ECE 1387 - CAD for Digital Circuit Synthesis and Layout

Assignment #2 – Analytical Placement with Heterogeneous Cell Types, Spreading, Snapping/Legalizing to Grid

Austin Liolli
1005641986

Program Flow

My program consists of 2 main algorithms, which are separated by the initial position and the recursive calls. All data pertaining to block location and net containment is stored in vector format. First, the file is parsed and stored as two separate 2-dimensional vectors, one vector stores the block numbers and the nets it is attached to, the other stores the fixed block numbers and their positions. The program begins by establishing a netlist, which each row of a 2D vector represents the net, and the contents in that row are the blocks attached to that net. This list of nets is used to create the clique weights, the number of vertices in the clique is used in the diagonal matrix definition.

Once all of these vectors are obtained, the Q matrix is solved. The size of the matrix is the difference between the size of the fixedblock vector and the block number vector. This will solve all the non-fixed block locations. Each positions value is calculated by its location, either on a diagonal of the matrix or non-diagonal elements of the matrix. If the position is on the matrix diagonal, the sum of the weights attached to that node are needed. The netlist file is checked by storing a vector of all nets attached to that block. Each net attached is summed for total weight on the node (clique model) and the position is added. If the position is not on the diagonal, then the sum of the weights between that node and its respective position are compared. Two vectors are obtained, one for the block we are in and one for the compare block, if any nets found in these vectors match, one of those weights is added. Next, the B matrix for both X and Y are solved. Similar to the diagonal matrix position, the nets between any fixed blocks and that respective position are checked using two comparative vectors. If there are connections found, that edge weight is multiplied by the respective position the fixed block is at and placed in the matrix.

The next function is to properly calculate the arrays needed to pass to the UMFPACK. This is done by creating 3 arrays which store the position of each first row value, the position that row value is found and the first column position. These arrays are then passed to the linear solver with the result of the x positions and y positions. Using these positions, the first question is completed and the half perimeter wire length is computed.

The second question is executed very similar to the first, and same with the recursion. After the initial split, the above algorithm is repeated for each block set to solve the updated positions. To determine which blocks will be pulled toward each quadrant, the median position in each x and y are determined from that block set (all blocks on first iteration). The blocks are separated by the section which they are closest to. The central position for each split is calculated and a similar algorithm, with smaller vectors and matrices is ran to now compute the updated positions. The highest level positions are updated once this algorithm is ran.

This section is repeated recursively by creating a queue of vectors from the previous iteration. By keeping track of which blocks were split into which quadrant of each iteration along with their respective centers, the split algorithm, which does not take fixed input, can be run again. Depending on the iteration, ie. first split has 4 vectors, second split has 16, the appropriate amount of vectors are popped off the queue to use. As the new vectors are computed they are pushed to the back of the queue to be later used. At the end of each iteration the positions are updated, showing the recursive split for each section.

For normal test cases, the splitting is limited to a position where the solved central position of the next iteration will not be within the first column of blocks. This means that splitting will not occur in very small regions where multiple blocks can be placed in a single location. This is determined by the column size / (4*iteration). At this point the program will seize to allow cuts and the design can be snapped into place.

Due to time constraints only the first snapping technique was implemented. This is done by ignoring the fixed blocks, as they are already in position, and iterating through the remaining unfixed blocks. Each position of the grid is compared to all of the nodes, and the minimum distance needed to travel from the center of the grid location to that node is the snap node needed. In the positions vector this value is now multiplied by -1, and only positions with positive locations are examined. At the completion of the algorithm, each block shall be located in a specific position for the grid. An issue with the created algorithm for the 4th test case is that when the spreading is run past 4 iterations the snapping does not recognize 4 nodes. For this reason this specific case is limited to this many iterations whereas the other two cases are not limited. With more time to complete, a debug session of where the missing position snaps fell would be found. For the purpose of completion this is limited.

The final output of the program shows the resulting positions of each iteration. The graphics will load with the first question of the respective file selected. (click proceed on the right hand side to initialize after the window pops up). For each case, the iterations are limited. By clicking on the “Spread” button, the next iteration of spreading can be displayed. Once the button is no longer active and the stopping criteria has been met, the snap button will lock the position of the nodes. The snapping can be done at any point in the program after the first iteration. Returned to the console is the HPBBWL of each iteration. At any point, the iteration being viewed can be viewed with or without nets by toggling the “Nets: On” button on the side.

<u>Results</u>

Question 1:

File	Half-Perimeter BBWL
cct1	99.8298
cct2	707.656
cct3	2536.62
cct4	7342.21

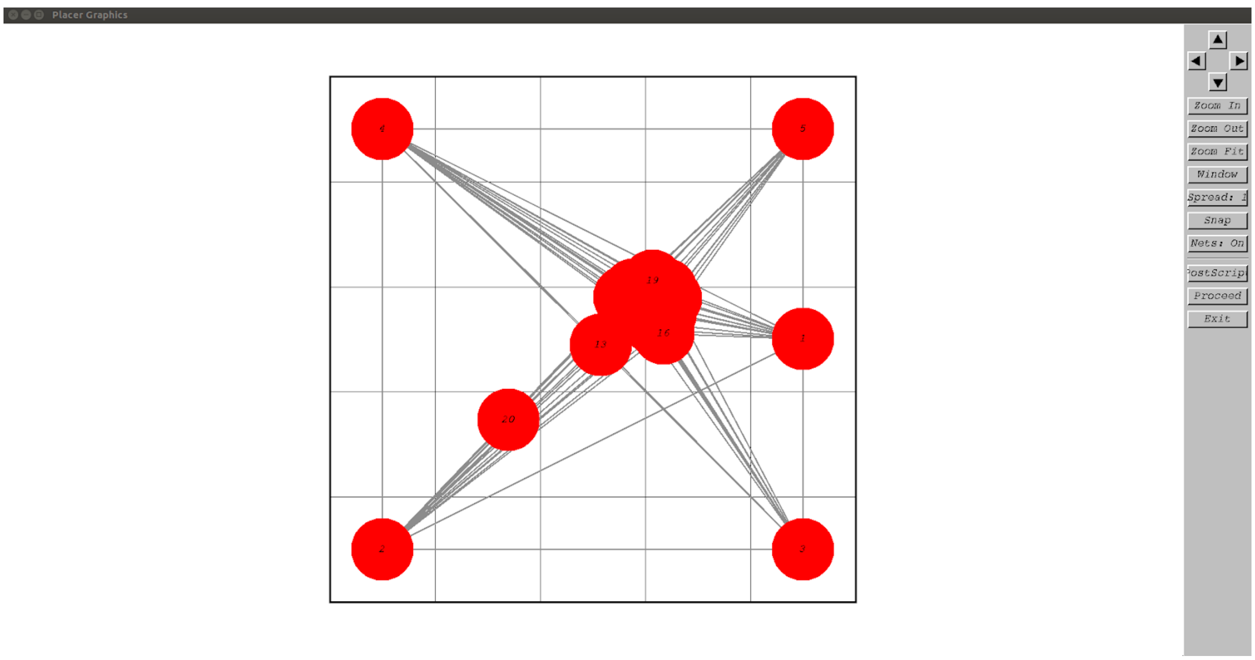

Figure 1: cct1 Initial Placement

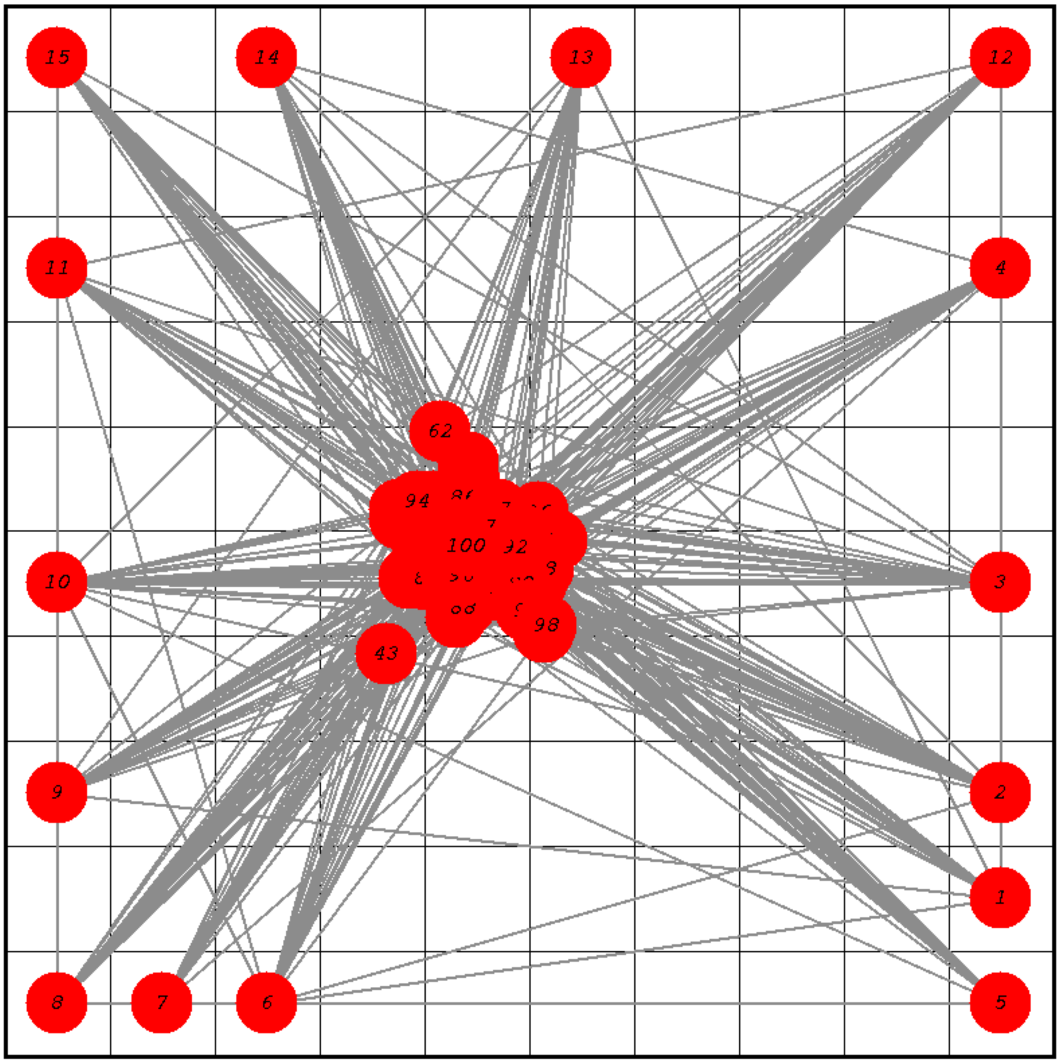

Figure 2: cct2 Initial Placement

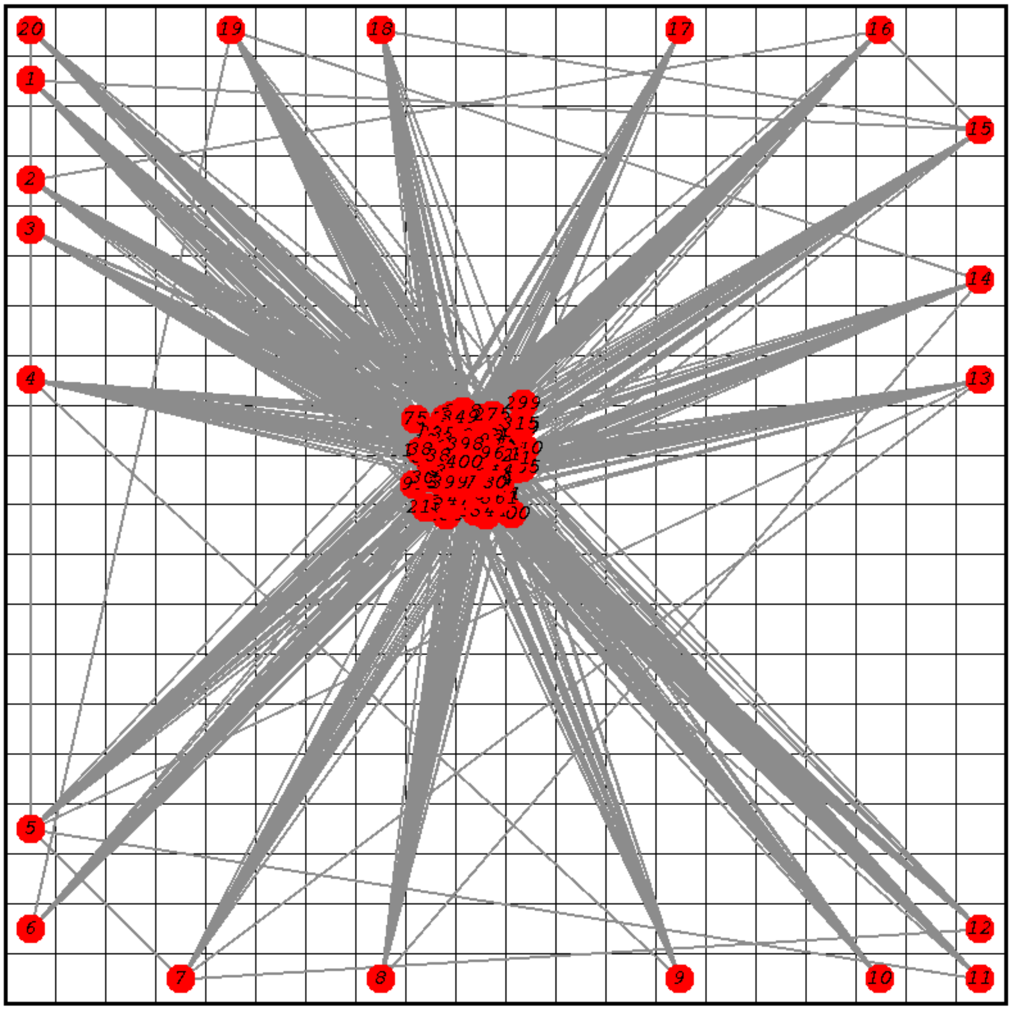

Figure 3: cct3 Initial Placement

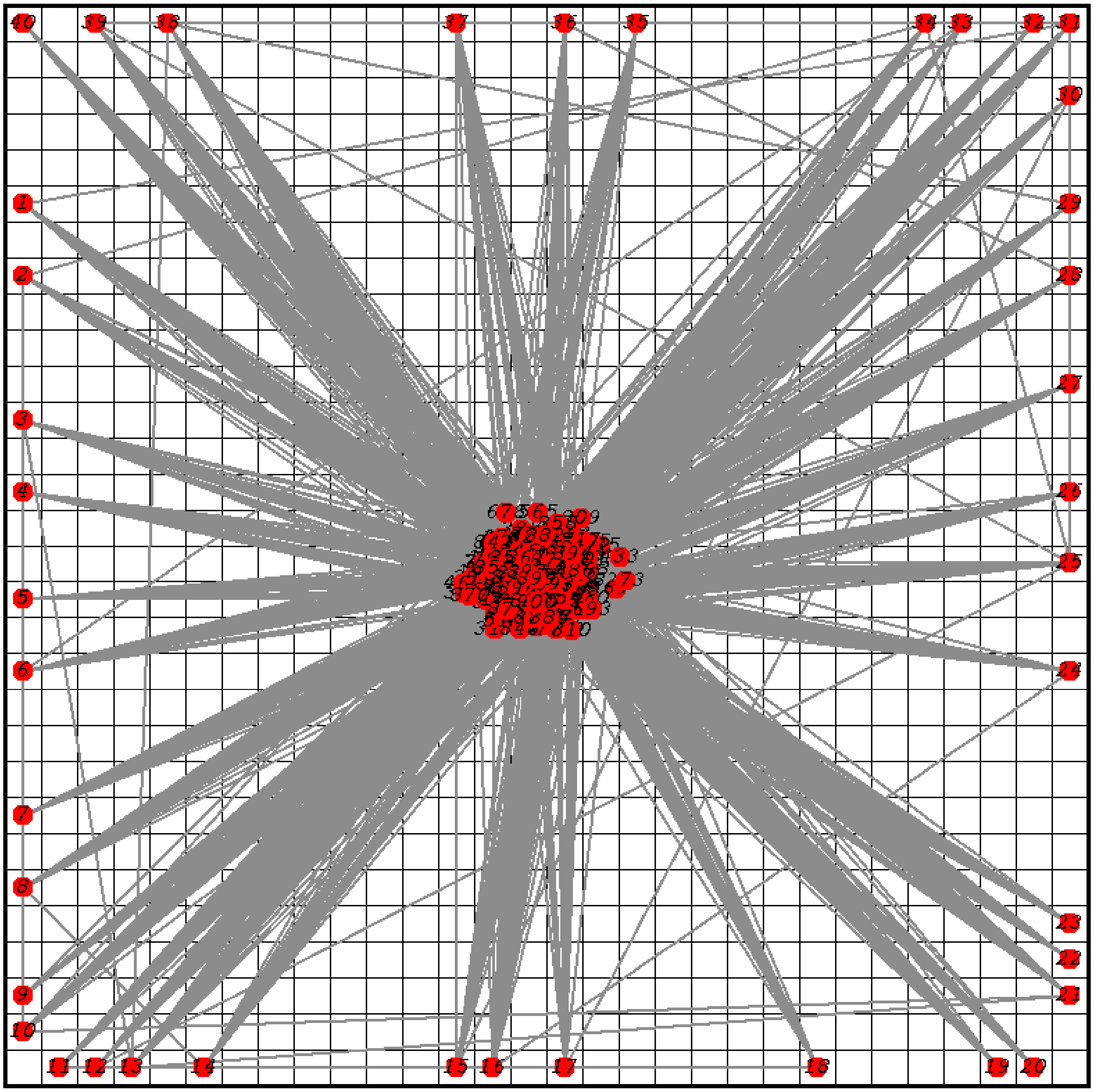

Figure 4: cct4 Initial Placement

Question 2:

With each new fixed block added, the wire length is increasing. This is because the initial position solved in question 1 is most optimal, regardless of actual location on the grid. As these fixed blocks are added, the nodes are pulled further from their optimal positions, meaning that with a larger weight, the farther from optimal they are the larger the wire length will be.

File	**W=0.5**	**W=1**	**W=2**	**W=5**	**W=10**	**W=20**
cct2	937.16	981.1	1080.38	1240.07	1330.33	1392.93
cct3	5493.37	6597.3	7703.63	8716.85	9108.05	9263.61
cct4	17050.7	20690.1	24164.4	27248.5	28401.3	28921.2

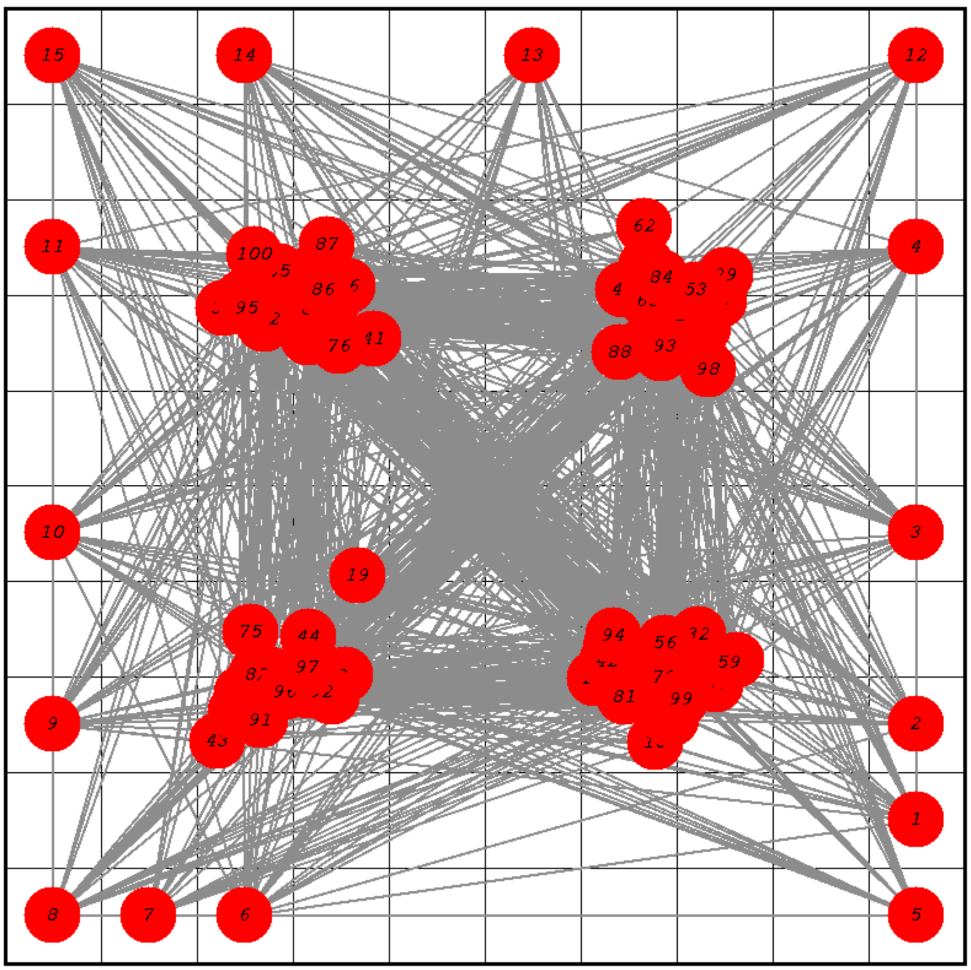

Figure 5: cct2 First Spread (W=10)

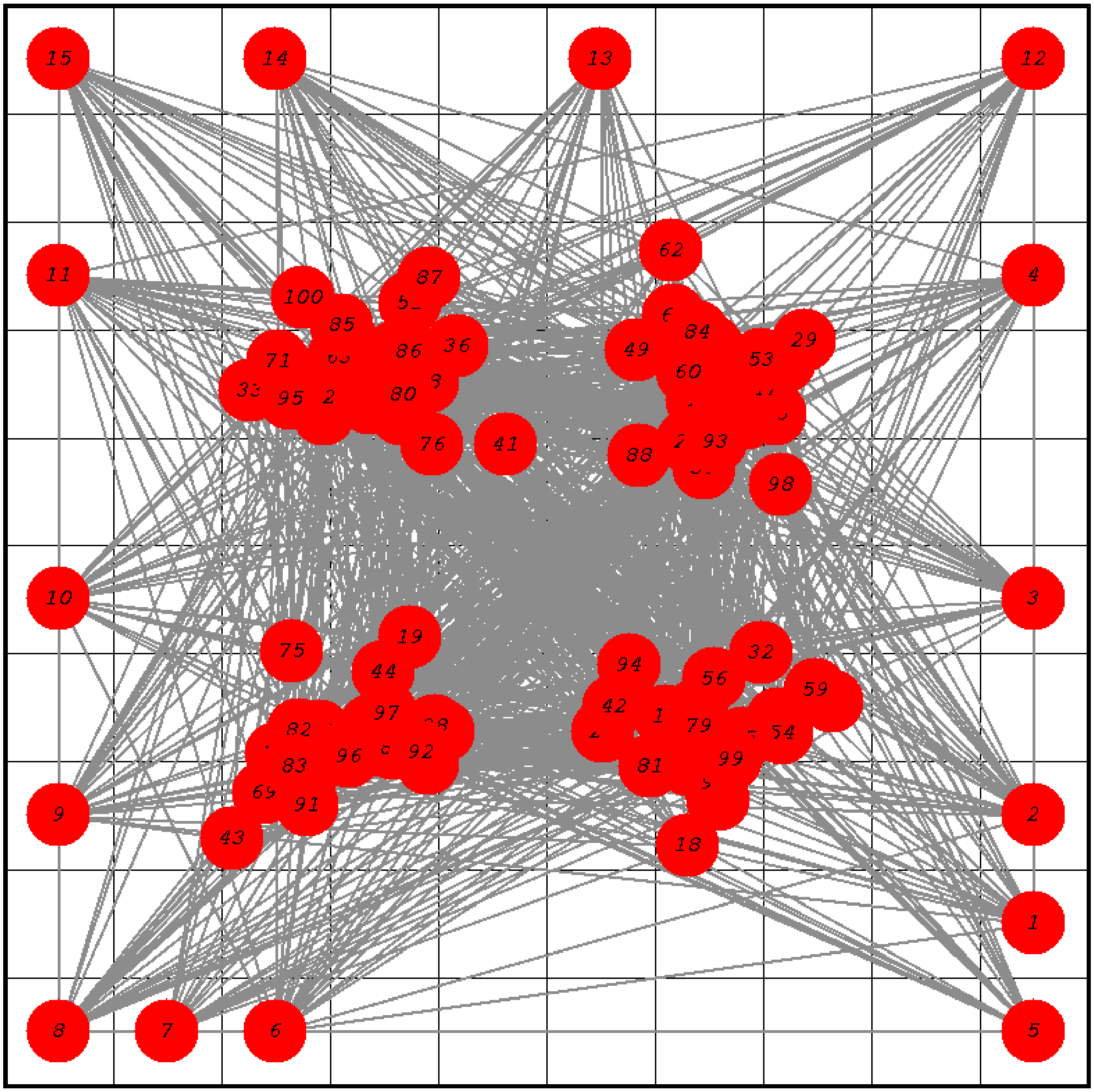

Figure 6: cct2 First Spread (W=5)

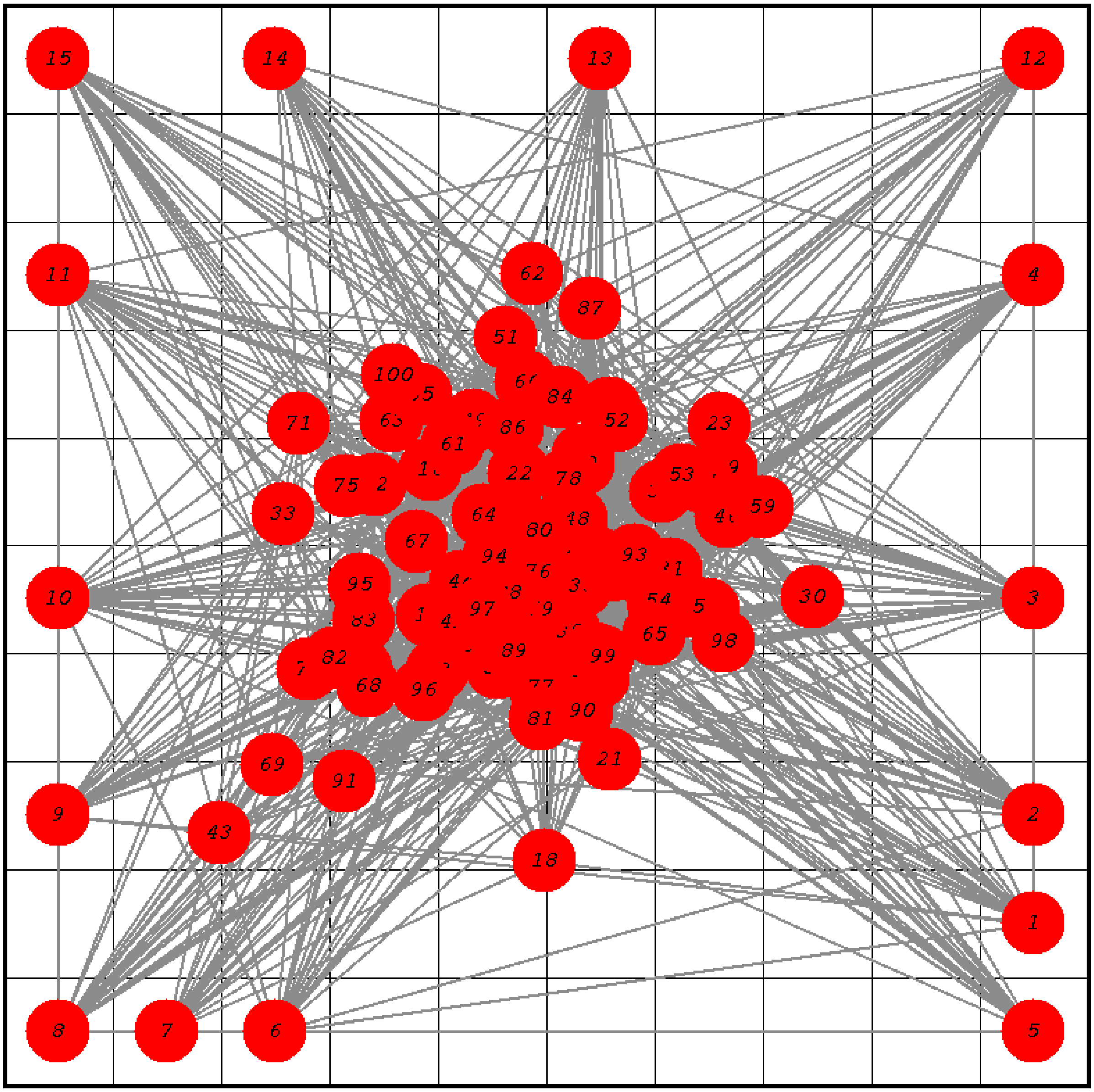

Figure 7: cct2 First Spread (W=1)

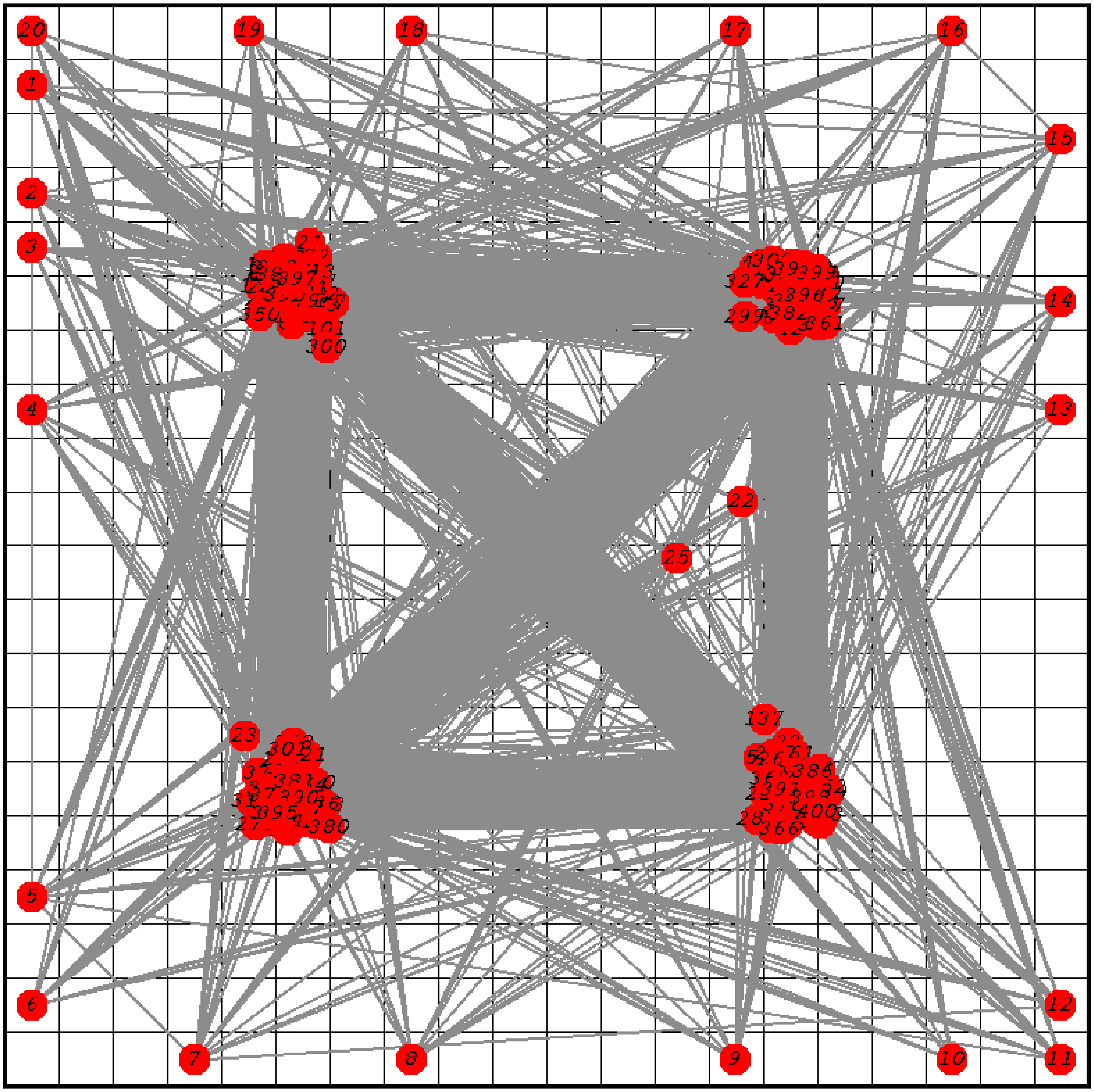

Figure 8: cct3 First Spread

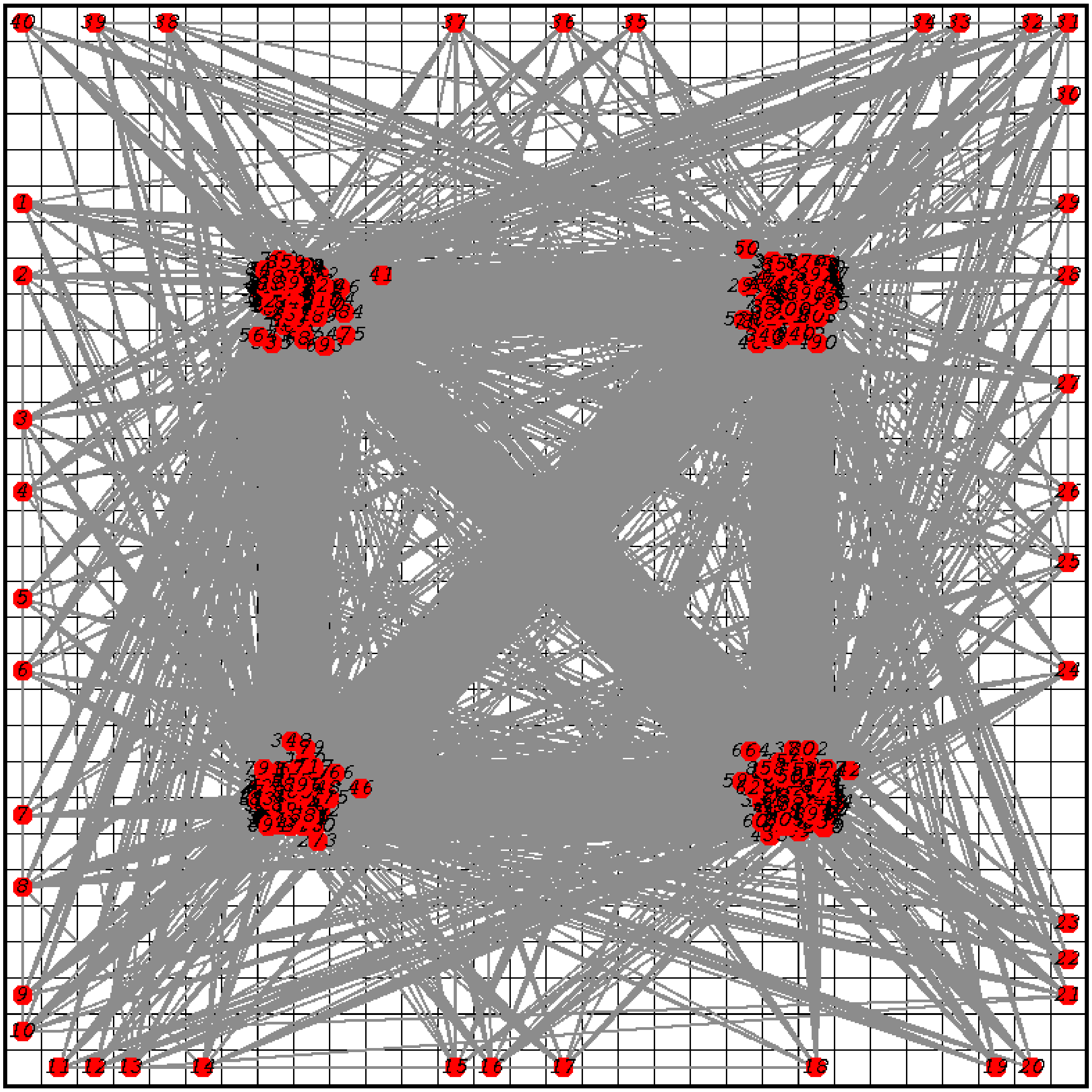

Figure 9: cct4 First Spread

Question 3:

File	HPBBWL, W=20
cct2	1538.36
cct3	11636.3
cct4	36947

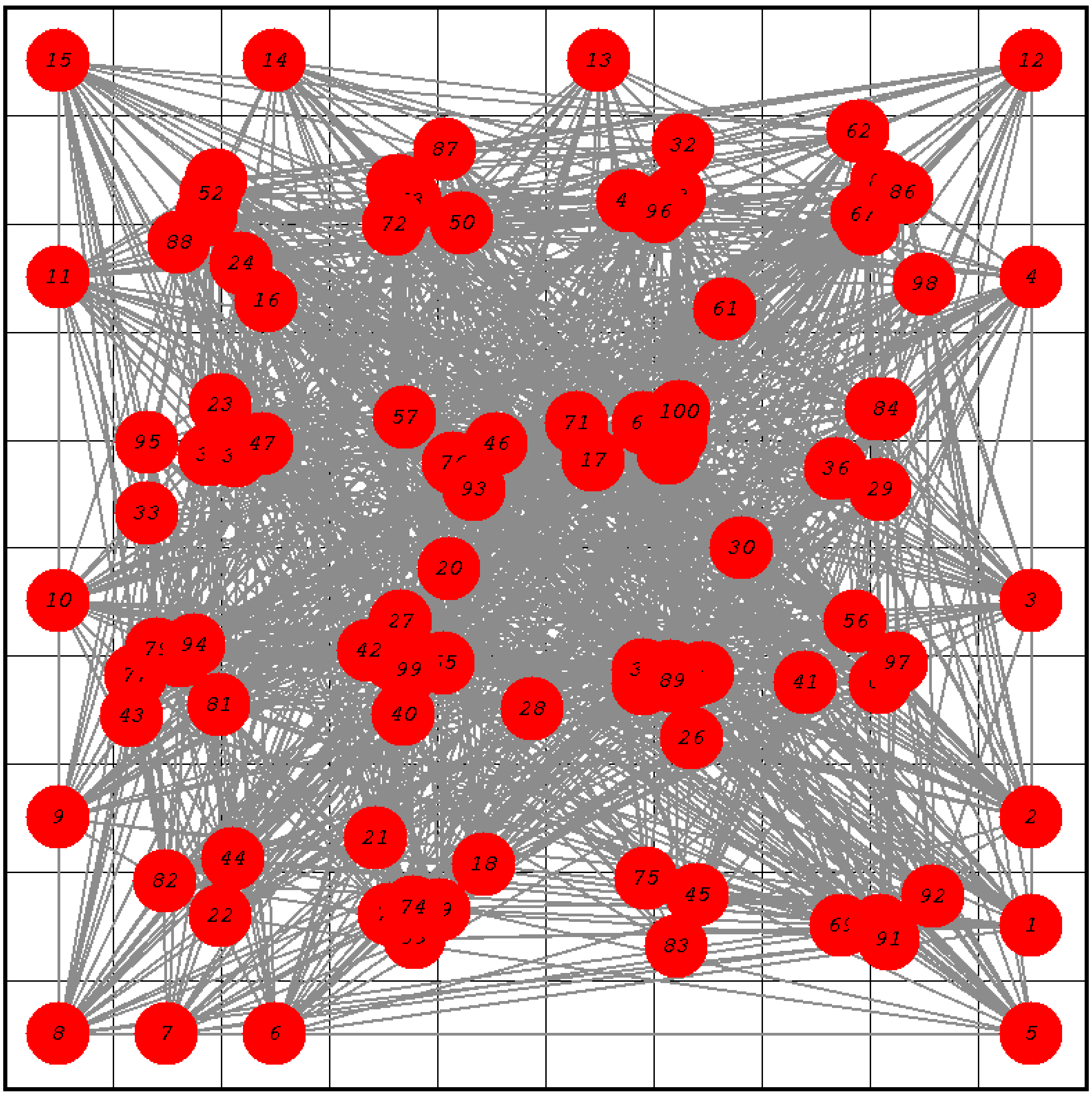

Figure 10: cct2 Full Spread

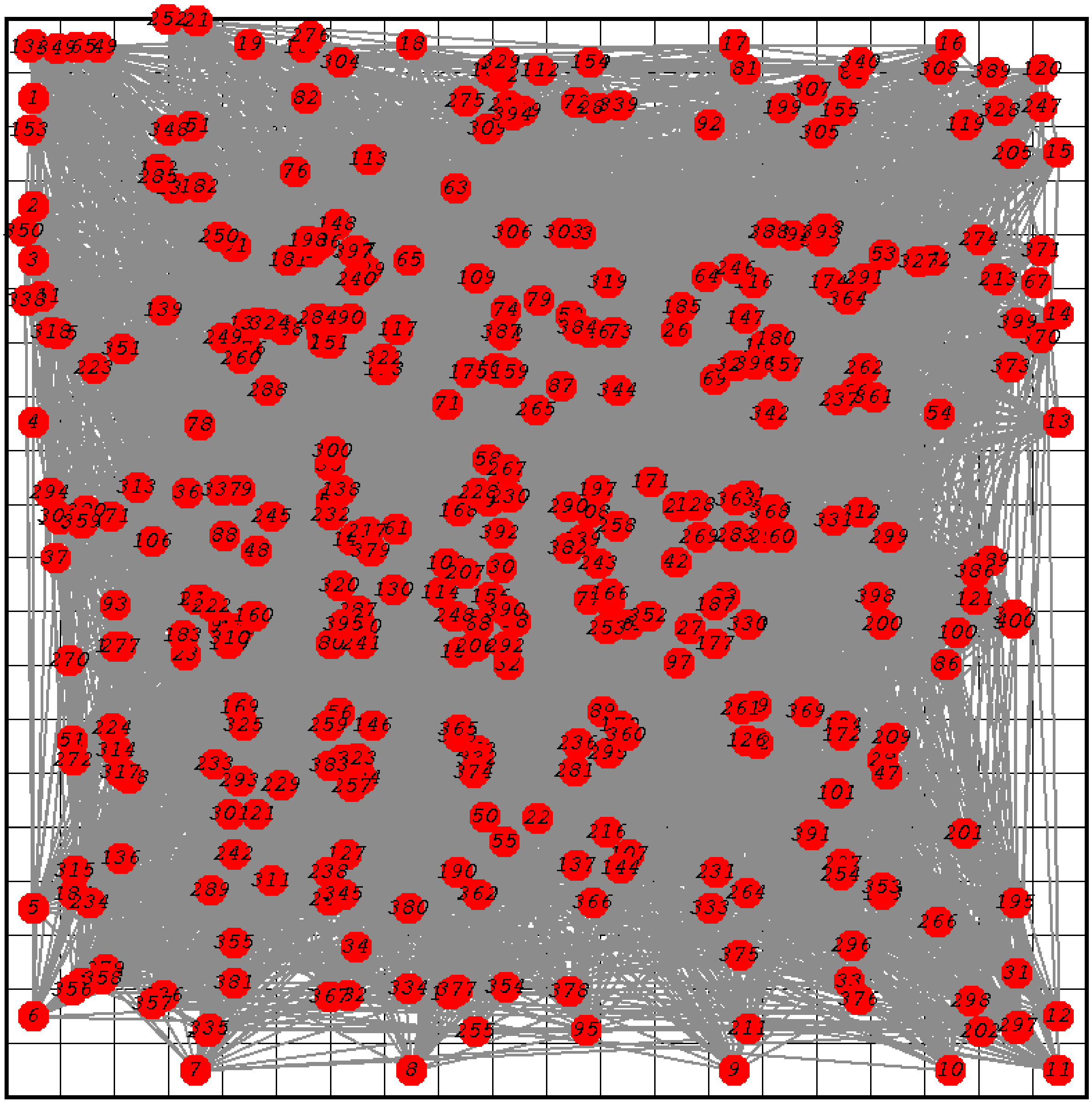

Figure 11: cct3 Full Spread

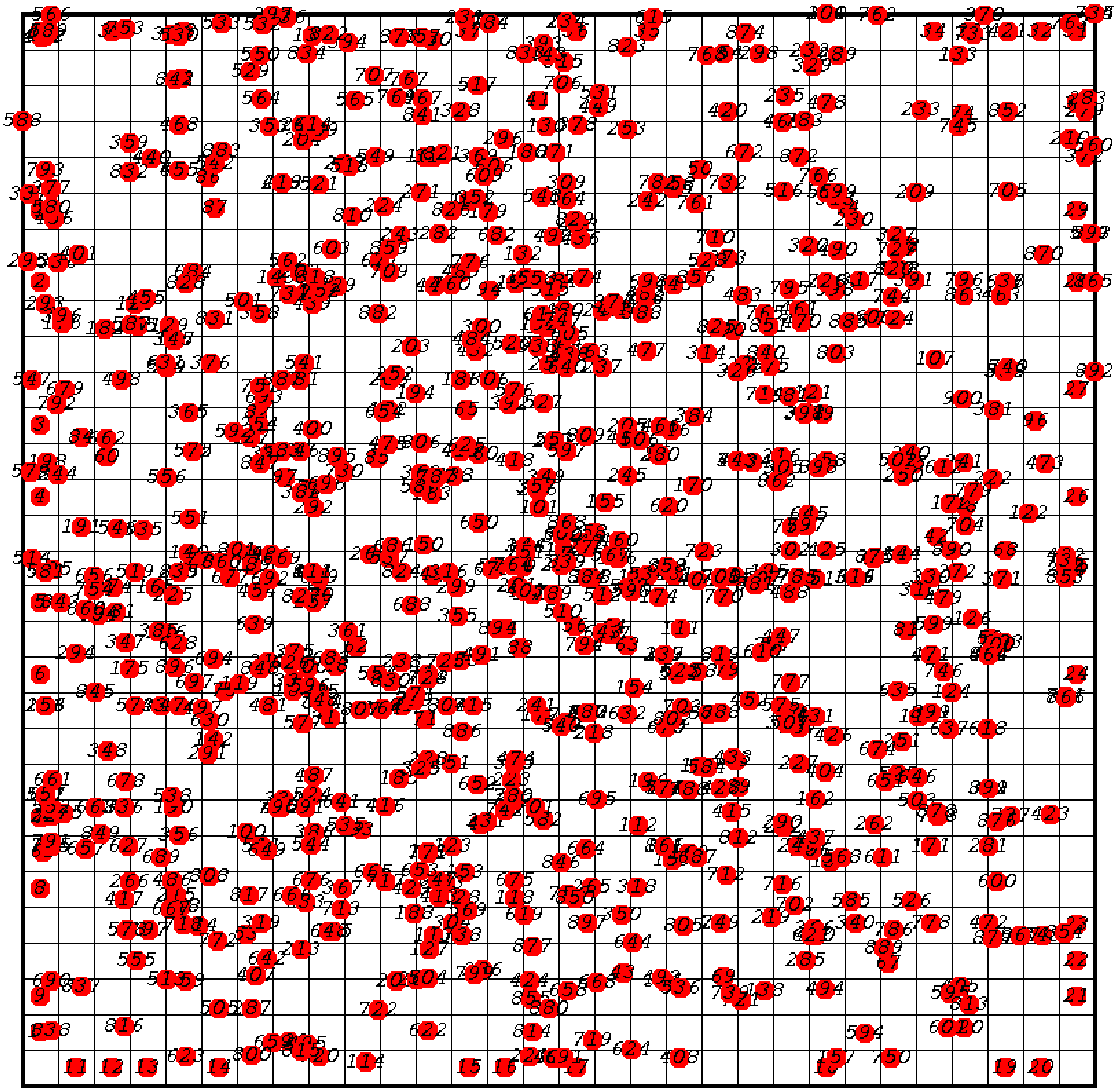

Figure 12: cct4 Full Spread (shown without nets)

Note: this spread level is not implemented in the program for bug where last 4 nodes of grid are not recognized. In final implementation cct4 is limited to 4 spreads.

Question 4:
The increase between the wirelength from the last iteration of spreading to the snapped locations is less drastic than the difference between the optimal and spread locations. This is because as the spreading increases, the HPBBWL becomes more optimal relative to the final position. If the original design was snapped, the snapped value would be larger than the final solved positions, because incrementally each position becomes slightly more optimal with each iteration.

File	**HPBBWL, W=20**
cct2	1659
cct3	12580
cct4	39377

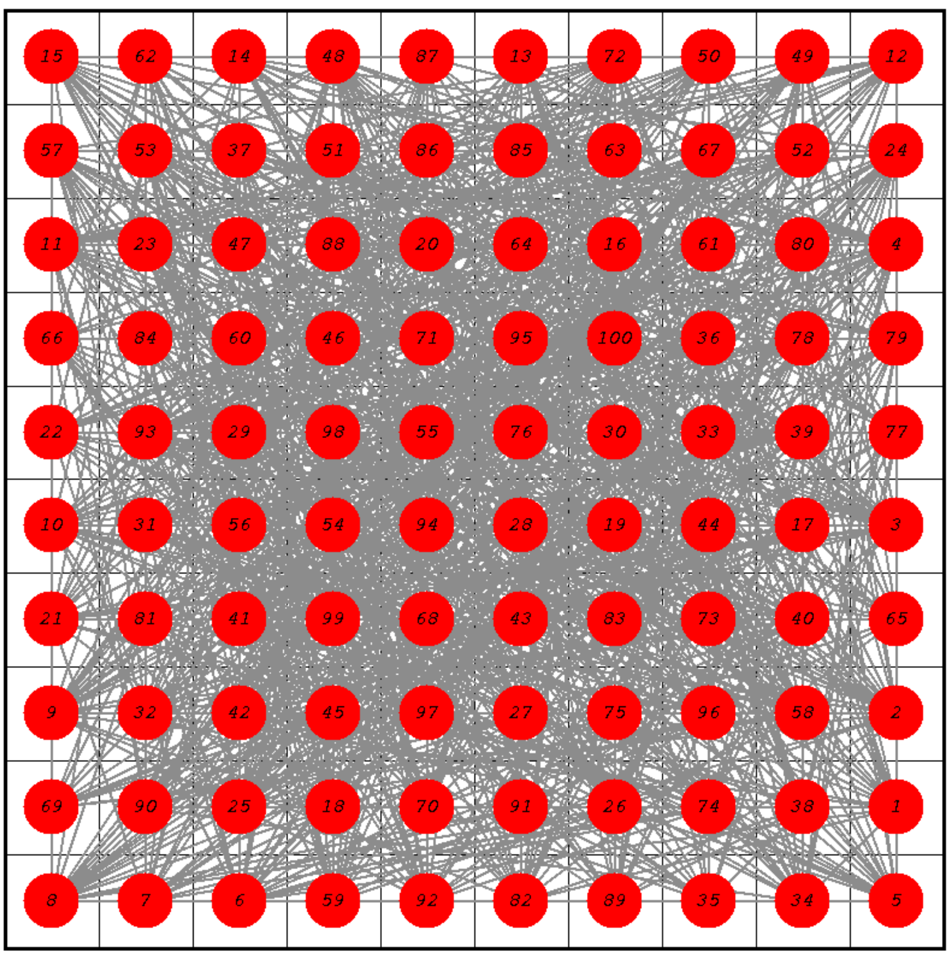

Figure 13: cct2 Snap Locations

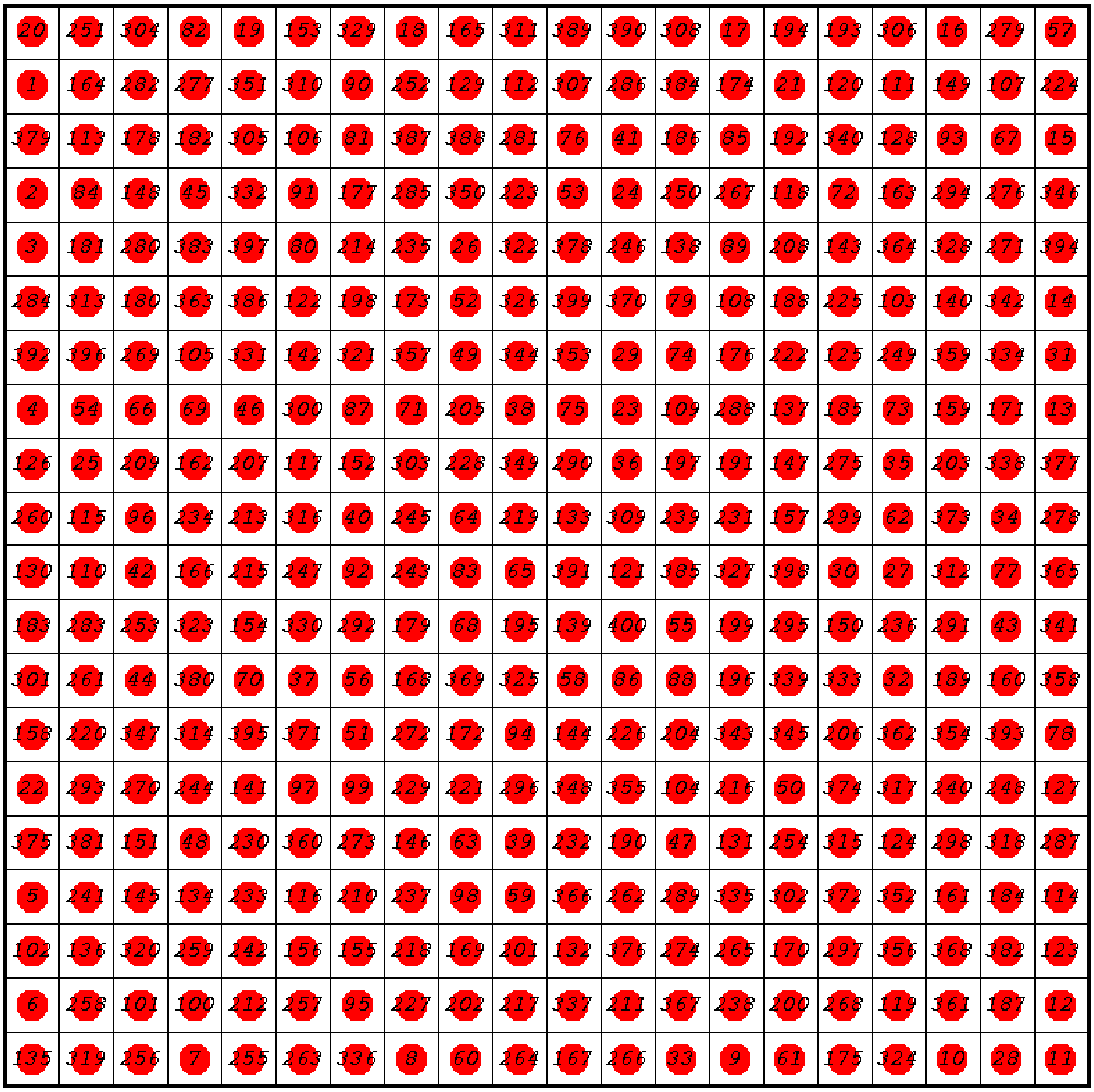

Figure 14: cct3 Snap locations (shown without nets)

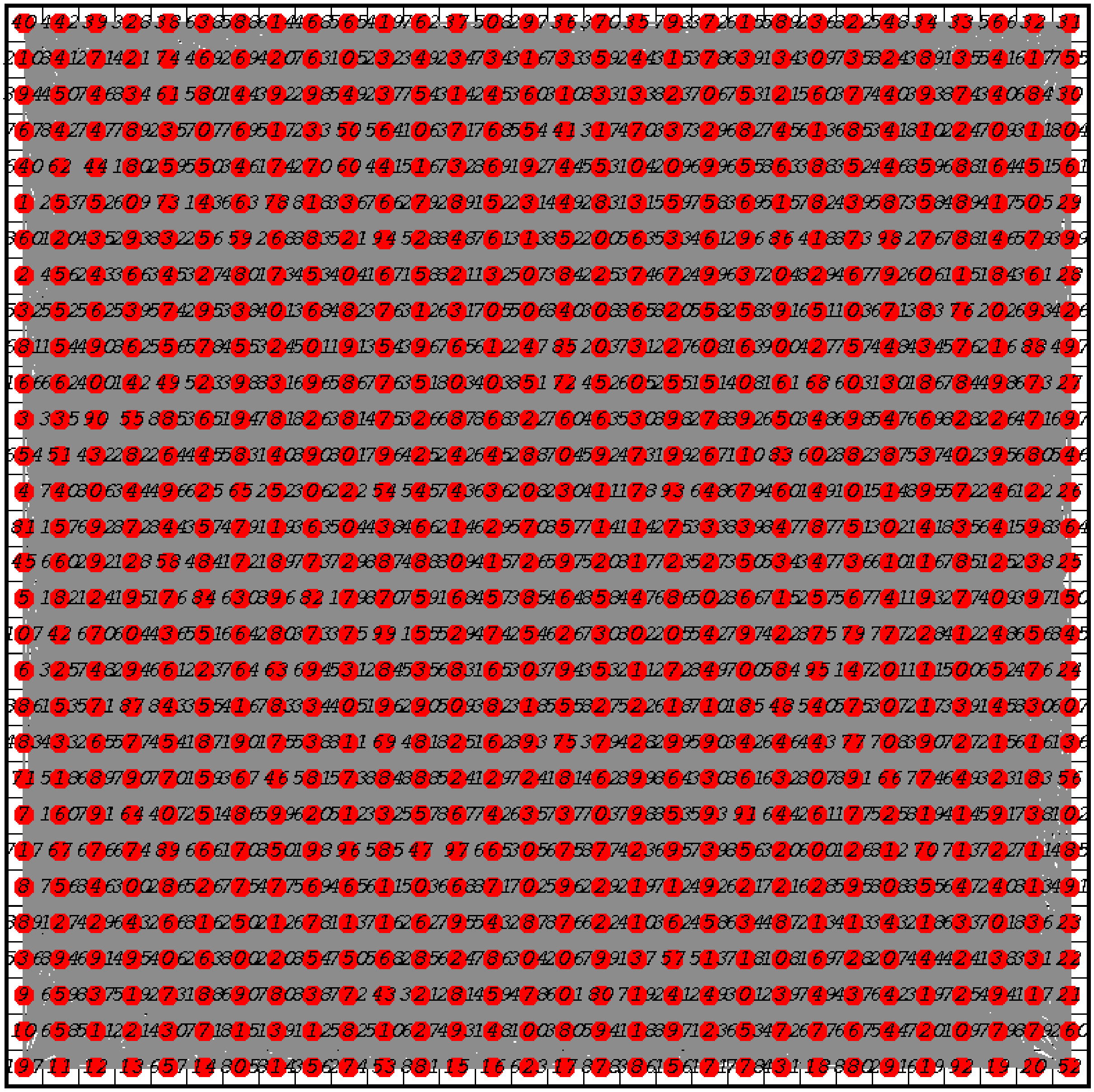

Figure 15: cct4 Snap Locations