Therefore in this routine, we had to formulate a matrix of the equations to be solved. Specifically, we had to formulate an $n \times n$ matrix, which we stored in a variable `int * Ax`. We also had to formulate a matrix `int * b_x` and a matrix `int * b_y`. These held the known locations (x and y) of fixed blocks. The derived matrices were then passed into a spare equation solver. We utilized `SuiteSparse`[1] to solve the set of linear equations. We had to solve for both the x and y positions of the blocks, which required the equation solver to be called twice (once with `int * b_x`, and then again with `int * b_y`). Once the new x and y positions of the blocks were calculated, this routine completes upon setting the blocks with their updated positions.

2. `spread`(): The initial solution from the sparse matrix solver places all of the blocks within the center of the integrated circuit (as this minimized the HPWL). However, this placement is not acceptable as blocks will be overlapping (two blocks cannot occupy the same space). Therefore, some method of spreading the blocks apart must exist. To spread the blocks from the center, we introduce the idea of *pseudo* blocks and *pseudo* connections. Pseudo blocks are fake blocks that are introduced to the integrated circuit, which are placed at fixed location. These blocks attach to a subset of the existing non-pseudo blocks through pseudo connections. By having some notion of weight exist on the connection between pseudo blocks and the subset of blocks, we can have the analytical placement's `place`() routine be fooled of the actual circuit requirements. By introducing significant weights on the connections, we can *spread* apart the attached sub group. This routine introduces new pseudo connections by (1) splitting the current working surface into four sub-surfaces and placing a pseudo block at the center of these sub-surfaces. A sub group of blocks are assigned to one particular pseudo block. This process repeats again at each of the new sub-surfaces. To assign a sub-group to a pseudo block, we must derive a methodology to allow uniform assignment from each of the new pseudo blocks to the subgroups. This routine uses a *multinomial distribution* to assign an equal amount of the blocks to each of the four new pseudo blocks. A multinomial distribution is probabilistic distribution, and has the probability mass function:

$$Pr(X_1 = x_1, ..., X_k = x_k) = \begin{cases} \frac{n!}{x_1! \cdots x_k!} p_1^{x_1} \cdots p_k^{x_k}, & \text{when } \sum_{i=1}^{k} x_i = n \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

   By inspecting this probability distribution, it can be set to have 4 classes, of which can occur relatively equally. Concisely, we must sample from a multinomial distribution which has four classes (which represent each of the four new pseudo blocks). That is, we can set class C_1, C_2, C_3, C_4 such that initially, all classes have equal probability of being selected. We

implemented a spreading function which assigned a pseudo block to a subgroup of blocks by assigning a class to each of the blocks through the sampling of this multinomial distribution. This was our probabilistic approach to analytical placement. When a class was selected, we re-weighted the probabilities. If we selected C_1, then the probability of select C_1 is reduced, while the probability of selecting any of the other classes is increased (linearly through an additive measure). This process is repeated until all blocks are assigned to a class. Then, each of the four classes are assigned to one of the new four pseudo blocks. The weight placed on each of the subgroups is exponentially dependent on the iteration round, following the formula:

$$w_{\text{subgroup}} := w_{\text{subgroup}} * 2^{\text{i}} \tag{2.2}$$

Where w_{subgroup} was initially 4, and i, the iteration number was initially 1. After assignment, some cleanup is performed within the integrated circuit to ensure the required structures for operation are updated accordingly.

3. `snap_to_grid`(): If we find that the placement of the blocks on the integrated circuit do not overfill more than 25% of the available grid slots, then we call upon this routine. In this experiment we had the following requirements:

   - No two blocks can occupy the same grid slot
   - The final placement had to center the blocks in their respective grid slots

   Therefore, this routine checked for these requirements. Firstly, it inspected the current blocks and which grid slots they were situated in. If the slot was not overfilled, the block was centered to the grid slot. Otherwise, a breadth-first search was conducted from an overfilled region to search for the nearest open grid slot. The offending block is then placed in this open space. This routine iterates over the entire grid space until no overfilled blocks exist.

2.2 High Level Operation

To gain a high level view of the operation, we have summarized the execution of this model implemented in this experiment:

1. Read User Specified Netlist: `configholder.cpp`
2. Using the configuration data, create an instance of integrated circuit which holds the required user specified blocks and networks between blocks: `ic.cpp`
3. Provide `ic.cpp` and the `configholder.cpp` to `Placer` :: `place`(), and solve for the initial placement of the blocks.
4. While the amount of overfilled grid slots is greater than 25%
   (a) Call upon `Placer` :: `spread`()
   (b) Re-solve for the updated placements by calling `Placer` :: `place`()
5. Snap the resulting blocks to the grid (while ensuring every grid slot has only 1 block residing inside of it) by calling `Placer` :: `snap_to_grid`()
6. End Placement.

Chapter 3

Experimental Results

In order to outline the performance of this analytical placement algorithm, we ran the implementation of this algorithm against several test circuits. We recorded the half perimeter wire length at different stages of the algorithm. Specifically, we recorded the HPWL at (1) Initial Solve, labeled as Stage 1, (2) First Spread, labeled as Stage 2, (3) Pre-snap, labeled as Stage 3, and (4) Snap, labeled as Stage 4. The results are summarized in Table 3. I have also included the results for the circuit file `cct1`

Circuit File	Stage 1	Stage 2	Stage 3	Stage 4
`cct1`	106.55	115.56	173.63	165.00
`cct2`	693.167	831.36	1613.18	1674.00
`cct3`	2510.30	3909.01	11384.88	12540
`cct4`	7159.67	11521.59	35850.61	40034

Table 3.1: The Half Perimeter Wire Length at Stage 1, Stage 2, and Stage 3 of the Analytical Placer

It is warranted to investigate the percent change $\Delta_1\%$ between Stages 2 and 3, and between Stages 3 and 4. By measuring the percent change in regards to Stage 2 and Stage 3, we can identify how the HPWL grows with every subsequent iteration. The percent change in Stage 3 and Stage 4 $\Delta_2\%$ can identify how well our `snap_to_grid`() function processes, in regards to its pre-snapped half perimeter wire length.

Circuit File	$\Delta_1\%$	$\Delta_2\%$
`cct1`	50.25	-4.97
`cct2`	94.04	3.77
`cct3`	191.25	10.15
`cct4`	211.16	11.67

Table 3.2: The Half Perimeter Wire Length at Stage 1, Stage 2, and Stage 3 of the Analytical Placer

When generating the pseudo blocks, different weights were used between the connected subgroups as iterations increased. Experimentation was necessary to gather a "good-enough" weight

distribution for each connected sub-group. Referring to Table 3 a comparison of HPWL was presented while changing the weight distribution per subgroup.

Initial Weight	Stage 3	Stage 4
2	34517.3	39851
3	35850.6	40034
...	...	...
10	37945.6	40175

Table 3.3: Experimentation of Weight Modification, results from Stage 3 and Stage 4 of `cct4`

As noted above, as the weight was increased incrementally, there was a significant increase in the HPWL recorded in Stages 3 and 4. This was tallied here for the circuit `cct4`, yet other circuit files had the same effect. It would appear that the initial weight to choose should be 2, however we chose the initial weight to be 3. This was chosen as this performed better on smaller circuit files. We also seeded the pseudo random number generator (used by the multinomial distribution) to reproduce the same environment. The weighting of new pseudo nets is done through heuristic process. Therefore, it is difficult to find an optimal weighting scheme.

This concludes the results of this experiment. Images from the experimental results are included in the appendix.

Chapter 4

Appendix

4.1 Initial Solve

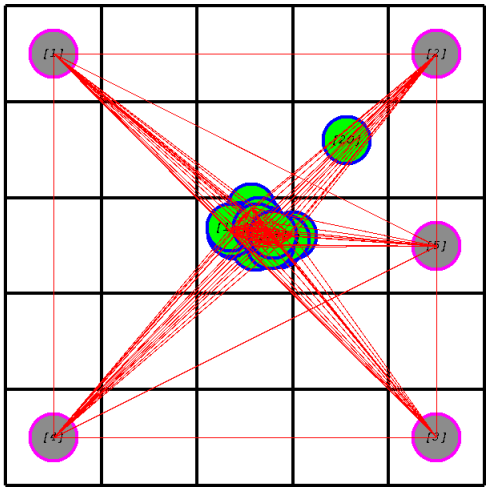

Figure 4.1: The Initial Solve of `cct1`

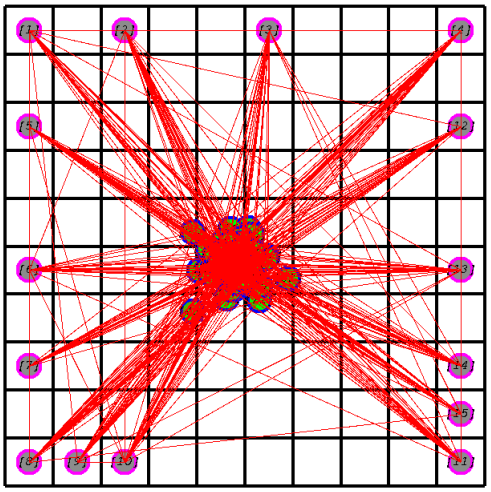

Figure 4.2: The Initial Solve of `cct2`

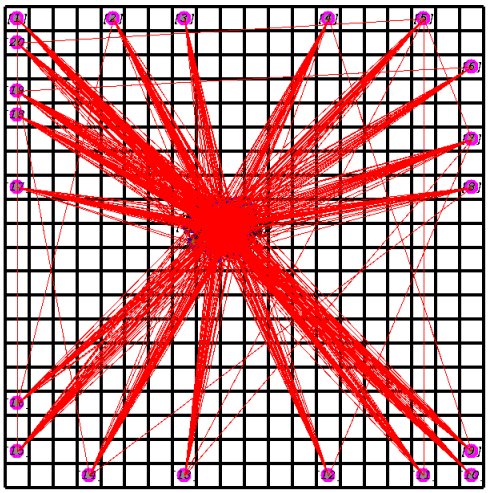

Figure 4.3: The Initial Solve of `cct3`

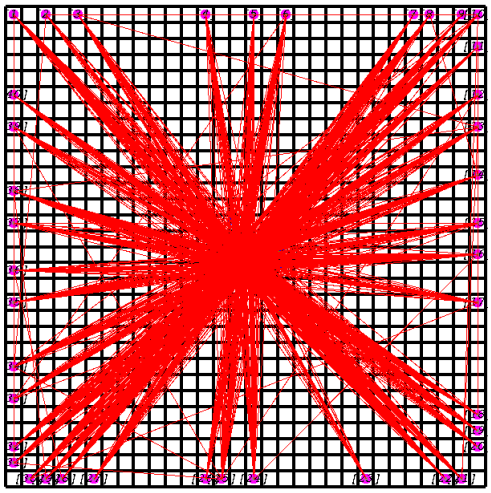

Figure 4.4: The Initial Solve of `cct4`

4.2 Initial Spread

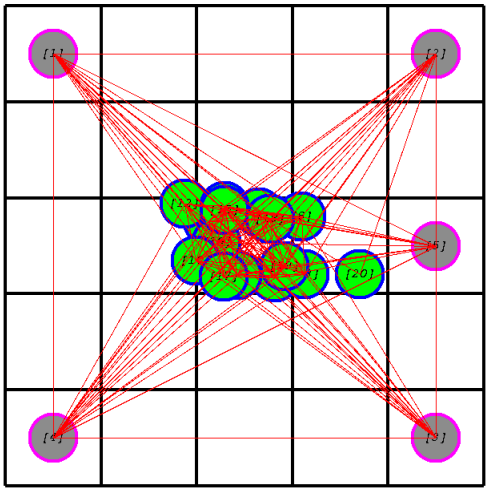

Figure 4.5: The Initial Spread of `cct1`

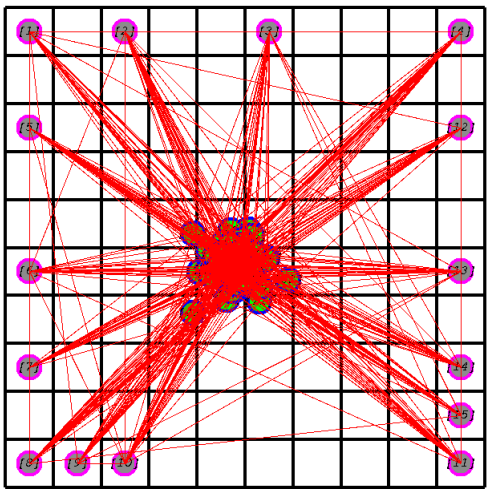

Figure 4.6: The Initial Spread of `cct2`

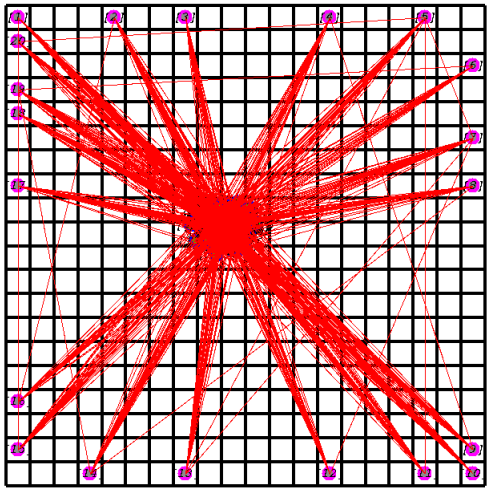

Figure 4.7: The Initial Spread of `cct3`

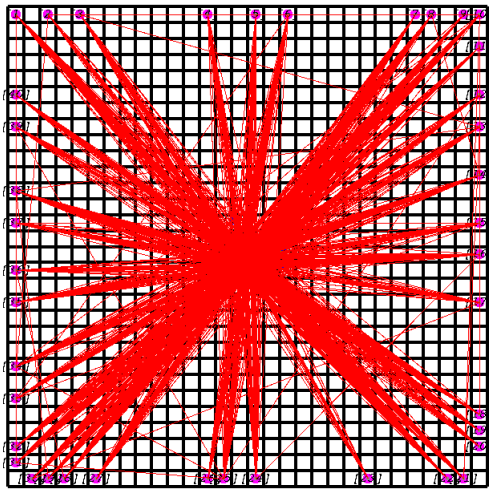

Figure 4.8: The Initial Spread of `cct4`

4.3 Pre-Snap

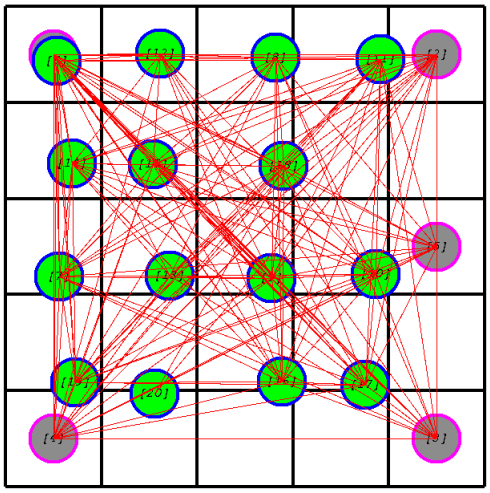

Figure 4.9: `cct1` at 25% fill

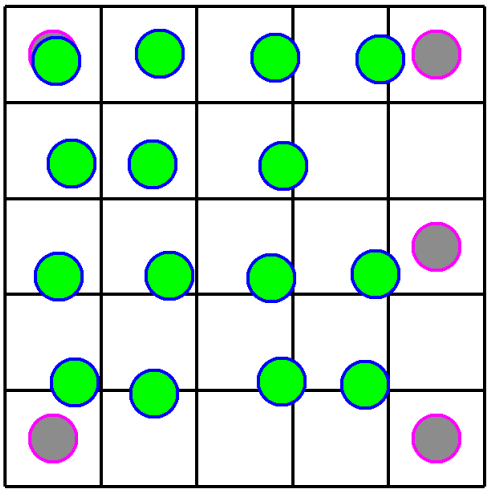

Figure 4.10: `cct1` at 25% fill, clique representation removed

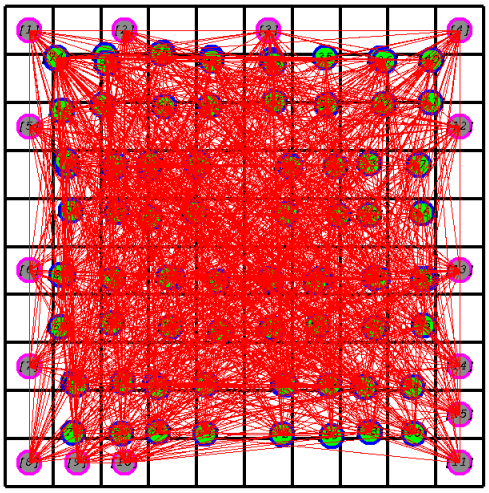

Figure 4.11: `cct2` at 25% fill

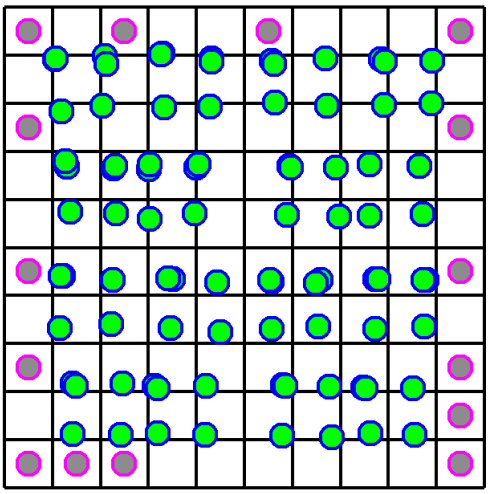

Figure 4.12: `cct2` at 25% fill, clique representation removed

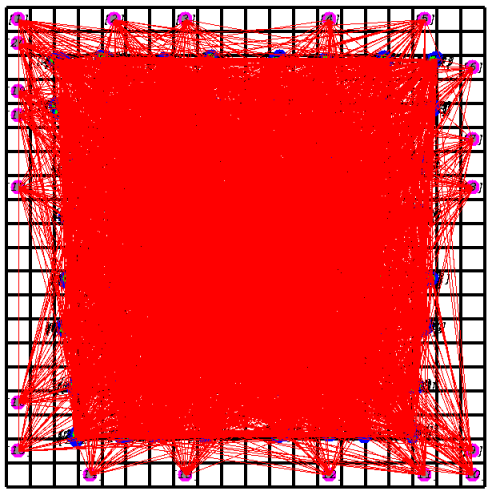

Figure 4.13: `cct3` at 25% fill

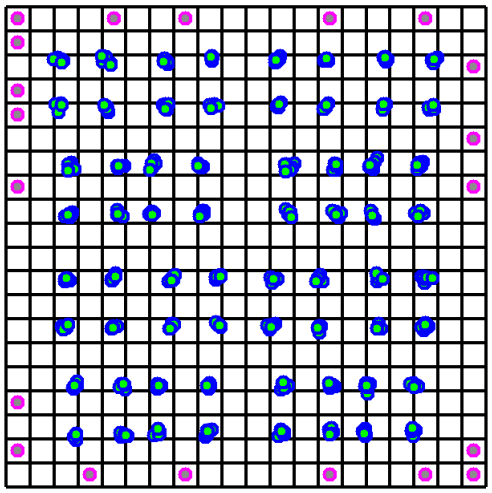

Figure 4.14: `cct3` at 25% fill, clique representation removed

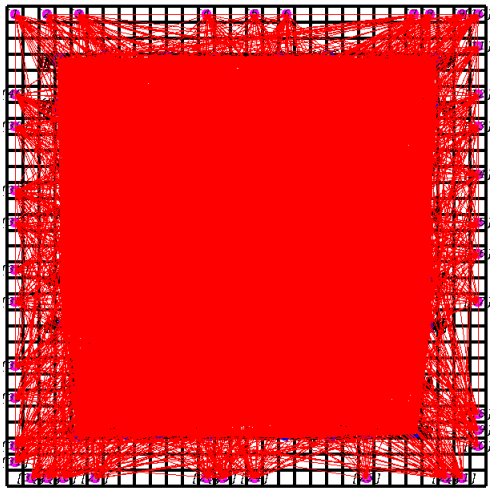

Figure 4.15: `cct4` at 25% fill

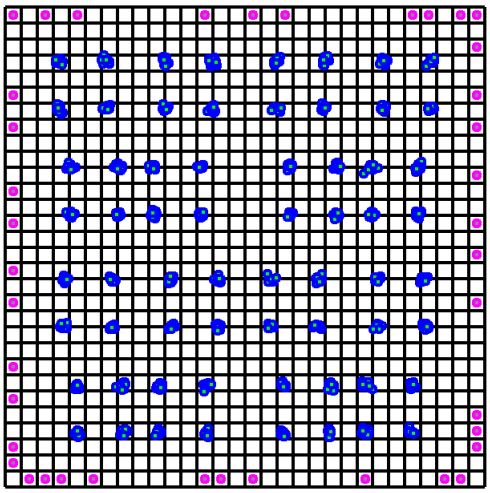

Figure 4.16: `cct4` at 25% fill, clique representation removed

4.4 Snapped To Grid

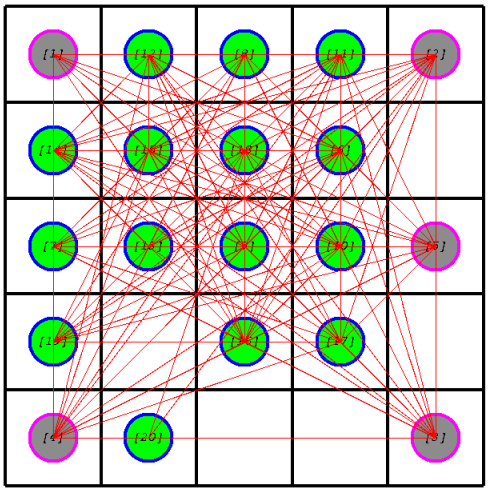

Figure 4.17: `cct1` snapped to grid

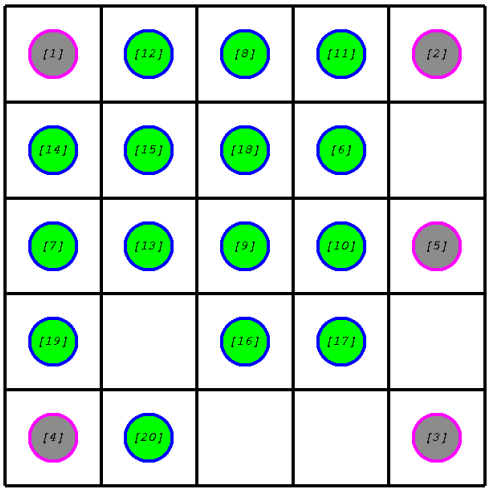

Figure 4.18: `cct1` snapped to grid, clique representation removed

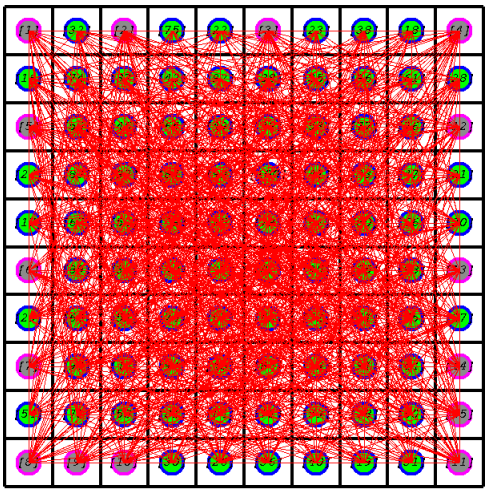

Figure 4.19: `cct2` snapped to grid

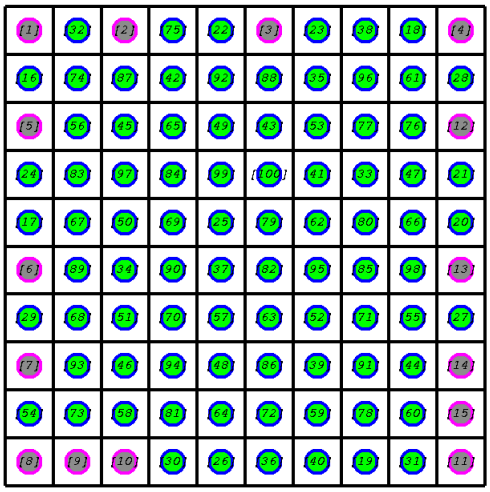

Figure 4.20: `cct2` snapped to grid, clique representation removed

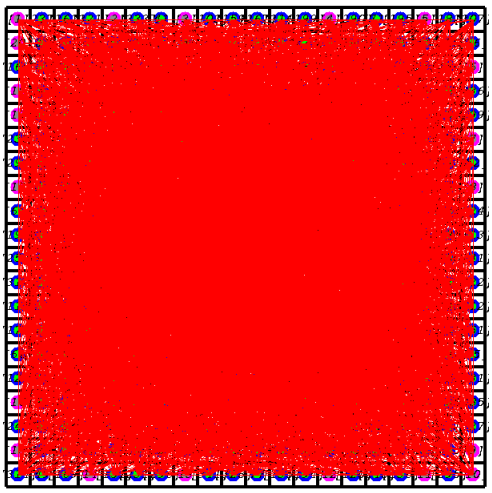

Figure 4.21: `cct3` snapped to grid

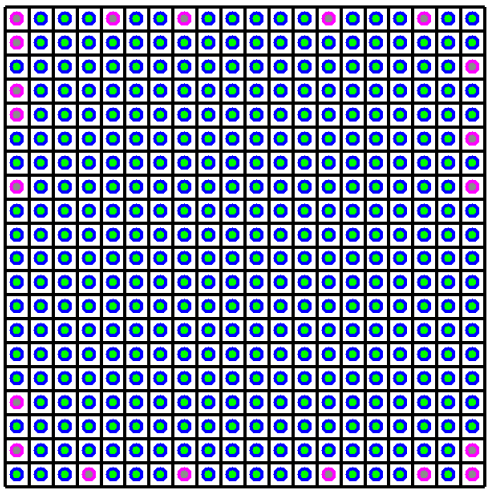

Figure 4.22: `cct3` snapped to grid, clique representation removed

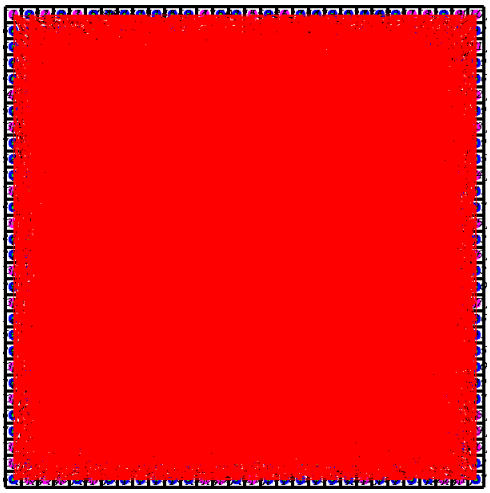

Figure 4.23: `cct4` snapped to grid

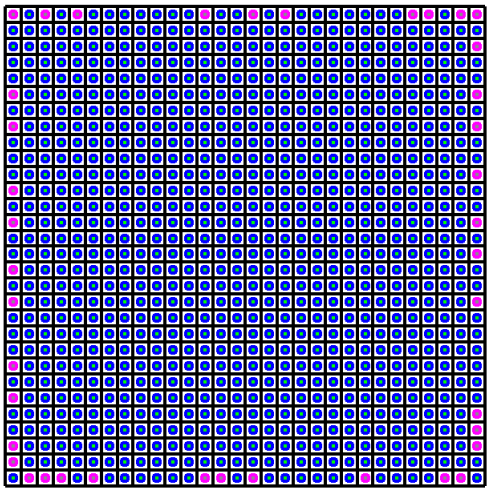

Figure 4.24: `cct4` snapped to grid, clique representation removed

Bibliography

[1] Tim Davis, WW Hager, and IS Duff. Suitesparse. *htt p://faculty. cse. tamu. edu/davis/suitesparse. html*, 2014.

[2] Andrew A Kennings and Igor L Markov. Analytical minimization of half-perimeter wirelength. In *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, pages 179–184. ACM, 2000.

[3] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[4] Fred S Roberts and Joel H Spencer. A characterization of clique graphs. *Journal of Combinatorial Theory, Series B*, 10(2):102–108, 1971.

[5] Georg Sigl, Konrad Doll, and Frank M Johannes. Analytical placement: A linear or a quadratic objective function? In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 427–432. ACM, 1991.