Physical Design Automation Lab3 Report

電子所 郭家均 31351017

❖ Pseudo Code

There are two main method I use in my lab, **FindVacant** and **FindSwap**

FindVacant is greedy method to place to insert MBFF, no moving cells required.
Basically started from the given x y coordinate, then sorted all the possible insert points within neighboring rows by their distance from the original coordinate. Then check overlap for each point in incremental order.

Algorithm 1 FindVacant

```
1: Clear Search set
2: for each row in cell Y-span do
3:     for each subcell in row do
4:         for each y in cell Y-span do
5:             Lpt = (subcell Left - cell width, y)
6:             Rpt = (subcell Right, y)
7:             compute Lpt, Rpt distance from cell
8:             Insert Lpt, Rpt into Search List (sorted by distance)
9:         end for
10:    end for
11: end for
12: for each pt in Search List do
13:    set cell to (pt.x, pt.y)
14:    if cell is legal do
15:        return true
16:    end if
17: end for
18: cell set to original (x, y)
19: return false
```

This method can be modified to be much more greedy. To ensure FindVacant search the entire Die area, after searching the neighboring rows, we need to check the upper and lower row regions til it reached boundaries.

Only using this method can pass all 4 released cases, but runs a bit long. That's why I added the second feature.

Further details on FindVacant function method are in the special features section.

FindSwap is a method that reuses the FindVacant method.
Basically, we first find a place where cell only overlap with single-bit FF. Then we removes those overlapped SBFF to insert the MBFF, and try to insert removed SBFF as near as possible to their original location. The modified FindVacant is **FindSRVacant** below.

```
Algorithm 2 FindSwap
1: FindSRVacant for cell
2: Overlap List clear
3: Cell Memory List clear
4: for each row in cell Y-span do
5:     for each subcell in row do
6:         if subcell overlapped with cell do
7:             subcell add to Overlap list
8:             remove subcell from placement row
9:         end if
10:    end for
11: end for
12: Insert cell into placement row
13: while overlap not empty do
14:    subcell = overlap front
15:    if accumulate distance > threshold do
16:        remove cell
17:        restore subcell coordinates from Cell Memory list
18:        insert subcell
19:        return false
20:    end if
21:    if FindVacant(subcell) do
22:        Cell Memory List added subcell
23:        distance += subcell traverse
24:        overlap pop
25:    else do
26:        remove cell
27:        restore subcell coordinates from Cell Memory list
28:        insert subcell
29:        return false
30:    end if
21: end while
22: return true
```

The problem of this method is the total distance of SBFF would possibly be very large, so we need a threshold value to quit this function at line 15.

Further details on FindSwap function method are in the special features section.

❖ **Time Complexity Analysis** (n = total cell num / m = row num)

<FindVacant>

Insertion of Search list point

Since Search list is implement by std::set, insertion would be around

log((cell_row) * (cell_row) * (# of cells in row)) = C*log(n)

Check overlap

Since element access time of set is log(n), checking process would be

C*n*log(n)

Greedy Search

Iterating through every row, we got worst case

C*m*n*log(n)

<FindSwap>

FindSRVacant

This function is FindVacant without greedy part

C*n*log(n)

FindVacant of overlapped subcell

C*n*log(n)

We can observe that FindSwap is more efficient than the space search only option.

	FindVacant	FindSwap & FindVacant
Case1_16900	12s	9s
Case1_ALL0_5000	200s	141s
Case2_100	59s	44s
Hellish_case	246s	177s

To further improve the speed of my code, there are several approach I think of but don't have the time to implement.

1. Instead of dumping all candidate point into search set, I can use spiral search method (heuristic approximate to finding point closest to original point). Though it may not be optimal answer, I can iterate next point if current point is not legal.

2. For specific type of MBFF, I can create a reject list to document all searched point. Avoiding traversing same point for multiple times.

3. Change the data structure to O(1) access time structure (e.g. vector). Theoretically if I'm doing space search only, there are little portion of time of inserting and removing cells. Therefore, access time reduction is an obvious win.

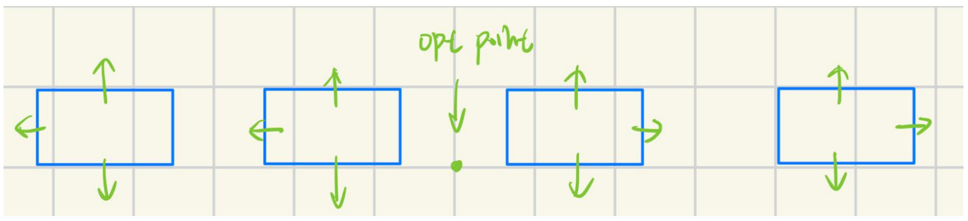
❖ Special Features

Set interval structure:

I use vector<*set<CELL*>> to implement the placement row, for each row I insert cell pointer to the row set sorting by its lower left x value. Since cells cannot overlap, I can use lower_bound function of std::set to quickly found the neighboring cell of one point. Which consider std::set is a red-black tree, only use log(n) to traverse.

Compact to center:

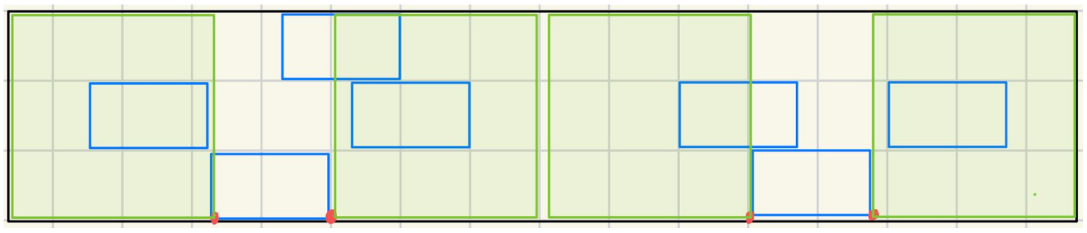
Instead of blindly inserting the cell toward lower left corner, my space search algorithm is leaning towards inserting cells to the assign points. Furthermore, I'm using neighboring cells as insert reference so that we don't iterate site one by one. (green arrows below are the searching directions)



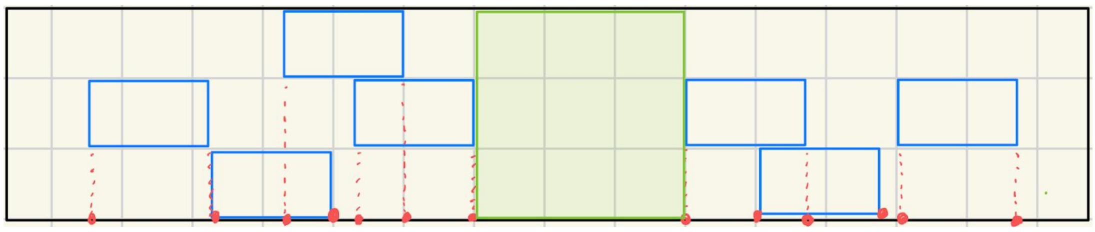
Multi-Row-Cell search candidate selection:

Instead of only using based row left & right, we uses all left & right boundary of every row cell occupied. This method greatly increase the chance of finding vacant place. Example is drawn below to demonstrate.

<Based row candidate>
all insert points are illegal



<Multiple row candidate>
We can find one legal insert point



FindSwap improvement

Only inserting the vacant place can lead to large cost because of large distance between given insertion points and vacant position. FindSwap changes the space insertion operation from bigger MBFF to much smaller SBFF, which greatly increase the chance of finding vacant place in near by positions.

❖ Feedback

This lab was definitely a challenging experience. Every step was crucial because of the extremely high number of cells involved. From selecting the right data structures to optimizing algorithmic time complexity, I spent nearly two weeks refining the data structures before even starting the implementation. Debugging this lab was especially tough, and using a visualizer to check the placement results turned out to be a tedious process. This week, I encountered all kinds of issues—segmentation faults, bad memory allocations, and more. I have to admit that my less-than-ideal coding style contributed to many of those late-night debugging sessions. :)

On the bright side, completing this project was incredibly rewarding! Seeing the legalized placement results pop up was fascinating. Although my performance might not compare to some of the other excellent work, I learned so much from this experience. I can confidently say this project is one of the best I've worked on so far (besides the VLSI Lab MAC Layout).

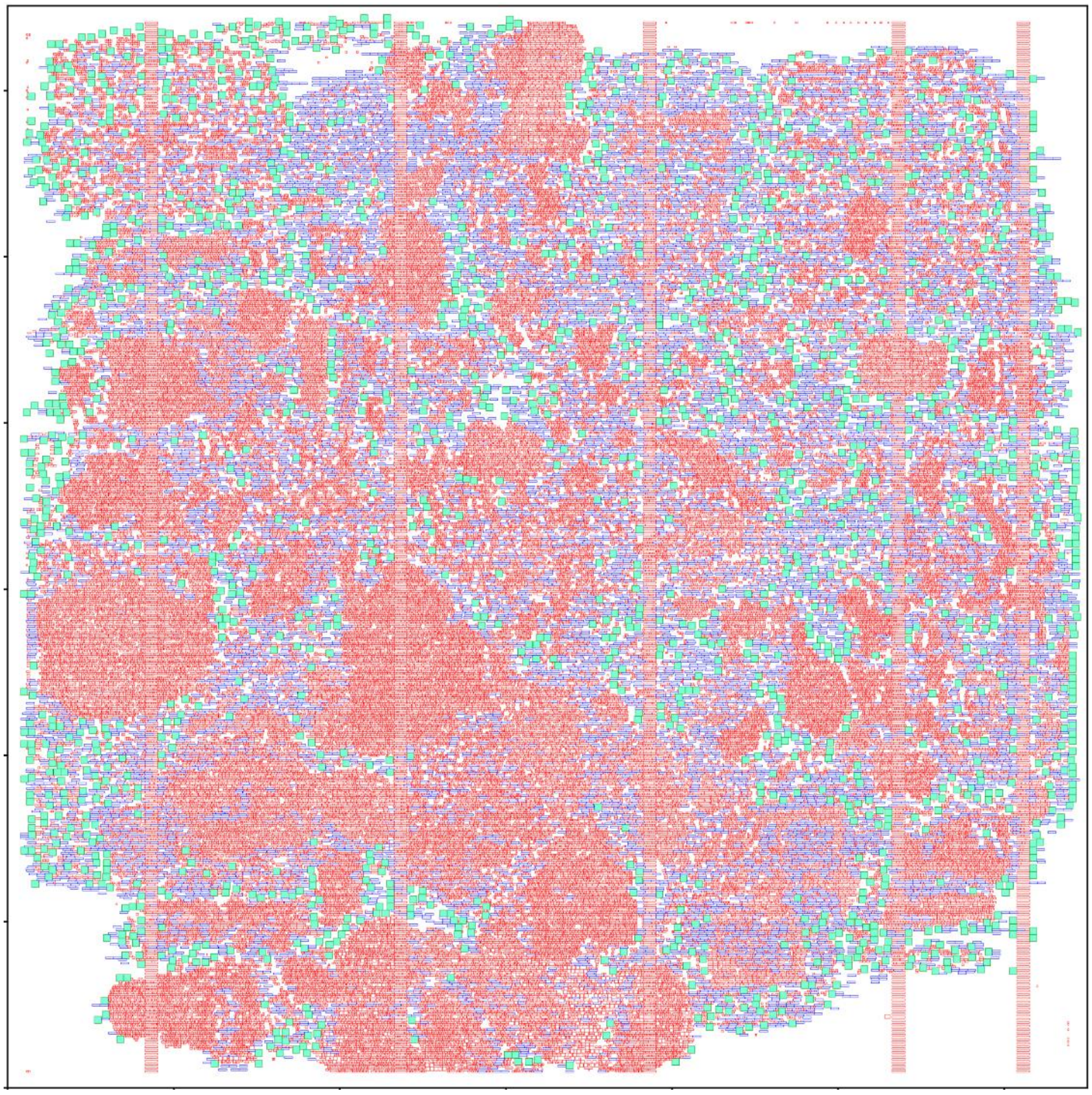
Thank you to the TAs for preparing such an excellent lab! Despite the many challenges, I gained a lot from it.

❖ Conclusion

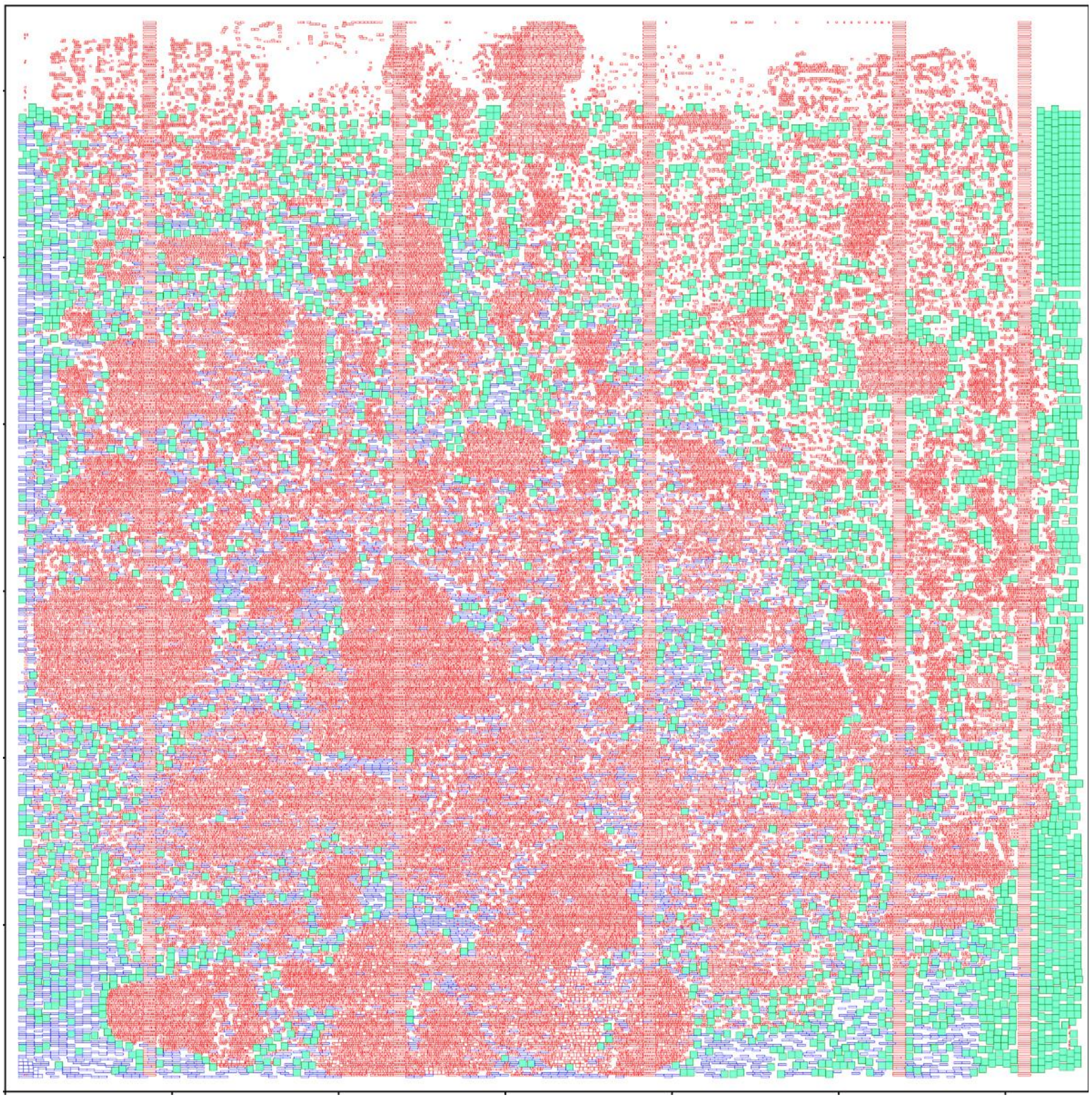
Legalization is a crucial steps in placement of PDA, and a good legalizer can locally legalize the design without changing the overall layout of the placement. I implement a vacant place finder, and a some-what moving swapper. The runtime is way less than 30 minutes, and the layout result are in the next couple of pages.

❖ Result

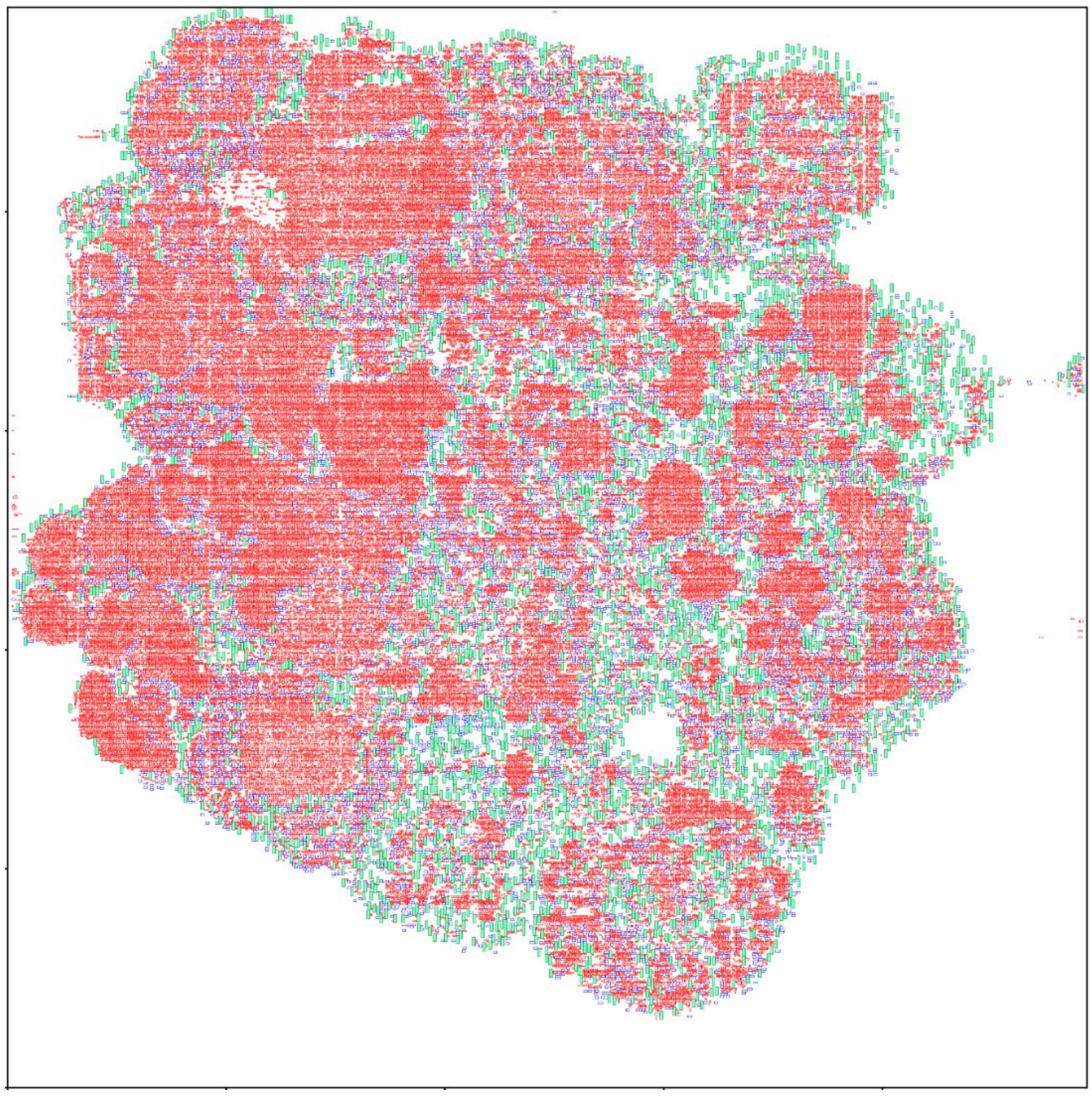
<testcase1_16900>



<testcase1_ALL0_5000>



<testcase2_100>



<testcase1_MBFF_LIB_7000>