Code Description

1 Data Structures

Two data structures (Net, Pin) are defined to save and store input data. These structures keep track of the existed connection between different pins. Then, I convert the cells' connections to a clique model to formulate and solve the problem. The other important data structure (Bin) is defined to spread the solved solution and make the solution legal. Details on these data structures are given below.

1. Pin/Fixed_Pin:

   - pin_number shows the pin id of the corresponding pin.
   - connection_number shows the number of connections that the corresponding pin has. For instance, if pin #2 is connected to net #1, #3, #5, then the connection_number will be 3.
   - connections keeps the nets id to which this pin is connected. For the previous example, the connection will be an array consisting of [1,3,5].
   - fixed is a bool attribute that shows whether this pin is fixed or not since the placement algorithm is not allowed to move fixed pins.
   - x_loc and y_loc are the locations of each pin after formulating and solving the placement problem. If a pin is fixed, these two attributes are given in the input file. Otherwise, these elements will be filled while the system has been solved.
   - moved_x_loc and moved_y_loc are the locations of each pin after spreading the solved system to reach a legal solution. It is clear that moved_x_loc and moved_y_loc for a fixed pin will be x_loc and y_loc since fixed pins will not move. These attributes are defined for two purposes. First, displacement value can be easily calculated using these attributes and the solved location of each pin (x_loc and y_loc). Second, while moving cells in the spreading algorithm, we will need to know the solved location of each pin to calculate whether moving this pin is violating the maximum-allowable cell movement parameter since a pin can be moved more than once.

2. Net:

   - connected_block_len shows the number of pins connected to this net id. For instance, if pin #3, #2, and #5 is connected to net #1. This attribute for net#1 will be 3.
   - connected_block is an array that keeps the pin id that is connected to this net id. For the previous example, connected_block for net#1 is [3,2,5].

- weight is a weight corresponding to each connection. This weight is calculated based on the clique model. For the previous example, net#1 has three pins connected to it. Therefore, the weight for each connection will become 2/3.

3. Bin:

   - x and y show the bin location on the x-axis and y-axis. Figure 1 shows the 4*4 grid and its indexing to clarify the value of x and y for each bin.

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0)	(1,0)	(2,0)	(3,0)

Figure 1: 4*4 Grid and its x and y indexing

   - block_count shows the number of pins that are within this bin after solving the problem. If this number is bigger than one, it means that this bin is overfilled.
   - blocks is an array that keeps the pin id that is within this bin id.
   - visited is a bool variable used by the BFS function to find various paths for each overfilled bin. The function will initialize this variable to false, and after visiting them, will change this variable to true to avoid visiting a bin in an iteration. After finding the needed number of the path for a specific bin, these attributes will reset to false for the next bin.

2 Routines

1. make_clique calculates the weight of each connection based on the clique model. It simply divides two by the number of cells connected to this net id. Therefore, it fills the weight attribute in the net structure. For instance, if four cells are connected to a net, the weight of each connection will be 1/2.

2. cal_weight returns a matrix for movable cells containing weights based on the clique model. It can calculate both matrices for the x-axis and y-axis based on its input parameter. The equation to calculate matrix elements in the x-axis are given below.

$$i \neq j, Qx(i,j) = -1 \times w(i,j)$$

$$i = j, Qx(i,j) = \sum_{k} w(i,k)$$

3. cal_fixed is like a previous function which returns a fixed matrix to solve the problem. It can calculate both matrices for the x-axis and y-axis based on its input parameter. The equation to calculate matrix elements in the x-axis are given below.

$$c(i) = \sum_{fixed\, k} w(i,k)$$

4. solve fucntion solves the matrix equation using the given Linear System Solver [1]. It has two output matrices corresponding to the x and y location of each movable cell.

5. create_bin creates a grid based on the maximum x and y of the fixed pins (die size). Therefore, it traverses pins for every bin, decides whether the pin is within the current bin, and sets the overfilled number for each bin. This function is executed after solving a system once. Hence, each pin has a potential location when this function is called.

6. sort_overfilled_bins sorts the overfilled bins, which have more than one pin within themselves, in ascending or descending order. It uses the default sort function in c++. Therefore, a compare function between two bins has been defined, which compares the overfilled number.

7. compute_cost computes the cost between a source and a sink bin. The third parameter is the maximum-allowable cell movement since if moving between two cells violates this number, the cost must be infinity. The compute cost function returns a pair of double and integer; The first double is cost value, and the integer is the pin id which can be moved within these two bins with the minimum cost. The cost is calculated based on the quadratic wire length.

8. find_path gives an overfilled bin and the maximum-allowable cell movement, and returns a set of possible paths which cells within the overfilled bin can move along this path to spread the cells. This function uses a simple BFS-like algorithm to find a path and returns a pair of queues and a double array regarding the paths and their costs. Therefore, paths can be sorted based on their cost in ascending order.

9. spread is the primary function that executed the flow-based spreading [2]. This function tries to spread the cells to convert the solved solution to a legal solution. It will be executed until no overfilled bin exists. It sorts overfilled bins using the sort_overfilled_bins

function and loops over each bin to find a set of possible paths for the corresponding bin using the find_path function. After discovering potential paths, it starts to move cells along the found paths to reduce the overfilled number of the bin. It terminates when the bin is not overloaded or runs out of the potential path. The important point in the moving is that it moves cells in the reverse order instead of starting from the overfilled bin toward the empty bin. After each movement, path cost will be recomputed to avoid violating the maximum-allowable cell movement. After execution of this function, the solution is legal, and no overfilled bin can be found.

10. add_anchor adds a fake net between a cell and itself with a determined weight. It changes the weight matrices in the x-axis and y-axis. Then, we again solve the problem using the new matrices to see that the solved location will gradually be converged to the spread locations.

3 Software Flow

The program will start to store the pins and their connection into their proper data structure. Then, it calculates weight matrices and fixed matrices to formulate the problem. Later, use these matrices to solve the problem and estimate the solved location of each movable pin. Since the solved location is not legal, the spread function spreads the pins without drastically damaging the HPWL. After applying the spread function, fake nets between a cell and itself will be added to weight matrices. Then, we solve the system again to see that the solved location will be converged to the spread location.

Results

- Solve without Spread
  In this section, Only formulation and solving have occurred, and no overlaps have been removed. The HPWL for each given test case has been reported in Table 1.

Table 1: HPWL After Solving System

cct#	HPWL
cct1	185.278
cct2	733.552
cct3	4408.49

- Solve and Spread
  In this section, overlaps have been removed using the spread function. I have initialized the iteration number to one in this step and sorted the overfilled bins in ascending order exactly as Darav's 2019 paper [2]. The HPWL before and after spread and displacement values have been reported in Table 2.

Table 2: HPWL After Solving and Spreading System

cct#	HPWL Before Spread	HPWL After Spread	Displacement
cct1	185.278	270.765	25
cct2	733.552	1003.02	119
cct3	4408.49	11067	1961

While spreading, overfilled bins have been sorted in ascending order which means the program moves the cells from less overfilled bins. However, the reverse approach can be taken to transfer cells from the most overfilled bin first. Darav's 2019 paper [2] states that changing this order does not significantly affect displacement number and final result. But it will increase the execution time since the potential paths for each bin in the loop may decrease. Executing the program in the reverse order shows the same result. However, the displacement number is not the same for each test case. For the first test case, the displacement number is 25 for both sorting approaches. For the second test case, the displacement number has been increased by one but does not have any significant change. For the last third case, the displacement number has decreased by about 1.52%. The results are shown in Table 3 for both sorting approaches. For these test cases descending approach has a better result. However, since we did not measure execution time for these approaches as said in the paper, we can not conclude that descending order always has a better displacement number.

Table 3: Displacement For Ascending or Descending Sort

cct#	Ascending	Descending
cct1	25	25
cct2	119	120
cct3	1961	1931

Table 4: cct1 Displacement and HPWL for Different ψ

Different Approaches for ψ	$\psi = (iter+1)^2$	$\psi = 2^{(iter+1)}$	$\psi = (iter+1)^4$	$\psi = (iter+1)$
HPWL	270.765	263.798	264.203	261.204
Displacement	25	25	25	25

Different attempts for ψ initialization and update have been explored in this section. Results for each test case are given in Tables 4, 5, 6,. Each approach has its advantages and its disadvantages compared to the paper initialization and update. When updating the ψ by little steps (last column of the tables), the HPWL decreases in all three test cases compared to the baseline (second column of the tables). However, updating ψ using tiny steps can increase the displacement value since more movement occurs. On the other hand, Updating the ψ using more significant steps can result in decreased displacement value for all three test cases since the number of valid paths for each overfilled at each iteration increases. In conclusion, for these test cases, the third column seems to have the best solution. However, we can not extend this conclusion to other placement problems.

These reports on ψ initialization and update are for the ascending sort. I also tried all four approaches with descending sort order for all three test cases. The best approach that I could find to result in less HPWL and less displacement number is to sort overfilled bins in ascending order and use the fourth power of iteration to initialize and update the ψ. Therefore, the next section for anchor weight uses this setting to compute anchor solve results.

By adding a fake connection between a pin and itself, weight matrices and right-hand side matrices change accordingly. Therefore, resolving the system will result in a different location for each pin. The HPWL for the new solution is reported in Table 7. If fake net weight is strong, then the new location of each pin will be around its spread location. If the fake net weight is small, then the new location of each pin will be toward its first time solve location. HPWL for strong weight will be around the spread HPWL, and for weak weight will be around solved HPWL.

Table 5: cct2 Displacement and HPWL for Different ψ

Different Approaches for ψ	$\psi = (iter+1)^2$	$\psi = 2^{(iter+1)}$	$\psi = (iter+1)^4$	$\psi = (iter+1)$
HPWL	1003.02	999.61	998.551	983.969
Displacement	119	120	119	121

Table 6: cct3 Displacement and HPWL for Different ψ

Different Approaches for ψ	$\psi = (iter+1)^2$	$\psi = 2^{(iter+1)}$	$\psi = (iter+1)^4$	$\psi = (iter+1)$
HPWL	11067	11098	11012.2	11043.7
Displacement	1961	1958	1933	2009

Table 7: Different Anchors HPWL

cct#	Solved HPWL	Spread HPWL	Small Anchor HPWL	Large Anchor HPWL
cct1	185.278	270.765	188.697	261.647
cct2	733.552	1003.02	745.253	968.737
cct3	4408.49	11067	4608.91	10086.9

Plots

While running the code, the screen shows only bins and fixed pins. The following buttons have been defined to avoid the screen becoming very crowded to see solve and spread and other options.

- Solve: When Solve button is pressed, the program will show the solve result without any overlaps removal. At this stage, only pins and their labels are shown on the screen. To see the connection between pins, Nets or 1 Net button should be used.
- Spread: The program will show the solve and spread result with removed overlaps when the Spread button is pressed. At this stage, only pins and their labels are shown on the screen. To see the connection between pins, Nets or 1 Net button should be used.
- Anchor: When the Anchor button is pressed, the program will add anchors to weight matrices and solve the problem again. Then, it shows the new location for each pin after re-solving the system.
- Nets: When the nets button is pressed, the entire connections between all cells are shown on the screen. The connections are drawn based on the clique model. Two pins connection's color is light gray, and clique connections' color is yellow. You can press this button again to remove the connections from the screen.
- 1 Net: To see connections without the screen becoming too crowded, this button shows one connection at a time. You can keep pressing this button to see all available connections.

Plot results after executing each given test case are attached to appendix A(3).

References

[1] Texas A&M University, Tim Davis (SuiteSparse: A Suite of Sparse matrix). https://github.com/DrTimothyAldenDavis/SuiteSparse

[2] Nima Karimpour Darav, Andrew A. Kennings, Kristofer Vorwerk, Arun Kundu: Multi-Commodity Flow-Based Spreading in a Commercial Analytic Placer. FPGA 2019: 122-131.

Appendix A

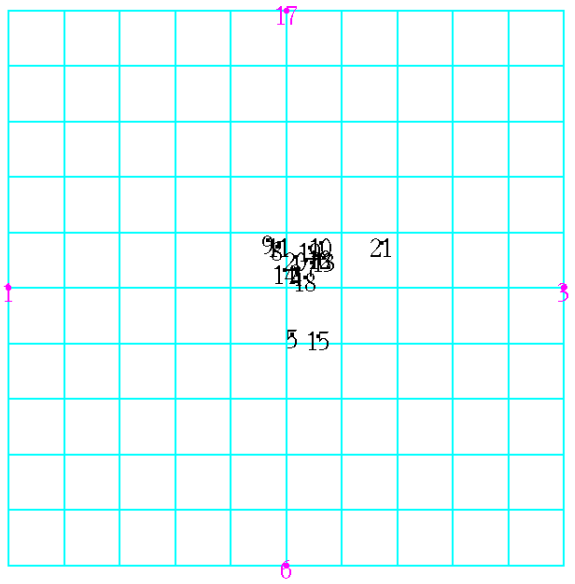

Figure 2: Solve Step - cct1

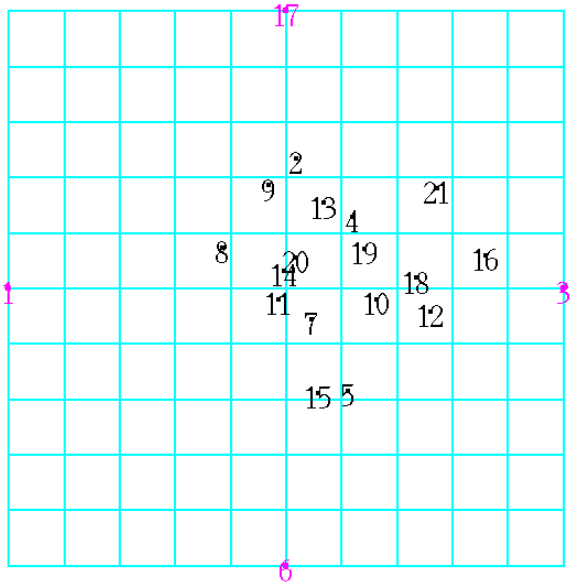

Figure 3: Spread Step - cct1

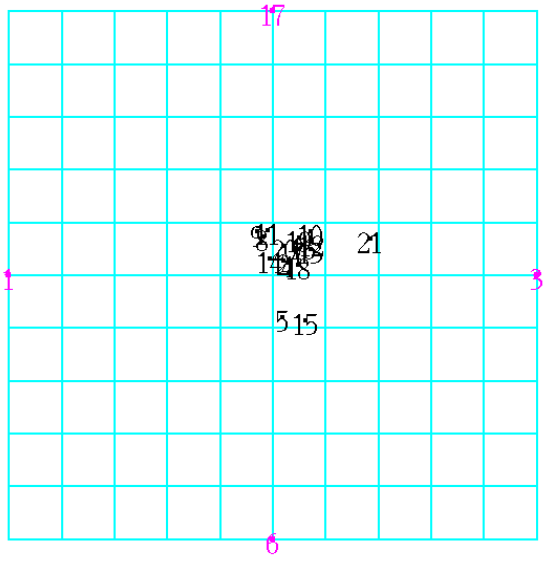

Figure 4: Small Anchor Step - cct1

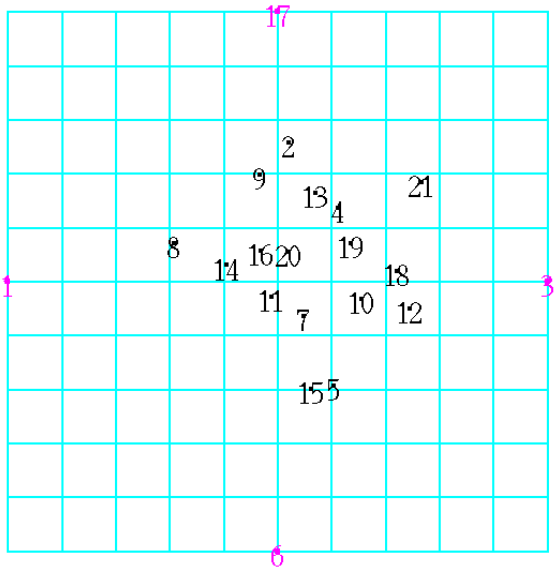

Figure 5: Large Anchor Step - cct1

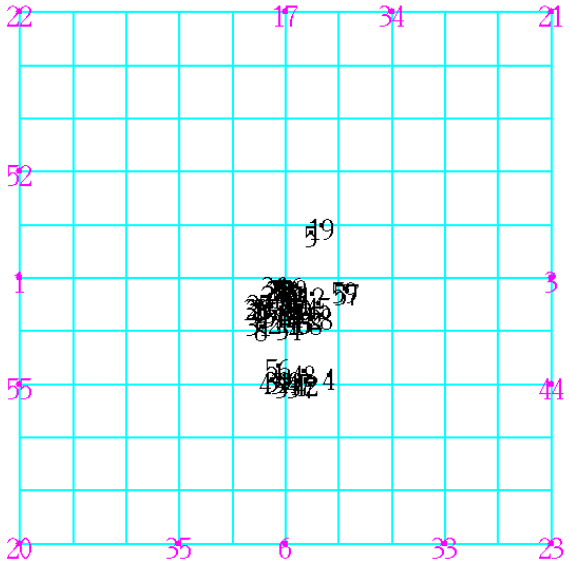

Figure 6: Solve Step - cct2

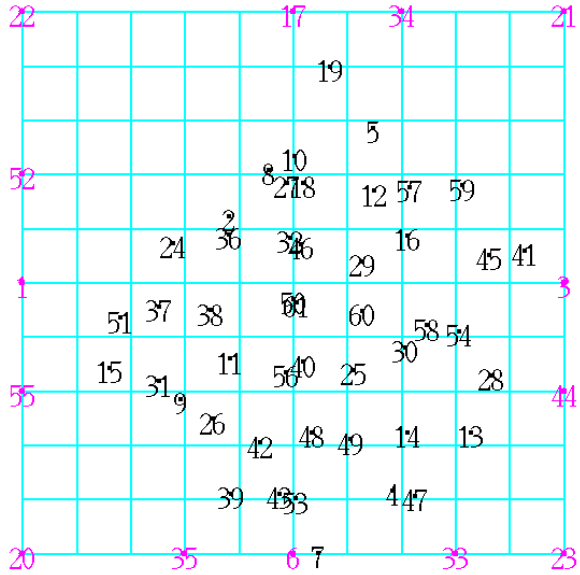

Figure 7: Spread Step - cct2

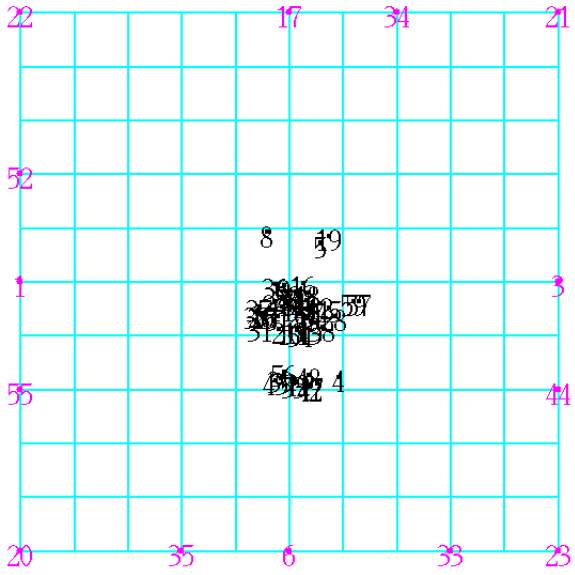

Figure 8: Small Anchor Step - cct2

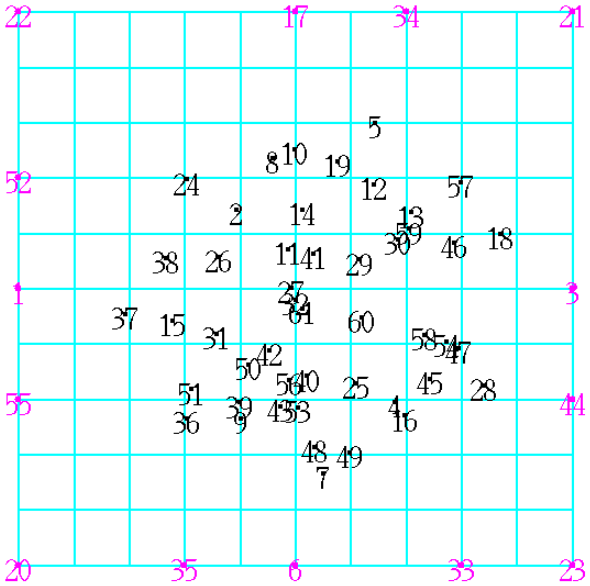

Figure 9: Large Anchor Step - cct2

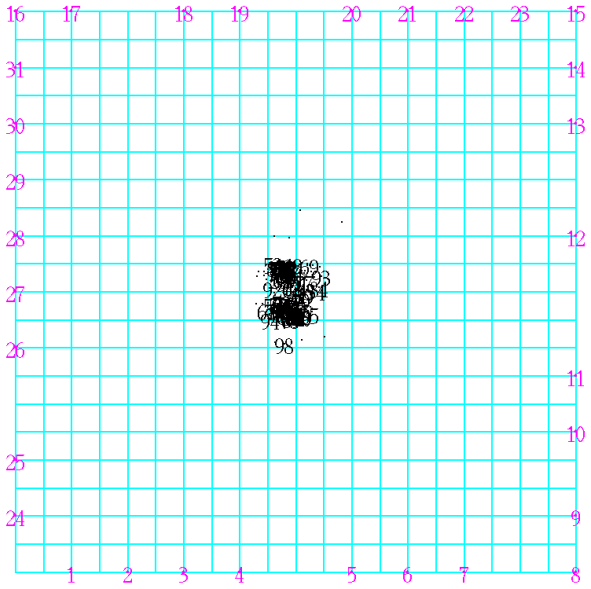

Figure 10: Solve Step - cct3

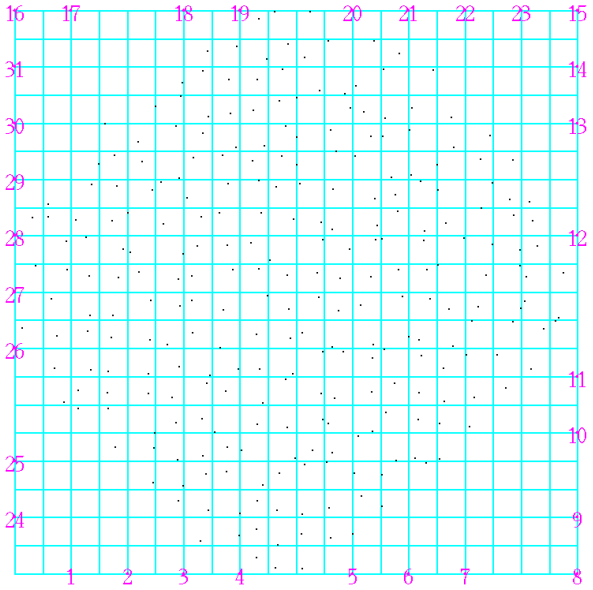

Figure 11: Spread Step - cct3

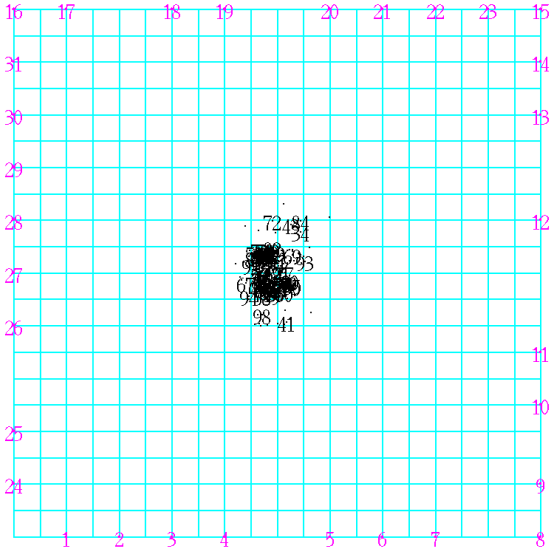

Figure 12: Small Anchor Step - cct3

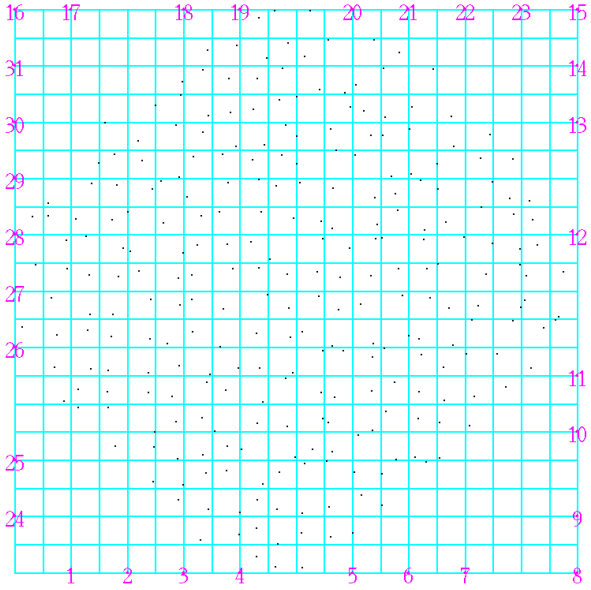

Figure 13: Large Anchor Step - cct3