**ECE1387: Assignment #2**

**Part1:**

**Initial Placement Plots:**

Connections can be seen by red lines while cells are shown as blue text. The grid is represented by black lines.

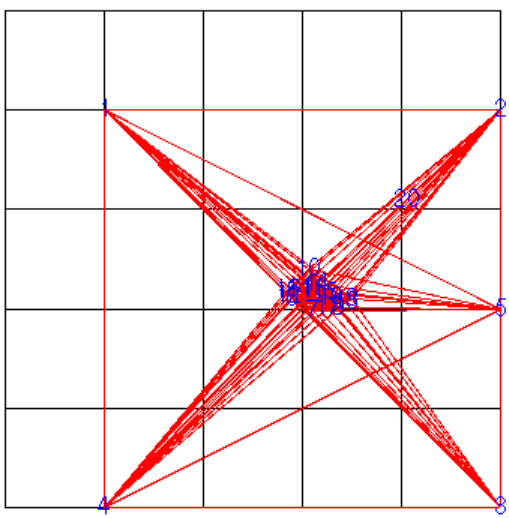


Figure 1: CCT1

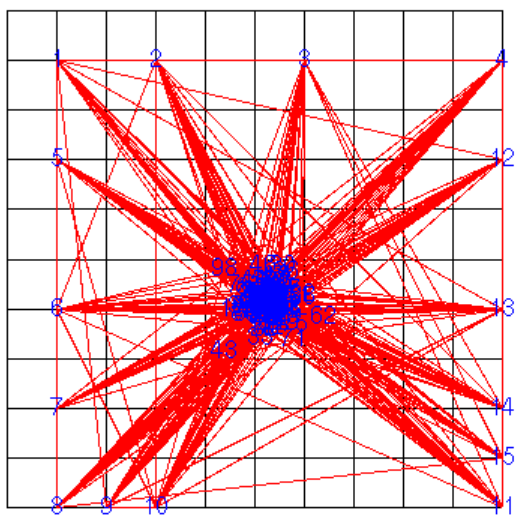


Figure 2: CCT2

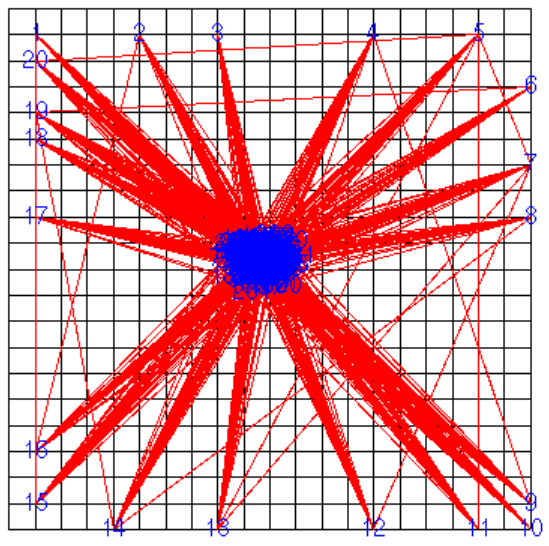


Figure 3: CCT3

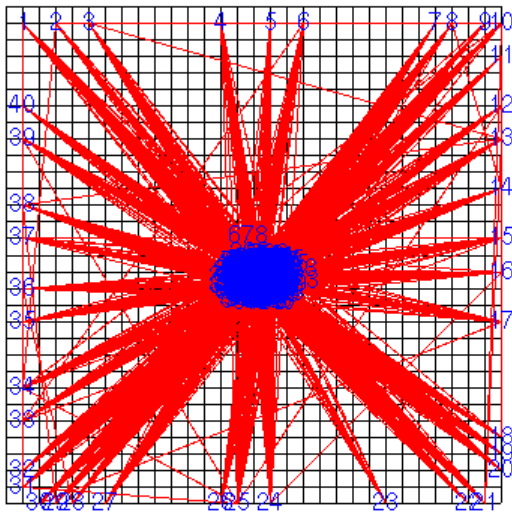


Figure 4: CCT4

**Half-Perimeter Bounding Box Wire Length (Table 1) (WL is in units of the grid)**

|  |  |
| --- | --- |
| **Circuit** | **HPBB WL [units]** |
| **CCT1** | 106.556 |
| **CCT2** | 693.167 |
| **CCT3** | 2390.31 |
| **CCT4** | 7099.67 |

**Part2:**

**Simple overlap removal plots:**

For the following set of plots, the green text represents the artificial blocks that are introduced, while the green lines represent the artificial connections. Please note that the plots and wirelengths shown are calculated with weights multiplied by a factor of 30.

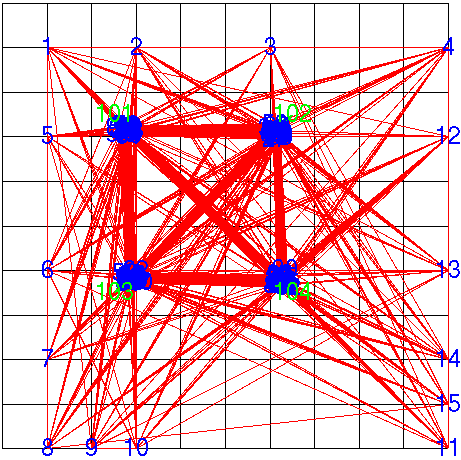


Figure 5: CCT2 with simple spreading

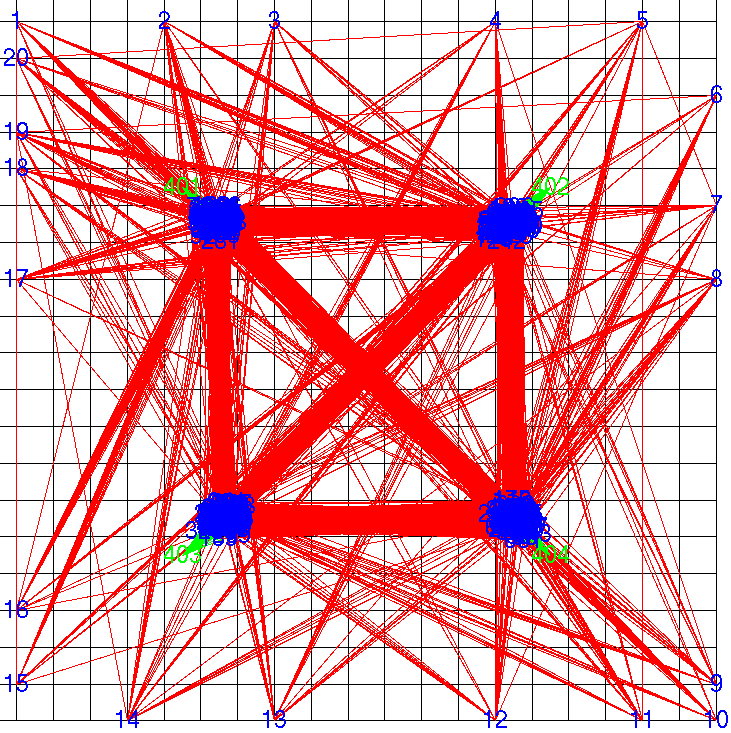


Figure 6: CCT3 with simple spreading

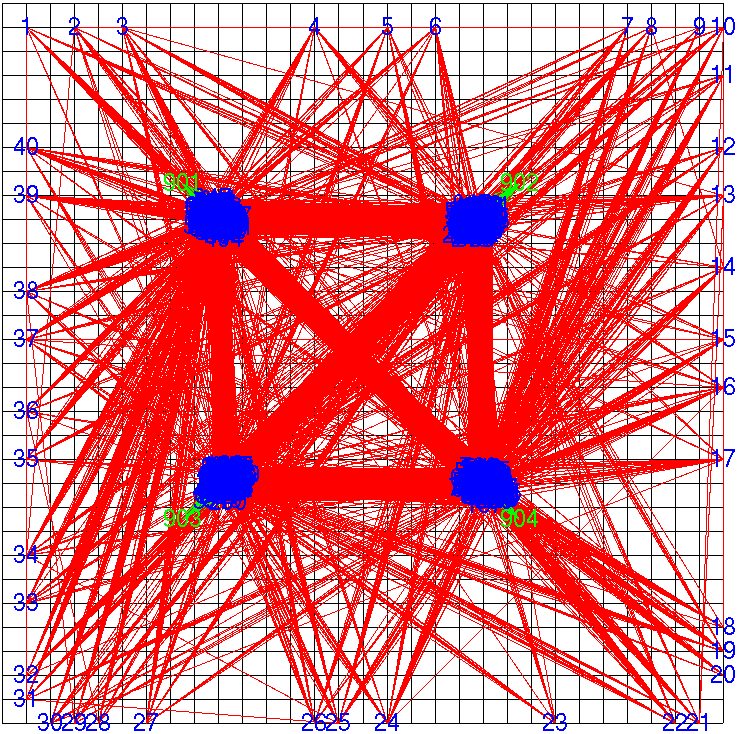


Figure 7: CCT4 with simple spreading

**Bounding Box Wire Length (Table 2):**

|  |  |
| --- | --- |
| **Circuit** | **BB WL (after introducing fixed blocks)** |
| **CCT2** | 1146.76 |
| **CCT3** | 7382.08 |
| **CCT4** | 21896.7 |

Regarding the partitioning of the blocks, I simply create a list (one dimensional) of block pointers that point to a block object that represents each block in the circuit (excluding fixed blocks). I then sort this list based on vertical position (in ascending order), after this I sort the same list based on horizontal position (ascending order as well). By this point, blocks closer to the top left of the grid will be in the first part of the list, followed by top right, then followed by the bottom left and finally the bottom right. The next step in the algorithm is to find the indices representing quarter intervals of the list (the first quarter, then half and finally three fourths). Utilizing these indecencies, I create four sub-lists that represent each quadrant of the grid, these sub-lists represent how the blocks are partitioned.

For WL we can refer to Table 2 (with new fixed blocks) and compare the results to Table 1. As we can see the WL has increased for all three circuits. What is interesting to note is that the delta increase in WL increases as the number of moveable cells is increased. This makes sense as the connections between the movable cells/blocks will have to increase as they are spread further apart.

The data generated in Table 2 as well as the plots shown are based on the weights for the artificial nets using a factor of 30 (to arrive at the final spread solution faster). As an example of the impact of the weight, I slowly decreased the factor used in the weight calculation from 30 down to 5 and noticed that the WL decreased (see Figure 8 for the solved placement of CCT4), down from 21896.7 to 13955.4 units. If I further decrease the weight factor down to 1, then the WL is only 8703.5 - slightly more than the initial solve. If I increase the weight factor to 40 then the WL increases to 22762.4 units. I continued to increase the weight factor, but at about 100000 the WL seems to saturate at around 26000 units. Thus, the WL is proportional to the weight applied to the artificial nets.

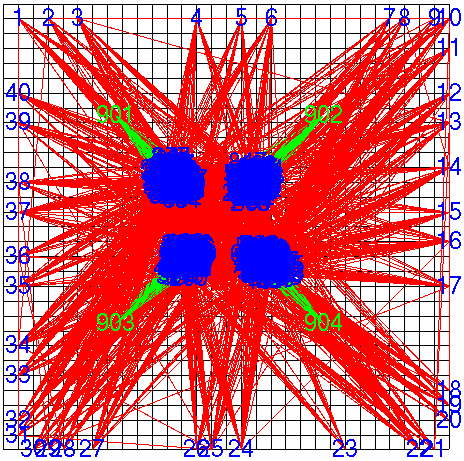
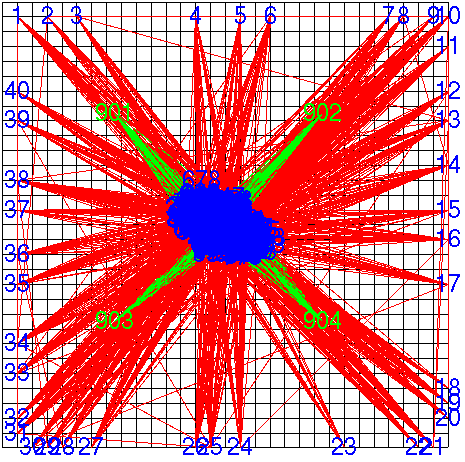
Figure 8: Simple spread of CCT4 with weight factor of 5 

Figure : Simple spread CCT4 with weight factor 1

**Part3:**

Unfortunately, I was not able to get recursion to work to a level of spreading of less than 15% overlapping. My max depth of recursion was four levels, after which I ran into a segmentation fault. There is an issue with how I am placing the virtual blocks as the level of recursion increases, that I was unable to resolve. Ideally there should be one virtual block placed in each of the four sub-quadrants, as can be seen from the plots – this isn’t the case.

However, I can still make observations about the WLs (summarized in Table 3). When comparing the WLs from a recursive spread to those from the simple spread (Table 2), we can see that the WLs are shorter. This follows intuition as the moveable blocks will be more optimally placed in comparison to a simple spread, as the distance to the fixed blocks on the periphery is minimized without increasing the distance between the moveable blocks to the extent seen in the simple overlap removal.

**Plots for recursive spreading:**

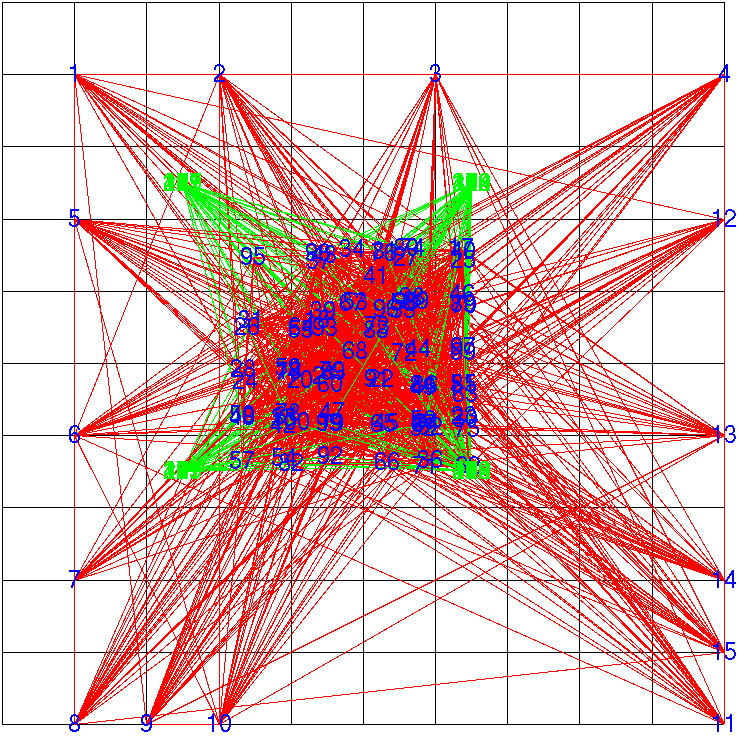


Figure 10: Recursive spreading for CCT2

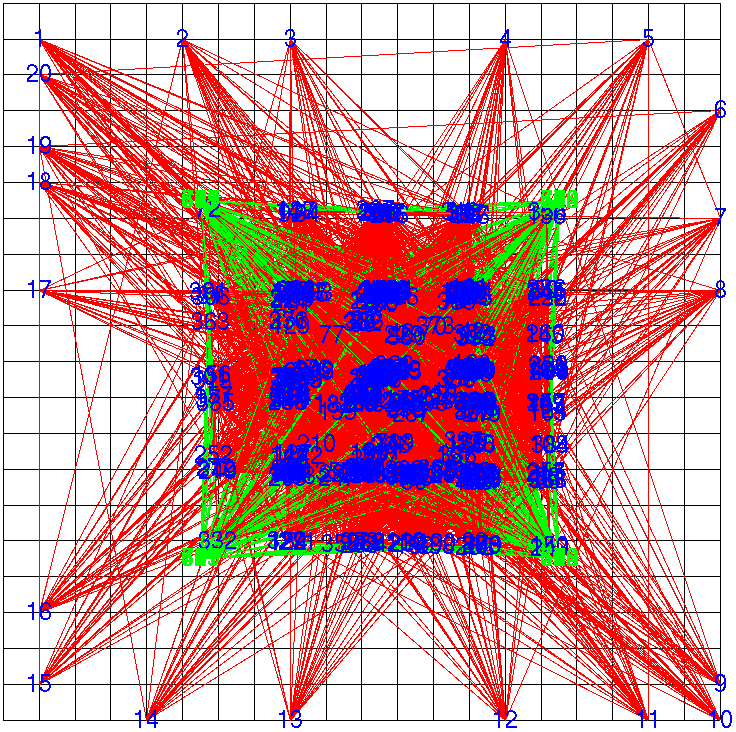


Figure 11: CCT3 recursive spreading

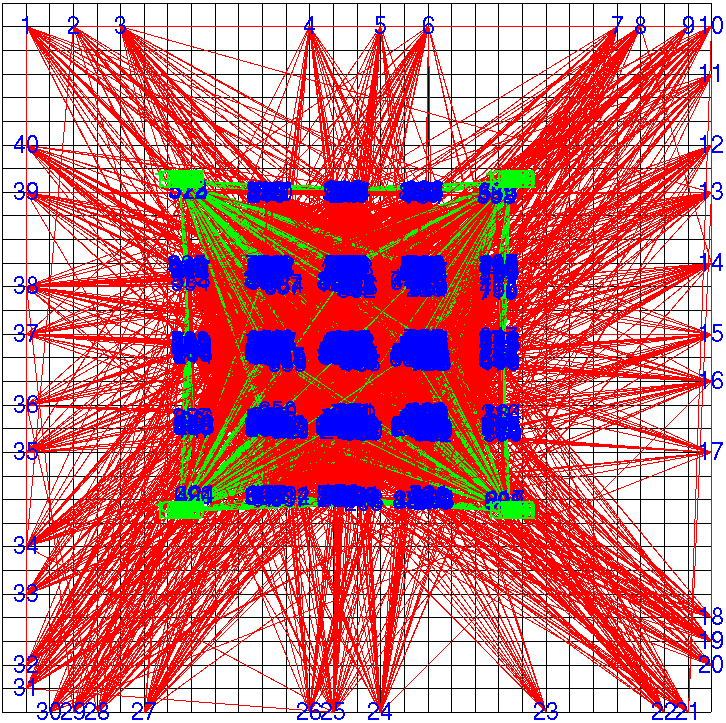


Figure 12: CCT4 recursive spreading

**Bounding Box Wire Length (Table 3):**

|  |  |
| --- | --- |
| **Circuit** | **BB WL** |
| **CCT2** | 962.353 |
| **CCT3** | 5908.54 |
| **CCT4** | 17543.7 |

**Part4:**

As mentioned in part 3, I was only able to get recursive spreading to partially work. So, I do not have data based on snapping a placement with one cell per slot. But my educated guess would be that the increase in WL would depend upon the ratio of moveable cells to fixed cells. For circuits with fewer moveable cells such as CCT2, we shouldn’t see as much of an increase. For CCT 4 we should see more of an increase. This is because during the process of snapping to the grid, the distance between the moveable blocks and the fixed blocks on the periphery should decrease, but the distance between the movable cells should increase as the blocks would be spaced further apart.

**Software Flow:**

The placer implemented is a variation of the basic placement algorithm described in class. The main routine is in the main() function within main.cpp. This program starts by parsing the input arguments from the command line, the first argument should be the circuit description file; while the second argument (which is optional) indicates whether to use a recursive approach to spreading. Next the circuit description is taken in by the parseInputFile() function and stored in the commonvars namespace which is used to pass data throughout the program. The commonvars has an allblocks member which is a list of pointers to block objects. This is where the blocks of each circuit are stored. Using the block class, we can keep track of the location of each block, whether it is movable or fixed, and whether it is artificial or not. There are also methods in the block class to keep track of connections between blocks. A Net class is used to keep track of the nets and their associated blocks, I also have methods implemented in this class to help calculate HPWL, which simplifies keeping track of the wire usage.

The major steps in placement are accomplished by separate functions in the program. There is an initialPlace() function that takes in the list of pointers to block objects in the circuit and then proceeds to formulate and solve the linear system for placement. In this function we iterate over the blocks and fill in the A and b matrices (vectors) and then leverage functions from UMFPACK to solve for the coordinates. Although the function is called intialPlace, it will be re-used whenever a new solution needs to be generated.

The other major step is to spread the movable blocks, and based on the command line argument this can either be a simple spread or a recursive one. For the simple spread the simpleSpreading() function is called. As described in Part2 of this report, the program iterates over the list of blocks and creates four sub-lists of pointers to blocks representing the four quadrants in the grid. Next an artificial fixed block is inserted and is associated with the movable blocks within the same quadrant using the addConnection method from the block class. Once the blocklist has been updated, we leverage the initialPlace() function to solve for the location of the movable blocks. If the recursive approach is selected, then the function simpleSpreadingRecursive() is called. This function is nearly identical to the simpleSpreading(), with the exception of a block list being passed in as an argument, along with a recursion depth level count. These arguments allow the function to call itself with each of the four sub-lists of blocks as inputs. The limit of recursive calls is based on the count, which is checked at the end of the function. This ideally should be based on the ratio of overfilled blocks to total number of blocks, unfortunately I wasn’t able to implement this correctly. Once the block list has been updated, intialPlace() can be called.

Once placement of the blocks has been determined all that is left is to draw the blocks along with the connections within the grid. This is accomplished by the drawscreen() function. The drawscreen() function utilizes drawline() to create the grid and visualize the connections and drawtext() to simulate the block placement. This draw function gets triggered in the main function by an event, such as clicking the proceed button within the GUI.

On a final note, I wasn’t able to implement the snap to grid functionality required in part 4 since there were issues with recursive spreading. However, the approach I would take is to iterate over the bins in the grid and then adjust the position of any blocks near that bin to snap it in place, provided the bin wasn’t full. To aid in this I have a binIndex member in the block class that can help keep track of the mapping.