EE 382V: VLSI PHYSICAL DESIGN AUTOMATN

Wirelength Driven Detail Placement With Instant Legalization

ZHIYUAN ZOU, YUCHAO HAN

CONTENT

1. Problem Formulation **1**
2. Overview **1**
3. Get Swap Target **2**
3.1 Optimal Region Based Swap Strategy 2
3.1.1 Find Optimal Region 2
3.1.2 Penalty On Overlap 3
3.1.3 Get Good Target In Optimal Region 5
3.2 HPWL-Driven Swap Strategy 6
3.3 Strategy Analysis 7
4. Instant Legalization **8**
4.1 Motivation 8
4.2 Problem Formulation 8
4.3 Fast Legalization Algorithm 8
4.3.1 Quadratic Programming Formulation 9
4.3.2 Algorithm for Computing Clusters 11
4.3.3 Improved Collapsing Algorithm 11
5. Run Time Optimization **14**
5.1 Lazy Update of Swapping and Legalization 14
5.2 Map Data Structure 14
5.3 Change_key Operation 15
5.4 Time Complexity Analysis 16
6. Results and Analysis **17**
7. Possible Improvement **19**
8. Reference **20**
9. Appendix **21**
A.get_target.h 21
B. class_define.h 28
C. result_func.h 33
D. placerow.h 34
E. main.cpp 56

1. Problem Formulation:

Our project is to make the detail placement on a given netlist file after globe placement. The goal is to minimize the total half-parameter wirelength. We use the benchmarks from ICCAD2014.

2. Overview:

Our implementation can be divided into two separate parts: the **Get Swap Target part** and the **Instant Legalization** part. In Get Swap Target part, we find the best swap target for a given cell by combining two different swap strategies - Optimal Region Based Swap Strategy and HPWL-Driven Swap Strategy. In Instant Legalization part, we achieve the goal of making the placement legal right after each swap, which makes it more accurate to calculate the HPWL that will be used in the next swap iteration.

The difficulty in our implementation is the run time. Because we need to find the good swap target and make instant legalization every swap iteration, we have to make each iteration blazing fast. So we use a special data structure to store all the information: each cell stores a list of its connecting nets, each net stores a list of its connecting cells, each row stores a map mapping to the position of all the cells in this row.

The following report will explain the details of the two major parts in our implementation.

3. Get Swap Target:

3.1 Optimal Region Based Swap Strategy

3.1.1 Find Optimal Region

The optimal region for a cell is: Given all other cells in the circuit are fixed, the optimal region for a cell i is defined as the region to place where the wirelength is optimal. This region is determined based on the median idea of [1] [2].

Here is the way we use to calculate the optimal region. From a given cell i, we traverse all the nets connecting to it (noted as N_i) and find their bounding boxes excluding cell i. Thus, we have the left, right, lower and upper boundaries for each bounding box: $x_l[p]$, $x_r[p]$, $y_l[p]$, $y_u[p]$. We put them into two series. The optimal position for cell i is given by (x_{opt}, y_{opt}), where x_{opt} and y_{opt} are the median of x series $(x_l[1], x_r[1], x_l[2], x_r[2], \dots)$ and y series $(y_l[1], y_r[1], y_l[2], y_r[2], \dots)$. Because the number in x and y series are all even, we can get a region of (x_{opt}, y_{opt}). But in some circumstance, the lower bound of x_{opt} may be the same as its upper bound, and y_{opt} as well. So, the region may degrade into a line or a point. But in our implementation, it still make sense. If the region is a line, we just search the target on that line, if the region is a point, we just search one possible target.

The follow is an example of finding optimal region:

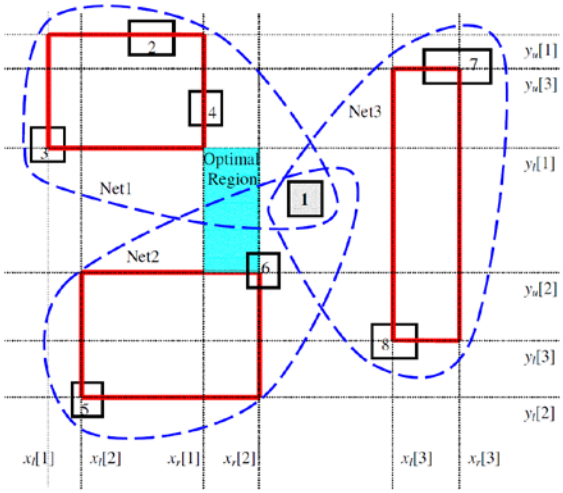

Figure 1. Optimal Region

We have x series ($x_l[1]$, $x_r[1]$, $x_l[2]$, $x_r[2]$, $x_l[3]$, $x_r[3]$) , y series ($y_l[1]$, $y_r[1]$, $y_l[2]$, $y_r[2]$, $y_l[3]$, $y_r[3]$) and get the optimal region of (x_{opt} , y_{opt}) in shadowed region.

3.1.2. Penalty On Overlap

If we get the optimal region, we will compute all the possible swap target in this optimal region, rank these possible targets and pick the best of them as our good target. In order to rank them, we have to make a criterion. So we introduce the penalty on overlap for quantification.

If we swap two cells that are not of the same size, the space at the smaller cell may not be enough to hold the bigger cell. Also, if we swap a cell with a space, the space may be smaller than the cell. Both cases may lead to an overlap after swapping. Here, we use the idea of [1], using penalty P1, which is the penalty on shifting the closest two cells to resolve overlap, and penalty P2, which denotes the penalty on shifting cells other than the closest two cells. Figure 2 illustrates how to compute the penalty. Here, w_i and w_j denotes the width of cell i and cell j. And $s_1 s_2 s_3 s_4$ denotes the width of space.

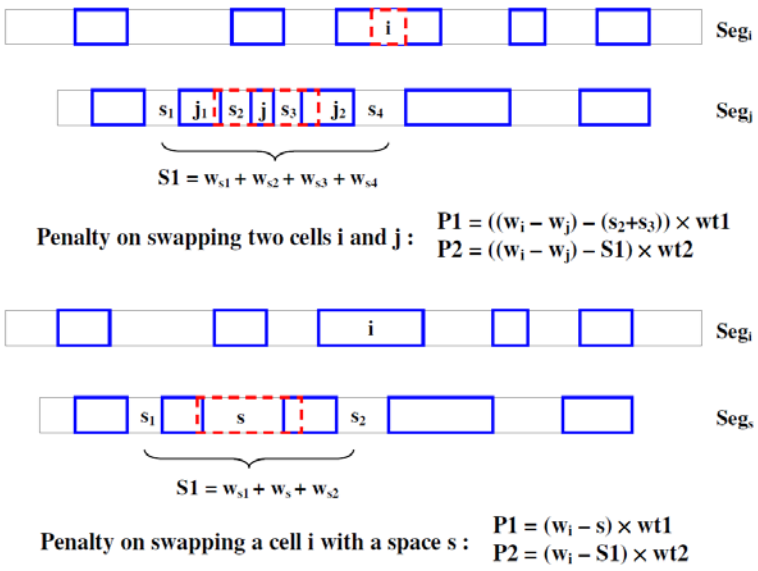

Figure 2. Computing Penalty

If we swap two cells i and j and assume $w_i > w_j$, it's possible we will introduce overlap in Seg_i. If $w_i - w_j > s_2 + s_3$, which means there is no enough space within the two closest cells (j_1 and j_2). I have to move these two closest cells to resolve the overlap. So we define $P1 = \big((w_i - w_j) - (s_2 + s_3)\big) * wt1.$

If $w_i - w_j > s_1 + s_2 + s_3 + s_4$, which means there is no enough space within the next two closest cells. I have to move cells other than the two closest cells to resolve the overlap. So we define

$$P2 = \big((w_i - w_j) - (s_1 + s_2 + s_3 + s_4)\big) * wt2.$$

Here, wt1 and wt2 are user define constant. Because we do not want to disturb the placement too much, large overlap is discouraged by setting wt2 much higher than wt1. The actual parameter is determined by the design.

3.1.3. Get Good Target In Optimal Region

In the optimal region, there may be several cells or whitespaces available for swapping. We use the criterion named “Benefit” for ranking these possible cells and whitespaces. The total “Benefit” consists of two parts: the penalty on overlap after swapping and the difference between the total wirelength before and after the swap. We use the equation:

$$B = (W1 - W2) - P1 - P2 \quad (1)$$

B denotes “Benefit”, W1 denotes the wirelength before swapping, while W2 is the wirelength after swapping. P1 and P2 are the penalty on overlap discussed above. In this criterion, the larger Benefit, the more improvement of swapping we will gain. So, in our implementation, we calculate all the benefit of possible cells and whitespaces, sort them in decreasing order, and pick the best 3 as our possible good targets.

In our implementation, we use a map to store all the possible cells’ and whitespaces’ benefit, which makes O(logN) for each insertion, O(NlogN) for total. In some small netlist input files, we notice that the optimal region for each cell is always not so large, which means the possible target number N is not a big number. Thus, the running time for traversing all the possible targets in the optimal region to find the best target is very fast. For example, And after traversing all the cells in b19 example, we count the goof swapping target number for each cell, and find 54.0% cells has no good swapping target, which means their optimal regions are themselves, 18.2% cells have only 1 good swapping target, 10.3% cells have 2 good swapping cells, and 17.5% cells have 3 or more good swapping cells. But this observation is not true for some big netlist input file, such as “netcard”. In such big netlist input file, which may contain millions of cells, the possible target number N can be thousands. So, we cannot traverse all the possible targets, which will make runtime very slow. So, we add two more user define parameters. One is the threshold of the benefit. That is, we compute a relative good swap target instead of the best one. We will stop traverse at the very time we find a possible target’s benefit satisfies our threshold. The second parameter is the “limit number”, which is the max

number of possible targets we will check in optimal region. By adding these two parameters, we gain a markedly speed up without too much performance degradation.

3.2 HPWL-Driven Swap Strategy

This strategy is based on an assumption: the best HPWL-wise position of each cell is in its optimal region. Thus, we evaluate the HPWL-wise optimality of the cell position by measuring the distance from the cell to its optimal region. When this distance is greater than our user defined constant (“x_dist” and “y_dist”), we consider that the cell’s placement with respect to HPWL is not good. And we will only swap those “poor HPWL position” cells into a random position towards its optimal region. The randomization is also determined by two use defined parameter (“x_move” and “y_move”).

This technique is applied only to the cells which are far enough from the optimal region, because otherwise it would slow down the placement speed. When this technique is applied to the cells which are relatively close to their optimal regions, it takes a long time to find swap target candidates and the swap success rate is low. However, when applied to the cells which are far enough from their optimal regions, this technique produces high swap success rate with fast runtime.[4] This rule is satisfied by assign appropriate values to our user defined constant (“x_dist” and “y_dist”). In our implementation, we set 12 and 2 to x_dist and y_dist respectively. And the other two parameters (“x_move” and “y_move”) determines the randomization we introduced into the system. The higher these two parameters, the more randomization, together with more inconsistency to the first strategy. In our implementation, x_move and y_move have positive relationship with the distance from its optimal region for a given cell.

3.3 Strategy Analysis

Our implementation employs two different global swapping strategies in order to achieve maximum solution quality improvement. The optimal-region based swap strategy tries to swap all cells into their optimal HPWL region. The HPWL-driven swap strategy identifies the cells, which are currently in a bad position HPWL wise, and tries to swap them into random position towards its optimal region.

According to our final result, optimal-region based swap strategy brings the most HPWL improvement but quickly reaches saturation. So we need HPWL-driven swap strategy to introduce randomization for further improvement. In our implementation, we interleave these two different strategies, which can maximize their different contributions. That is, we consider each strategy one followed by strategy two as a single loop. Repeat the loop again and again. End when the HPWL improvement between two consecutive loops smaller than a threshold.

4. Instant Legalization

4.1 Motivation

In our strategy the improvement comes from swapping cells for million times, many of which may not actually improve the result but instead make the result worse. Therefore, we want to check the result of each swap and decide whether to take it or not. If the result is not desirable, we can reverse the swap immediately. In order to perform result checking and reverse immediately, we need to use the technique of instant legalization, which means after every swap we perform legalization immediately, and then check the HPWL of affected nets. We nail down the result of swapping and legalization if the result is “good”, or reverse right away otherwise. Note that the criterion of “goodness” is also subject to change, which gives another parameter to tune the model and can expand the solution space by adapting a loose criterion if necessary.

4.2 Problem Formulation

The input of the instant legalization algorithm is already placed in the row, but may have some overlapping as the result of swapping two cells. Therefore, the algorithm is to place the cells within the row to eliminate overlapping.

4.3 Fast Legalization Algorithm

In our work, a fast legalization technique[3] is used for instant legalization, because we need the legalization to be very fast since it is performed after every swap. In addition, the main improvement of HPWL comes from swapping cells to optimal regions,

therefore the goal of legalization is to respect the swapping result and try to minimize the total movement of cells. That is to say, if a legalization process takes too much effort and changes the placement too much, we simply reject and reverse that swap without further carrying out the legalization process, which makes our legalization method even faster. Also, when legalizing a row, we only consider placing the cells in the same row.

The core of this approach is to optimize the total (quadratic) movement of all cells within one row. This optimization is called PlaceRow. In the following, PlaceRow is described.

4.3.1 Quadratic Programming Formulation

First, it is assumed that the row has N_r standard cells, indexed from 1 to N_r. Table 1 shows different properties of one cell i. Given is the position (of the lower left corner of the cell) in global placement ($x'(i),y'(i)$), the width w(i), and the weight e(i). The weight can be for example one, the area of the cell, or the number of pins of the cell. PlaceRow determines the legal x-position x(i) of each cell. The legal y-position should remain the same.

Moreover, it is assumed that all cells are sorted by their global x-position, i.e., $x'(i) \geq x'(i-1)$. Based on these definitions, PlaceRow is described by the following quadratic program:

$$\min \sum_{i=1}^{N_r} e(i)[x(i)-x'(i)]^2 \quad (1)$$

s.t. $$x(i)-x(i-1) \geq w(i-1), \text{ i=2,3…} \quad (2)$$

Although this quadratic program can be solved, it consumes too much time. However, if we can change the constraint to using "=" instead of ">=", then the problem can be solved relatively fast.

With only "=" constrain, the constraint now becomes:

$$x(i) = x(1) + \sum_{k=1}^{i-1} w(k), \text{ i=2, 3, } \ldots \quad (3)$$

Inserting equation (3) to equation (1) and set the derivative with respect to x(1) to zero to compute the minimum value yields the following:

$$\sum_{i=1}^{N_r} e(i)x(1) - [e(1)x'(1) + \sum_{i=2}^{N_r} e(i)[x'(i) - \sum_{k=1}^{i-1} w(k)]] = 0 \quad (4)$$

The following table shows the computation of ec, qc, and wc.

Init	**Iteration** $(i = 1, 2, ..., N_r)$
$e_c = 0$	$e_c \leftarrow e_c + e(i)$
$q_c = 0$	$q_c \leftarrow q_c + e(i)\,[x'(i) - w_c]$
$w_c = 0$	$w_c \leftarrow w_c + w(i)$

Figure.4 Computing parameters for clusters

However, if changing the constraints to allow only "=" relationship, all cells would be adjacent to each other, which is obviously not optimal and reduce the solution space significantly. Therefore, cells are divided into clusters, in which cells are adjacent to each other inside the cluster.

4.3.2 Algorithm for Computing Clusters

In the original paper, the algorithm of computing clusters is as follows:

```
   // Determine clusters and their optimal positions x_c(c):
 1 for i = 1, ..., N_r do
 2 |   c ← Last cluster;
   |   // First cell or cell i does not overlap with last
   |      cluster:
 3 |   if i = 1 or x_c(c) + w_c(c) ≤ x'(i) then
 4 |   |   Create new cluster c;
 5 |   |   Init e_c(c), w_c(c), q_c(c) to zero;
 6 |   |   x_c(c) ← x'(i);
 7 |   |   n_first(c) ← i;
 8 |   |   AddCell(c, i);
 9 |   else
10 |   |   AddCell(c, i);
11 |   |   Collapse(c);
12 |   end
13 end
```

Figure.5 Pseudo Code for Computing Clusters

Each cluster has a pointer to the cluster before it as well as the cluster right after it. Essentially, the algorithm computes the optimal position(which minimize the total movement of the cells inside the cluster), and see if the new position of the cluster still have overlaps with other clusters. If overlaps still exist, the two(or three) clusters are collapsed into a larger cluster, and the optimal position of which is recomputed. Then we check if the overlap still exists. If there is still overlap, keep collapsing the clusters until finally no overlaps exist.

4.3.3 Improved Collapsing Algorithm

Obviously, the proposed collapsing algorithm is a greedy heuristic which aims at minimizing the total movement of all the cells in the cluster. However, there is no guarantee the minimization of such objective function can yield the best results. For example, if there are two clusters c1 and c2, where c2 is the modified cluster(i.e. one of

the cells in the clusters has been swapped). After calculation, c2's new optimal position has overlap with c1. In the original algorithm, c1 and c2 will be collapsed into a large cluster c3 and be moved together. However, it is possible that c1 is actually connected to a lot of nets and moving c1 together with c2 will increase the HPWL in a lot of nets and thus increase the total HPWL. In a pessimistic situation, the cluster will keep collapsing and grow like a snowball, in which case finally all the cells in the row will be collapsed into a single cluster, which is very likely to be suboptimal, and the computation of so many collapsing is also time consuming.

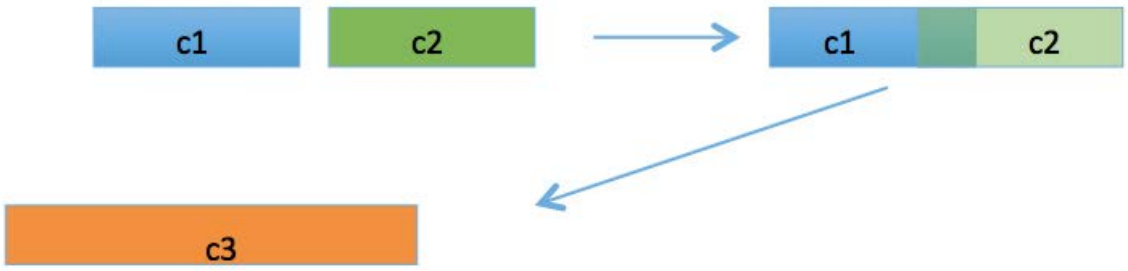

Figure.6 Original Cluster Collapsing Algorithm

Therefore, instead of collapsing the two cluster and move them together, we will modify the algorithm to avoid collapsing. There are two possible situation: (1) The new cluster can fit in the whitespace between the cluster before it and the cluster after it. In this case, if the new optimal position of the new cluster has overlap with the cluster before it, just put it adjacent to the cluster before it without collapsing. If the new optimal position has overlap with the cluster after it, put it adjacent to the cluster after it without collapsing. If the new position has no overlap, put it in the new position. (2) The new cluster cannot fit int the whitespace(this can happen when a cell in the original cluster has swapped with a larger cell), perform collapsing using the original algorithm proposed in the paper.

Note that the modification will utilize the benefits of the original algorithm but reduce the greediness. This is because our algorithm will run several passes. If the optimal solution is to move both c1 and c2 to the computed position, in our new method this can still be achieved in the second pass. Because during the second pass, c1 and c2 will be computed as one cluster due to their adjacent position and will be moved together if the result is beneficial.

5. Run Time Optimization

5.1 Lazy update of swapping and legalization

The data structure in our algorithm provides fast search operation, but the update operation is relatively slow(more detail in the next section). Therefore, we will delay the modification of the data structure whenever we can.

In addition, when performing swapping and instant legalization, more than two thirds of the operation gives undesirable results and thus needs to be reversed.

Given consideration of the above facts, we calculate the “profit” of the current swap in a temporary data structure(a linked list of clusters, which don’t need to support much operation, but only serves as a record of temporary results), without modifying the final data structure. This also makes reversing pretty fast, since we only need to discard everything in the temporary data structure.

5.2 Map data structure

In our algorithm, we need to find out the cell based on its position. In order to perform this search operation quickly, we use a map structure for each row, and a vector of maps for the entire layout. For standard cells, each standard cells is placed in only one row, therefore we can use the x position of the cell as the key, and a pointer to the cell as the value, to form the key-value pair of the map. For macros that occupy several rows, because they are fixed and cannot be moved, we can store them as several fixed single row cells in our map structure. Since there is no two cell of the same x position in the

same row, the map structure can support search operation in O(log n) time, which is almost as good as constant time. However, maintaining such a map structure can be time consuming, because after moving the cells, the key of the map(x position of the cells) needs to be updated. Since the map of the c++ standard library is implemented using balanced binary search tree, the only way to update the key is to delete the original key-value pair and insert new ones, which will involve a lot of modification of the tree structure (rotation of the subtrees, etc). Such operation can be relatively slow. Therefore, faster way to update the map structure is needed.

5.3 Change_key Operation

After studying the collapsing algorithm carefully, one can see that it has a very nice property, i.e, the relative position of the cells remains basically the same(except for the one or two swapped cells). In other words, the structure of the balanced binary search tree inside the map remains the same. That is to say, even if we delete the original nodes and insert new nodes, we will get the same structure as before, except that now the keys of the new nodes has been changed. Therefore, it is actually not necessary to perform the delete and insert operation, if we can hack the c++ standard library to make it support change_key operation.

As a result, we modify the c++ standard libray to add a new method called change_key, and use it to update the map structure if a swap is beneficial. This modification has reduced the runtime significantly.

5.4 Time Complexity analysis of Our Algorithm

As for the legalization method, under extreme situation each pass takes O(n) time, n is the number of cells in one row. This happens when the whole row collapse into a single cluster, which is unlikely in our new algorithm, but happens frequently in the original algorithm. If we assume that the number of cells and clusters being collapsed is constant, and the nets connecting those cells are also constant(i.e has a limited upper bound), the runtime of the legalization method is O(log n), which is the time of the search operation.

6. Results and Analysis

The following table illustrates the results we have on all the ICCAD 2014 test benches. Among the test cases, mgc_edit_dist, mgc_matrix_mult, netcard have fixed macros while others do not. For both cases, our algorithm gives correct and legal results. The average improvement is 2.27%. The runtime for smaller test benches is reasonable, for each cell it takes on average 2.4ms to process in the linux virtual machine on our laptop. The following shows the result and the machine configuration.

	Cell number	**Displace-ment(um)**	**Original HPWL(um)**	**Result HPWL(um)**	**Improve-ment**	**Run Time(s)**
b19	219268	20, 200	332115710	321858570	3.115%	437
vga_lcd	164891	10, 200	417944050	409862760	1.934%	615
mgc_edit_dist	130674	30, 200	505261940	488376860	3.342%	346
mgc_matrix_mult	155341	30, 200	293728200	285863840	2.678%	214
leon2	794286	40, 400	3194151280	3160655630	1.049%	16103
leon3mp	649191	30, 300	1501104830	1466888960	2.279%	16053
netcard	958792	50, 400	4118594110	4056218350	1.515%	18808

Table.1 Results on Test Benches

Hardware:
CPU: Intel® Core(TM) i7-4700HQ CPU @ 2.40GHz Memory: main memory 8G
VMware on Windows System:
OS: Ubuntu 14.04 Memory: 4G Processors #: 4 GCC version: gcc (GCC) 4.4.7

Table.2 Machine and Virtual Machine Configuration

Given consideration that the test benches of ICCAD 2014 has already been optimized for HPWL, and that other teams has an average improvement of roughly 1%, the performance of our algorithm is pretty good.

As for runtime, our algorithm has some problem dealing with the large test benches（4 to 5 hours）, but still runs faster than most of other teams.

7. Possible Improvement

Judging from the result, the largest drawback of our algorithm is the runtime of processing large test benches. It doesn't scale accordingly. One possible reason is that the time for computing the HPWL during each pass is increased tremendously, since in larger test benches there may be less whitespace, and each cell may also be connected to more nets. Therefore, when calculating the influence of collapsing, the computation for the HPWL affected is increased tremendously. Accordingly, two possible improvement can be made.

First, we may use some method to roughly predict the HPWL after collapsing without calculating the real HPWL, this will hopefully reduce the amount of calculation per swap, but the accuracy may be impaired.

Second, for now we scan all the cells and swap them(and probably reverse the swap)during each pass, and we perform several passes for each test bench. Therefore the amount of swapping is huge. One possible improvement is to have some algorithm to identify the cells that needs to be swapped, and swap those cells only. This should reduce the total swap numbers and thus the total runtime is reduced.

8. Reference:

[1] Goto S. An efficient algorithm for the two-dimensional placement problem in electrical circuit layout[J]. Circuits and Systems, IEEE Transactions on, 1981, 28(1): 12-18.

[2] Pan M, Viswanathan N, Chu C. An efficient and effective detailed placement algorithm[C]//Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on. IEEE, 2005: 48-55.

[3] Spindler P, Schlichtmann U, Johannes F M. Abacus: fast legalization of standard cell circuits with minimal movement[C]//Proceedings of the 2008 international symposium on Physical design. ACM, 2008: 47-53.

[4] Popovych S, Lai H H, Wang C M, et al. Density-aware Detailed Placement with Instant Legalization[C]//Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference. ACM, 2014: 1-6.

9. Appendix

A. get_target.h

```
#ifndef GET_TARGET_H_
#define GET_TARGET_H_

#include "class_define.h"
ofstream ffout;
const int x_far = 12;
const int y_far = 12;
const int x_move_random = 64;
const int y_move_random =16;

const int x_far_3 = 12;
const int y_far_3 = 1;
//const int x_move_random_3 = 32;
//const int y_move_random_3 =2;
double possibility = 1.0;                //the larger, the more likely swap


void get_optimal_region(cell* ptr_cell, int& xl, int& xr, int& yl, int& yh){
   multiset<int> x_series;
   multiset<int> y_series;

   int xl_temp, xr_temp, yl_temp, yh_temp;

   for(auto ptr_net = (ptr_cell->list_net).begin(); ptr_net != (ptr_cell->list_net).end(); ++ptr_net){
         (*ptr_net)->bounding_exclude(ptr_cell, xl_temp, xr_temp, yl_temp, yh_temp);
         //xl_temp = xr_temp = yl_temp = yh_temp = 16;
         x_series.insert(xl_temp);
         x_series.insert(xr_temp);
         y_series.insert(yl_temp);
         y_series.insert(yh_temp);
   }

         //original same as paper
   int x_size_half = x_series.size() / 2;
   int y_size_half = y_series.size() / 2;
   auto x_begin = x_series.begin();
   auto y_begin = y_series.begin();
   std::advance(x_begin, x_size_half - 1);
   xl = *x_begin;
   ++x_begin;
   xr = *x_begin;
   std::advance(y_begin, y_size_half - 1);
   yl = *y_begin;
   ++y_begin;
   yh = *y_begin;
}

void get_s(map<int, cell*>::iterator current_cell, int& s1, int& s2, int& s3, int& s4, map<int,
cell*>::iterator be, map<int, cell*>::iterator en){
         auto current_copy1 = current_cell;
```

```
	auto current_copy2 = current_cell;
	if(current_copy1 == be){
		s1 = s2 = 0;
		++current_copy2;
		s3 = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
		++current_copy1; ++current_copy2;
		s4 = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
		return;
	}
	--current_copy1;
	if(current_copy1 == be){
		s1= 0;
		s2 = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
		++current_copy1; ++current_copy2;
		s3 = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
		++current_copy1; ++current_copy2;
		s4 = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
		return;
	}
	--current_copy1; --current_copy2;

	s1 = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
	++current_copy1; ++current_copy2;
	s2 = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
	++current_copy1; ++current_copy2;
	if(current_copy2 == en){
		s3 = s4 = 0;
		return;
	}
	s3 = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
	++current_copy1; ++current_copy2;
	if(current_copy2 == en){
		s4 = 0;
		return;
	}
	s4 = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
}

void get_s_space(map<int, cell*>::iterator current_cell, int& s1, int& s2, int& s,map<int, cell*>::iterator be, map<int, cell*>::iterator en){
	auto current_copy1 = current_cell;
	auto current_copy2 = current_cell;
	if(current_copy1 == be){
		s1 = 0;
		++current_copy2;
		s = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
```

```
                ++current_copy1; ++current_copy2;
                s2 = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
                return;
        }
        --current_copy1;
        s1 = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
        ++current_copy1; ++current_copy2;
        if(current_copy2 == en){
                s = s2 = 0;
                return;
        }
        s = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
        ++current_copy1; ++current_copy2;
        if(current_copy2 == en){
                s2 = 0;
                return;
        }
        s2 = current_copy2->second->x_pos - current_copy1->second->x_pos - current_copy1->second->width;
}

void get_W(cell* current_cell, cell* ptr_cell, int& W1, int& W2){
        current_cell->x_pos = ptr_cell->x_pos_old;
        current_cell->y_pos = ptr_cell->y_pos_old;
        ptr_cell->x_pos = current_cell->x_pos_old;
        ptr_cell->y_pos = current_cell->y_pos_old;

        W1 = 0; W2 = 0;
        for(auto c: current_cell->list_net)
                W1 += c->get_hpwl_old();
        for(auto c: ptr_cell->list_net)
                W1 += c->get_hpwl_old();
        for(auto c: current_cell->list_net)
                W2 += c->get_hpwl_now();
        for(auto c: ptr_cell->list_net)
                W2 += c->get_hpwl_now();

        current_cell->reverse();
        ptr_cell->reverse();
}

void get_W_space(cell* current_cell, cell* ptr_cell, int& W1, int& W2){
        ptr_cell->x_pos = current_cell->x_pos_old + current_cell->width;
        ptr_cell->y_pos = current_cell->y_pos_old;

        W1 = 0; W2 = 0;
        for(auto c: ptr_cell->list_net)
                W1 += c->get_hpwl_old();
        for(auto c: ptr_cell->list_net)
                W2 += c->get_hpwl_now();

        ptr_cell->reverse();
}
```

```
void get_penalty(map<int, cell*>::iterator current_cell, cell* ptr_cell, multimap<double, point>& temp){
	double penalty;		//the smaller, the better
	int W1, W2;
	get_W(current_cell->second, ptr_cell, W1, W2);
	double P1, P2;

	point my_point{current_cell->second->x_pos, current_cell->second->y_pos};

		if(current_cell->second->width == ptr_cell->width){
			penalty = W2 - W1;
		}
		if(current_cell->second->width < ptr_cell->width){
			int s1, s2, s3, s4;
			get_s(current_cell, s1, s2, s3, s4, subrow[current_cell->second->y_pos].begin(),
subrow[current_cell->second->y_pos].end());
			int S = s1+ s2 + s3 + s4;
			P1 = (ptr_cell->width - current_cell->second->width - s2 - s3) * wt1;
			P2 = (ptr_cell->width - current_cell->second->width - S) * wt2;
			penalty = P1 + P2 + W2 -W1;
		}
		if(current_cell->second->width > ptr_cell->width){
			int row_pos = ptr_cell->y_pos;
			auto it_ptr_cell = subrow[row_pos].lower_bound(ptr_cell->x_pos);

			int s1, s2, s3, s4;
			get_s(it_ptr_cell, s1, s2, s3, s4, subrow[row_pos].begin(),
subrow[row_pos].end());
			int S = s1+ s2 + s3 + s4;
			P1 = (current_cell->second->width - ptr_cell->width - s2 - s3) * wt1;
			P2 = (current_cell->second->width - ptr_cell->width - S) * wt2;
			penalty = P1 + P2 + W2 -W1;
		}
		if(penalty < 0){
			temp.insert(std::pair<double, point>(penalty, my_point));
		}
}

void get_penalty_space(map<int, cell*>::iterator current_cell, cell* ptr_cell, multimap<double, point>&
temp){
	double penalty;
	int W1, W2;
	get_W_space(current_cell->second, ptr_cell, W1, W2);

	point my_point{current_cell->second->x_pos + current_cell->second->width, current_cell-
>second->y_pos};

		int s1,s2,s;
		get_s_space(current_cell, s1, s2, s, subrow[current_cell->second->y_pos].begin(),
subrow[current_cell->second->y_pos].end());
		int S = s1 + s2 + s;
		double P1 = (ptr_cell->width - s) * wt1;
		double P2 = (ptr_cell->width - S) * wt2;
		penalty = P1 + P2 + W2 -W1;

		penalty = W2 -W1;
```

```
                    if(penalty < 0){
                            temp.insert(std::pair<double, point>(penalty, my_point));
                    }
}

vector<point> get_swap_target(cell* ptr_cell, int& xl, int& xr, int& yl, int& yh){
        multimap<double,point> temp;
        vector<point> good_target;
        int limit = 0;
        for(int y_row = std::max<int>(yl, ptr_cell->y_pos_origin - y_displacement); y_row <
std::min<int>(yh, ptr_cell->y_pos_origin + y_displacement) + 1; ++y_row){

                    int x_lower_bound = std::max<int>(xl, ptr_cell->x_pos_origin - x_displacement);
                    int x_upper_bound = std::min<int>(xr, ptr_cell->x_pos_origin + x_displacement);
                    if (x_upper_bound < x_lower_bound) break;

                    auto start_cell = subrow[y_row].lower_bound(x_lower_bound);
                    auto end_cell = subrow[y_row].lower_bound(x_upper_bound);

                    for(auto current_cell = start_cell; (current_cell != end_cell) && limit < limit_num;
++current_cell){
                            ++limit;
                            if(current_cell->second == ptr_cell) continue;
                            get_penalty(current_cell, ptr_cell, temp);
                            auto next_cell = current_cell;
                            ++next_cell;
                            if(next_cell == subrow[y_row].end())                    //next also should not be
end()
                                    break;
                            if(next_cell->second->x_pos > (current_cell->second->x_pos + current_cell-
>second->width))//avoid two cell is a cluster
                                    get_penalty_space(current_cell, ptr_cell, temp);
                    }
        }
        auto ptr_temp = temp.begin();
        good_target.clear();
        for(int i = 0; (i < good_num) && (ptr_temp != temp.end()); ++i){
                    good_target.push_back(ptr_temp->second);
                    ++ptr_temp;
        }
        return good_target;
}

vector<point> get_target(cell* ptr_cell){
        int xl, xr, yl, yh;
        get_optimal_region(ptr_cell, xl, xr, yl, yh);
        return get_swap_target(ptr_cell, xl, xr, yl, yh);
}

bool far_from_optimal(cell* ptr_cell){
        int xl, xr, yl, yh;
        get_optimal_region(ptr_cell, xl, xr, yl, yh);
        if(xl - ptr_cell->x_pos > x_far) return true;
        if(ptr_cell->x_pos - xr > x_far) return true;
        if(yl - ptr_cell->y_pos > y_far) return true;
```

```
		if(ptr_cell->y_pos - yh > y_far) return true;

		return false;
}
vector<point> get_target_s2(cell* ptr_cell){
		vector<point> xx;
		xx.clear();

		if(!far_from_optimal(ptr_cell))
			return xx;

		srand((unsigned)time(NULL));
		int x = rand() % x_move_random - x_move_random / 2 + ptr_cell->x_pos;
		if(x < ptr_cell->x_pos_old - x_displacement) x = ptr_cell->x_pos_old - x_displacement;
		if(x > ptr_cell->x_pos_old + x_displacement) x = ptr_cell->x_pos_old + x_displacement;
		if(x <= 0) x = 0;
		if(x >= column_num - 1) x = column_num - 1;

		srand((unsigned)time(NULL));
		int y = rand() % y_move_random - y_move_random / 2 + ptr_cell->y_pos;
		if(y < ptr_cell->y_pos_old - y_displacement) y = ptr_cell->y_pos_old - y_displacement;
		if(y > ptr_cell->y_pos_old + y_displacement) y = ptr_cell->y_pos_old + y_displacement;
		if(y <= 0) y = 0;
		if(y >= row_num - 1) y = row_num - 1;


		if(x != ptr_cell->x_pos || y != ptr_cell->y_pos){
			point my_point{x, y};
			xx.push_back(my_point);
		}
		return xx;
}
vector<point> get_target_s3(cell* ptr_cell){
		vector<point> xx;
		xx.clear();

		int xl, xr, yl, yh;
		get_optimal_region(ptr_cell, xl, xr, yl, yh);

		int x_center = (xl + xr) / 2;
		int y_center = (yl + yh) / 2;

		int x = -1;
		int y = -1;

		if(xl - ptr_cell->x_pos > x_far_3){
			srand((unsigned)time(NULL));
			//x = rand() % x_move_random_3 + ptr_cell->x_pos;
			x = rand() % (2 * (x_center - ptr_cell->x_pos)) + ptr_cell->x_pos + 1;
			if(x < ptr_cell->x_pos_old - x_displacement) x = ptr_cell->x_pos_old - x_displacement;
			if(x > ptr_cell->x_pos_old + x_displacement) x = ptr_cell->x_pos_old + x_displacement;
			if(x <= 0) x = 0;
			if(x >= column_num - 1) x = column_num - 1;

		}
		else if(ptr_cell->x_pos - xr > x_far_3){
```

```
                srand((unsigned)time(NULL));
                //x = ptr_cell->x_pos - rand() % x_move_random_3;
                x = ptr_cell->x_pos - rand() % (2 * (x_center - ptr_cell->x_pos)) - 1;
                if(x < ptr_cell->x_pos_old - x_displacement) x = ptr_cell->x_pos_old - x_displacement;
                if(x > ptr_cell->x_pos_old + x_displacement) x = ptr_cell->x_pos_old + x_displacement;
                if(x <= 0) x = 0;
                if(x >= column_num - 1) x = column_num - 1;

        }

        if(yl - ptr_cell->y_pos > y_far_3){
                srand((unsigned)time(NULL));
                //y = rand() % y_move_random_3 + ptr_cell->y_pos;
                y = rand() % (2 * (y_center - ptr_cell->y_pos)) + ptr_cell->y_pos + 1;
                if(y < ptr_cell->y_pos_old - y_displacement) y = ptr_cell->y_pos_old - y_displacement;
                if(y > ptr_cell->y_pos_old + y_displacement) y = ptr_cell->y_pos_old + y_displacement;
                if(y <= 0) y = 0;
                if(y >= row_num - 1) y = row_num - 1;

        }
        else if(ptr_cell->y_pos - yh > y_far_3){
                srand((unsigned)time(NULL));
                //y = ptr_cell->y_pos - rand() % y_move_random_3;
                y = ptr_cell->y_pos - rand() % (2 * (y_center - ptr_cell->y_pos)) - 1;
                if(y < ptr_cell->y_pos_old - y_displacement) y = ptr_cell->y_pos_old - y_displacement;
                if(y > ptr_cell->y_pos_old + y_displacement) y = ptr_cell->y_pos_old + y_displacement;
                if(y <= 0) y = 0;
                if(y >= row_num - 1) y = row_num - 1;

        }

        if(x == -1 && y != -1 ){
                point my_point{ptr_cell->x_pos, y};
                xx.push_back(my_point);
        }
        if(x != -1 && y == -1){
                point my_point{x, ptr_cell->y_pos};
                xx.push_back(my_point);
        }
        if(x != -1 && y != -1){
                point my_point{x, y};
                xx.push_back(my_point);
        }


        srand((unsigned)time(NULL));
        double possi = rand() / (double)RAND_MAX;
        if(possi <  (1 - possibility))
                xx.clear();
        return xx;

}

#endif /* GET_TARGET_H_ */
```

B. class_define.h

```
#ifndef CLASS_DEFINE_H_
#define CLASS_DEFINE_H_

int row_counter = 0;
int row_num;
int column_num;
int x_step;
int y_step;
int x_length;
int y_length;

int component_num;
int idnum;
int pin_num;
int net_num;
int unit_distance_micro;
/*****************************************************************************/

//const double x_displacement = 20 * 100 / 10;////////////////displacement for b19
//const double y_displacement = 200 * 100 / 200;
const double x_displacement = 10 * 100 / 10;////////////////displacement for vga
const double y_displacement = 200 * 100 / 200;
//const double x_displacement = 30 * 100 / 10;////////////////displacement for leon3mp
//const double y_displacement = 300 * 100 / 200;
//const double x_displacement = 40 * 100 / 10;////////////////displacement for leon2
//const double y_displacement = 400 * 100 / 200;
//const double x_displacement = 30 * 100 / 10;////////////////displacement for mgc_edit
//const double y_displacement = 200 * 100 / 200;
//const double x_displacement = 30 * 100 / 10;////////////////displacement for mgc_matrix
//const double y_displacement = 200 * 100 / 200;
//const double x_displacement = 50 * 100 / 10;////////////////displacement for netcard
//const double y_displacement = 400 * 100 / 200;

const double wt1 = 0;////////////////////////////////get target: wt2 much higher than wt1
const double wt2 = 0;

//double pre_thresh=100;

const int good_num = 1;////////////////////////////good number in vector<point>
const int limit_num = 10000;

#include <iostream>
#include <list>
#include <string>
#include <vector>
#include <unordered_map>
#include <cmath>
//#include <map>
#include "map_change_key"
#include <limits.h>
#include <set>
#include <iterator>
#include <fstream>
```

```
#include <stdlib.h>
#include <algorithm>
#include <float.h>
#include <time.h>
#include <stdlib.h>

using namespace std;

class cluster;
class net;

class cell{
public:
   string cell_name;
   int x_pos;
   int y_pos;
   int width;
   double width_d;
   int height;
   double height_d;
   bool fixed;                  //1 - fixed
   bool direction; //1 - input; 0 - output;    just make sense to pin
   cluster* ptr_cluster;
   list<net*> list_net;
   int x_pos_origin;
   int y_pos_origin;
   int x_pos_old;
   int y_pos_old;
   bool is_pin;

   cell(string cell_name, int x_pos, int y_pos, int width, double width_d, int height, double height_d, bool
fixed, bool direction, bool is_pin): cell_name(cell_name), x_pos(x_pos), y_pos(y_pos), width(width),
width_d(width_d), height(height), height_d(height_d), fixed(fixed), direction(direction),
x_pos_origin(x_pos), y_pos_origin(y_pos), x_pos_old(x_pos), y_pos_old(y_pos), is_pin(is_pin){};

   bool check_displacement(){
      bool x_check=abs(x_pos - x_pos_origin) <= x_displacement;
      bool y_check=abs(y_pos - y_pos_origin) <= y_displacement;
      return (x_check) && (y_check);
   }
   void reverse(void){
      x_pos=x_pos_old;
      y_pos=y_pos_old;
   }
   void uniform(void){
      x_pos_old=x_pos;
      y_pos_old=y_pos;
   }

};

class point{
public:
   int x;
   int y;
   int w2w1;
```

```
    point(int x, int y): x(x), y(y){};
};

std::ostream& operator<<(ostream& out, point xx){
    out<<"("<<xx.x<<" , "<<xx.y<<")"<<endl;
    return out;
}

class net{
public:
    string net_name;
    list<cell*> list_cell;
    net(string net_name): net_name(net_name){};
    void reverse(void){
        for(auto c:list_cell){
            c->reverse();
        }
    }
    void uniform(void){
        for(auto c:list_cell){
            c->uniform();
        }
    }

            double get_hpwl_now(){
                    double x_pos_min = DBL_MAX;
                    double x_pos_max = 0;
                    double y_pos_min = DBL_MAX;
                    double y_pos_max = 0;
                    for(auto ptr = list_cell.begin(); ptr != list_cell.end(); ++ptr){
            double x_center = (*ptr)->x_pos * x_step + (*ptr)->width * x_step / 2.0;
            double y_center = (*ptr)->y_pos * y_step + (*ptr)->height * y_step / 2.0;;

                                if( x_center > x_pos_max )          x_pos_max = x_center;
                                if( x_center < x_pos_min )x_pos_min = x_center;
                                if( y_center > y_pos_max )          y_pos_max = y_center;
                                if( y_center < y_pos_min )y_pos_min = y_center;
                    }
                    return (x_pos_max - x_pos_min)  + (y_pos_max - y_pos_min) ;
            }

    double get_hpwl_old(){
                    double x_pos_min = DBL_MAX;
                    double x_pos_max = 0;
                    double y_pos_min = DBL_MAX;
                    double y_pos_max = 0;
                    for(auto ptr = list_cell.begin(); ptr != list_cell.end(); ++ptr){
            double x_center = (*ptr)->x_pos_old * x_step + (*ptr)->width * x_step / 2.0;
            double y_center = (*ptr)->y_pos_old * y_step + (*ptr)->height * y_step / 2.0;;

                                if( x_center > x_pos_max )          x_pos_max = x_center;
                                if( x_center < x_pos_min )x_pos_min = x_center;
                                if( y_center > y_pos_max )          y_pos_max = y_center;
                                if( y_center < y_pos_min )y_pos_min = y_center;
                    }
                    return (x_pos_max - x_pos_min)  + (y_pos_max - y_pos_min) ;
```

```
    }

    void bounding_exclude(cell* ptr_cell, int& xl, int& xr, int& yl, int& yh){
        xl = INT_MAX;
        xr = 0;
        yl = INT_MAX;
        yh = 0;
        for(auto ptr = list_cell.begin(); ptr != list_cell.end(); ++ptr){
            int x_cen = (*ptr)->x_pos + (*ptr)->width / 2;
            int y_cen = (*ptr)->y_pos + (*ptr)->height / 2;


            if(ptr_cell != (*ptr)){
                if( x_cen > xr )   xr = x_cen;
                if( x_cen < xl )   xl = x_cen;
                if( y_cen > yh )   yh = y_cen;
                if( y_cen < yl )   yl = y_cen;
            }
        }
        xr -= ptr_cell->width / 2;
        xl += ptr_cell->width / 2;
        yl -= ptr_cell->height / 2;
        yh += ptr_cell->height / 2;
    }
};

class cluster{
public:
    int width;  //width
    int e;      //weight
    int q;
    int x1;//the left most position of cluster
    bool fixed;
    bool abandoned;
    list<cell*> celllist;
    int myid;
    int y_pos;

    cluster(void):width(0),e(0),y_pos(-
1),q(0),x1(0),myid(idnum),fixed(false),abandoned(false),prev(nullptr),next(nullptr){
        if(idnum==10851)
            cout<<""<<endl;
        ++idnum;
    }
    void reverse(void){
        for(auto cit:celllist){
            cit->reverse();
        }
    }
    void uniform(void){
        for(auto cit:celllist){
            cit->uniform();
        }
    }

    cluster(cluster&
```

```
rhs):width(rhs.width),e(rhs.e),q(rhs.q),x1(rhs.x1),myid(idnum),fixed(rhs.fixed),abandoned(false),prev(rhs.p
rev),next(rhs.next),celllist(rhs.celllist),y_pos(rhs.y_pos){
        if(idnum==10851)
           cout<<""<<endl;
        ++idnum;
    }

    bool update_cell(void){
        int w=0;
        for(auto it:celllist){
           if(it!=nullptr){
              it->x_pos=x1+w;
              it->y_pos=y_pos;
              w+=it->width;
              if(!it->check_displacement()) return false;
           }
        }
        return true;
    }

    cluster* prev;
    cluster* next;

};
/*************************************************************************************
**/
vector<map<int,cell*>> subrow;

cell* vec_cell;
int vec_cell_counter = 0;
net* vec_net;
int vec_net_counter = 0;


#endif /* CLASS_DEFINE_H_ */
```

C. result_func.h

```
#ifndef RESULT_FUNC_H_
#define RESULT_FUNC_H_
#include "class_define.h"

bool check_legal(){
	int** globe_map = new int*[row_num];
	for(int i = 0; i < row_num; ++i)
		globe_map[i] = new int[column_num];
	for(int i = 0; i < row_num; ++i)
		for(int j = 0; j < column_num; ++j)
			globe_map[i][j] = 0;

	for(int i = 0; i < vec_cell_counter; ++i){
		if(!(vec_cell + i)->direction){
			int row_t = (vec_cell + i)->y_pos;
			int column_t = (vec_cell + i)->x_pos;
			int width_t = (vec_cell + i)->width;
			int height_t = (vec_cell + i)->height;

			for(int hi = 0; hi < height_t; ++hi){
				for(int wi = 0; wi < width_t; ++wi){
					++globe_map[row_t + hi][column_t + wi];
					if(globe_map[row_t + hi][column_t + wi] > 1){
						for(int i = 0; i < row_num; ++i)
							delete[] globe_map[i];
						delete[] globe_map;
						return false;
					}
				}
			}
		}
	}
	for(int i = 0; i < row_num; ++i)
		delete[] globe_map[i];
	delete[] globe_map;
	return true;
}

double total_hpwl(){
	double result = 0;
	for(int i = 0; i < net_num; ++i)
		result += (vec_net + i)->get_hpwl_now();
	return result;
}

void display_time(){
	time_t t=time(0);
	struct tm* now=localtime(&t);
	cout<<"now   time: "<<now->tm_hour<<":"<<now->tm_min<<":"<<now->tm_sec<<endl;
}

#endif /* RESULT_FUNC_H_ */
```

D. placerow.h

```
#ifndef PLACEROW_H_
#define PLACEROW_H_


#include "class_define.h"

using std::vector;
using std::string;
using std::unordered_map;
using std::list;
using std::map;
using std::ostream;
using std::cout;
using std::endl;

int whitespaceswap=0;
int samerow_non_whitespace=0;

/*** swap criterion
   if new_hpwl<old_hpwl + thresh, it is a "good" swap
 ***/

double thresh=50;


class row_placer{
public:
   vector<map<int,cell*>>& grid;
   int x_size;
   int y_size;
   int x_max;
   int sitewidth;
   int decrease_count;
   int increase_count;
   int swapcount;
   int displacement;
   bool merged;
   cluster* c_head, * c_tail, * t_head, * t_tail;

   vector<cluster*> cluster_v;

   void check(cluster* clp){
      if(c_head!=nullptr&&c_head==t_head){
         cout<<""<<endl;
      }
   }

   void initialize(void){
      int i=0;
      for(auto row:grid){
         auto b=row.begin();
         auto e=row.end();
         int lastx=0;
```

```
        cluster* clu_p;
        cluster* head=nullptr;
        cluster* current=nullptr;
        int cluster_count=0;
        while(b!=e){
           if(head==nullptr||((*b).second)->fixed==true||lastx!=((*b).second)-
>x_pos||(clu_p!=nullptr&&clu_p->fixed==true)){
              clu_p=new cluster();
              if(((*b).second)->fixed==true)
                 clu_p->fixed=true;
              clu_p->y_pos=i;
              if(((*b).second)->y_pos!=i){
                 cout<<"error in initialization"<<endl;
              }
              ++cluster_count;
              clu_p->e=1;
              clu_p->x1=((*b).second)->x_pos;//the x coordinate of the first cell
              clu_p->q=((*b).second)->x_pos-clu_p->width;
              clu_p->width=((*b).second)->width; //the width of cell, the name may change
              clu_p->celllist.push_back((*b).second);
              ((*b).second)->ptr_cluster=clu_p;
              lastx=((*b).second)->x_pos+((*b).second)->width;
              if(head==nullptr){
                 head=clu_p;
                 current=clu_p;
              }else{
                 current->next=clu_p;
                 clu_p->prev=current;
                 current=clu_p;
              }
           }
           else{
              ++(clu_p->e);
              clu_p->q+=clu_p->e*(((*b).second)->x_pos-clu_p->width);
              clu_p->width+=((*b).second)->width;
              clu_p->celllist.push_back((*b).second);
              ((*b).second)->ptr_cluster=clu_p;
              lastx+=((*b).second)->width;
           }
           ++b;
        }
        ++i;
    //     cout<<"this row has "<<cluster_count<<" clusters"<<endl;
        cluster_v.push_back(head);


     }
  }

  row_placer(vector<map<int,cell*>>& table, int x_s, int y_s, int unit):displacement(0),grid(table),
x_size(x_s),y_size(y_s),sitewidth(unit){
     x_max=x_size*sitewidth;
     decrease_count=0;
     increase_count=0;
     swapcount=0;
     idnum=0;
```

```
    initialize();
    cluster_v.reserve(y_size);

}

bool double_collapse_c(cluster* clup){
    cluster* prevc=clup->prev;
    cluster* nextc=clup->next;
    if(clup==nullptr){
        cout<<""<<endl;
    }
    if(clup->x1<0||clup->x1+clup->width>x_size) return false;

    bool forward_collapse=(prevc!=nullptr)&&(clup->x1<=prevc->x1+prevc->width);
    bool backward_collapse=(nextc!=nullptr)&&((clup->x1+clup->width)>=nextc->x1);

    if(forward_collapse&&backward_collapse){
        /******* copy prevc and nextc and merge them with clup   *******/
        if(prevc->fixed==1||nextc->fixed==1) return false;

        /*** new code **/
        clup->e=0;
        clup->q=0;
        clup->width=0;
        list<cell*> temp=prevc->celllist;
        temp.splice(temp.end(), clup->celllist);
        clup->celllist=std::move(temp);
        list<cell*> temp1=nextc->celllist;
        clup->celllist.splice(clup->celllist.end(), temp1);

        for(auto c_it:clup->celllist){
            ++clup->e;
            clup->q+=c_it->x_pos-c_it->width;
            clup->width+=c_it->width;

        }
        /*** end of new code ***/
        clup->x1=clup->q/clup->e;
        if(clup->x1>x_size-clup->width)
            clup->x1=x_size-clup->width;
        if(clup->x1<0)
            clup->x1=0;
        /******** update c_head, c_tail  **********/
        c_head=prevc;
        c_tail=nextc;
        /***** clup points to new neighbours, but neighbours still points to original clusters ****/
        clup->prev=prevc->prev;
        clup->next=nextc->next;

        return double_collapse_c(clup);
    }else if(forward_collapse){
        /******** copy prevc and merge it with clup *********/
        if(prevc->fixed==1) return false;

         /** new code to prevent collapse **/
        int nextx;
```

```
    int prevx=prevc->x1+prevc->width;

    if(nextc==nullptr) nextx=x_size;
    else nextx=nextc->x1;

    if(nextx-prevx>=clup->width){
       clup->x1=prevx;
       return true;
    }
    /**** end of new code ****/

    /*** new code to calculate x1 ***/
    clup->e=0;
    clup->q=0;
    clup->width=0;
    list<cell*> temp=prevc->celllist;
    temp.splice(temp.end(), clup->celllist);
    clup->celllist=std::move(temp);
    for(auto c_it:clup->celllist){
       ++clup->e;
       clup->q+=c_it->x_pos-c_it->width;
       clup->width+=c_it->width;

    }
    /*** end of new code ***/
    clup->x1=clup->q/clup->e;
    if(clup->x1>x_size-clup->width)
       clup->x1=x_size-clup->width;
    if(clup->x1<0)
       clup->x1=0;
    /****** update c_head and c_tail  ******/
    c_head=prevc;
    if(c_tail==nullptr) c_tail=c_head;
    /******** clup points to new neighbours   ********/
    clup->prev=c_head->prev;

    /** for test **/
    prevc->abandoned=true;
    /** end of test ***/

    /*** for test ****/
    check(clup);
    check(prevc);
    check(nextc);
    /**** for test ****/

    return double_collapse_c(clup);
  }else if(backward_collapse){
    /********* copy nextc and merge it with clup ******/
    if(nextc->fixed==1) return false;

    /** new code to prevent collapse**/
    int nextx=nextc->x1;
    int prevx;

    if(prevc==nullptr) nextx=0;
```

```
        else prevx=prevc->x1+prevc->width;

        if(nextx-prevx>=clup->width){
           clup->x1=nextx-clup->width;
           return true;
        }
        /**** end of new code ****/

        /*** new code **/
        clup->e=0;
        clup->q=0;
        clup->width=0;
        list<cell*> temp=nextc->celllist;

        clup->celllist.splice(clup->celllist.end(), temp);
        for(auto c_it:clup->celllist){
           ++clup->e;
           clup->q+=c_it->x_pos-c_it->width;
           clup->width+=c_it->width;

        }
        /*** end of new code ***/

        clup->x1=clup->q/clup->e;
        if(clup->x1>x_size-clup->width)
           clup->x1=x_size-clup->width;
        if(clup->x1<0)
           clup->x1=0;
        /****** update c_head and c_tail  ******/
        c_tail=nextc;
        if(c_head==nullptr) c_head=c_tail;
        /******** clup points to new neighbours   *******/
        clup->next=c_tail->next;

        /** for test **/
        nextc->abandoned=true;
        /** end of test ***/

        /*** for test ****/
        check(clup);
        check(prevc);
        check(nextc);
        /**** for test ****/

        return double_collapse_c(clup);
     }else{
        return true;
     }
  }

  bool double_collapse_t(cluster* clup){

     cluster* prevc=clup->prev;
     cluster* nextc=clup->next;
     if(clup->x1<0||clup->x1+clup->width>x_size) return false;
     bool forward_collapse=(prevc!=nullptr)&&(clup->x1<=prevc->x1+prevc->width);
```

```
bool backward_collapse=(nextc!=nullptr)&&((clup->x1+clup->width)>=nextc->x1);


/*** for test ****/
check(clup);
check(prevc);
check(nextc);
/**** for test ****/

if(forward_collapse&&backward_collapse){
   /****** copy prevc and nextc and merge them with clup  *******/
   if(prevc->fixed==1||nextc->fixed==1) return false;

   /*** new code **/
   clup->e=0;
   clup->q=0;
   clup->width=0;
   list<cell*> temp=prevc->celllist;
   temp.splice(temp.end(), clup->celllist);
   clup->celllist=std::move(temp);
   list<cell*> temp1=nextc->celllist;
   clup->celllist.splice(clup->celllist.end(), temp1);

   for(auto c_it:clup->celllist){
      ++clup->e;
      clup->q+=c_it->x_pos-c_it->width;
      clup->width+=c_it->width;

   }
   /*** end of new code ***/

   clup->x1=clup->q/clup->e;
   if(clup->x1>x_size-clup->width)
      clup->x1=x_size-clup->width;
   if(clup->x1<0)
      clup->x1=0;
   /******** update t_head, t_tail  **********/
   t_head=prevc;
   t_tail=nextc;
   /******** clup points to new neighbours   *******/
   clup->prev=prevc->prev;
   clup->next=nextc->next;

   /** for test **/
   prevc->abandoned=true;
   nextc->abandoned=true;
   /** end of test ***/

   /*** for test ****/
   check(clup);
   check(prevc);
   check(nextc);
   /**** for test ****/

   return double_collapse_t(clup);
}else if(forward_collapse){
```

```
    /****** copy prevc and merge them with clup  *******/
    if(prevc->fixed==1) return false;

    /** new code  to prevent collapse **/
    int nextx;
    int prevx=prevc->x1+prevc->width;

    if(nextc==nullptr) nextx=x_size;
    else nextx=nextc->x1;

    if(nextx-prevx>=clup->width){
       clup->x1=prevx;
       return true;
    }
    /**** end of new code ****/


    /*** new code **/
    clup->e=0;
    clup->q=0;
    clup->width=0;
    list<cell*> temp=prevc->celllist;
    temp.splice(temp.end(), clup->celllist);
    clup->celllist=std::move(temp);
    for(auto c_it:clup->celllist){
       ++clup->e;
       clup->q+=c_it->x_pos-c_it->width;
       clup->width+=c_it->width;

    }
    /*** end of new code ***/


    clup->x1=clup->q/clup->e;
    if(clup->x1>x_size-clup->width)
       clup->x1=x_size-clup->width;
    if(clup->x1<0)
       clup->x1=0;
    /******** update t_head, t_tail  **********/
    t_head=prevc;
    if(t_tail==nullptr) t_tail=t_head;
    /******** clup points to new neighbours   *******/
    clup->prev=prevc->prev;


    return double_collapse_t(clup);
  }else if(backward_collapse){
    /****** copy nextc and merge them with clup  *******/
    if(nextc->fixed==1) return false;

    /** new code  **/
    int nextx=nextc->x1;
    int prevx;
```

```
        if(prevc==nullptr) nextx=0;
        else prevx=prevc->x1+prevc->width;

        if(nextx-prevx>=clup->width){
           clup->x1=nextx-clup->width;
           return true;
        }
        /*** end of new code ***/

        /*** new code **/
        clup->e=0;
        clup->q=0;
        clup->width=0;
        list<cell*> temp=nextc->celllist;
        clup->celllist.splice(clup->celllist.end(), temp);
        for(auto c_it:clup->celllist){
           ++clup->e;
           clup->q+=c_it->x_pos-c_it->width;
           clup->width+=c_it->width;

        }
        /*** end of new code ***/


        clup->x1=clup->q/clup->e;
        if(clup->x1>x_size-clup->width)
           clup->x1=x_size-clup->width;
        if(clup->x1<0)
           clup->x1=0;
        /******** update t_head, t_tail  **********/
        t_tail=nextc;
        if(t_head==nullptr) t_head=t_tail;
        /******** clup points to new neighbours   *******/
        clup->next=nextc->next;

        /** for test **/
        nextc->abandoned=true;
        /** end of test ***/

        /*** for test ****/
        check(clup);
        check(prevc);
        check(nextc);
        /**** for test ****/

        return double_collapse_t(clup);
     }else{
        return true;
     }
  }



  void same_row_swap(int x, int y, cell* c){
     ++swapcount;
     merged=false;
```

```
                if(swapcount % 10000 == 0)
                        cout<<"swapcount is "<<swapcount<<endl;

int cell_y=c->y_pos;
int cell_x=c->x_pos;
bool whitespace;
int same_x1;
cell* t=nullptr;
map<int,cell*>& target_row=grid[y];
map<int,cell*>& cell_row=grid[c->y_pos];
unordered_map<string,net*> netlist;

c_head=nullptr;
c_tail=nullptr;
t_head=nullptr;
t_tail=nullptr;

auto b=target_row.begin();
auto e=target_row.end();

cluster* c_backup=c->ptr_cluster;
cluster* t_backup=nullptr;

/** iterator to indicate where the cell and target are in their cluster ***/
list<cell*>::iterator cell_temp, target_temp;

/**** new clusters to be passed to double_collapse and modfied ****/
cluster* cluster_new=nullptr;  // cluster_new is the cluster to replace in cell_row
cluster* target_new=nullptr; //target_new is the cluster to replace in the target_row




/****************** get all the nets involved  ****************/
while(b!=e){
   for(net* n:((*b).second)->list_net){
      if(netlist.find(n->net_name)==netlist.end()) netlist[n->net_name]=n;
   }
   ++b;
}

/******** calculate the HPWL of the netlist before swap ************/
int old_hpwl=0;
for(auto i:netlist){
   old_hpwl+=(i.second)->get_hpwl_now();
}
/****************** back up the old position of the cell ****************/
c->x_pos_old=c->x_pos;
c->y_pos_old=c->y_pos;
c->x_pos=x;
c->y_pos=y;
cluster* target_cluster=cluster_v[y];
cluster* cell_cluster=cluster_v[cell_y];
```

```
      bool legalswap=true;
      bool samecluster=false;

      if(target_row.upper_bound(x)==target_row.begin()||
        ((*(--target_row.upper_bound(x))).second)->x_pos+((*(--target_row.upper_bound(x))).second)-
>width<x){
         ++whitespaceswap;
         c->reverse();
         return ;

      }else{
         t=(*(--target_row.upper_bound(x))).second;
         if(c==t) {
            c->reverse();
            ++whitespaceswap;
            return;}  //can do better
         c_backup=c->ptr_cluster;
         t_backup=t->ptr_cluster;
         t->x_pos_old=t->x_pos;
         t->y_pos_old=t->y_pos;
         t->x_pos=cell_x;
         t->y_pos=cell_y;
         if(c_backup==t_backup){  // in the same cluster
            samecluster=true;
            target_new=new cluster(*c_backup);
            cell_temp=std::find(target_new->celllist.begin(), target_new->celllist.end(),c);
            target_temp=std::find(target_new->celllist.begin(), target_new->celllist.end(),t);
            *cell_temp=t;
            *target_temp=c;
            target_new->e=0;
            target_new->q=0;
            target_new->width=0;
            for(auto c_it:target_new->celllist){
               ++target_new->e;
               target_new->q+=c_it->x_pos-c_it->width;
               target_new->width+=c_it->width;
            }
            target_new->x1=target_new->q/target_new->e;
            if(target_new->x1<0) target_new->x1=0;
            if(target_new->x1+target_new->width>x_size) target_new->x1=x_size-target_new->width;

            cluster* prevc=target_new->prev;
            cluster* nextc=target_new->next;

            bool forward_collapse=(prevc!=nullptr)&&(target_new->x1<prevc->x1+prevc->width);
            bool backward_collapse=(nextc!=nullptr)&&((target_new->x1+target_new->width)>nextc->x1);
            if(forward_collapse)
               target_new->x1=prevc->x1+prevc->width;
            if(backward_collapse)
               target_new->x1=nextc->x1-target_new->width;
         }else{
            ++samerow_non_whitespace;
            //not in the same cluster
            c->reverse();
            t->reverse();
            return;
```

```
        }
      }

        if(legalswap){
          if(samecluster){
            if(target_new->update_cell()==false) legalswap=false;
          }
        }

        int new_hpwl=0;
        if(legalswap){
          for(auto c_it:netlist){
            new_hpwl+=(c_it.second)->get_hpwl_now();
          }
          if(new_hpwl>=old_hpwl+thresh) {
            legalswap=false;
            ++increase_count;
          }
          else{
            ++decrease_count;
          }
        }

        if(legalswap){
          if(samecluster){
            for(auto it_net:c_backup->celllist){
              it_net->uniform();
            }
            for(auto c_it:target_new->celllist){
              c_it->ptr_cluster=target_new;
            }
            if(target_new->prev!=nullptr) target_new->prev->next=target_new;
            else cluster_v[y]=target_new;
            if(target_new->next!=nullptr) target_new->next->prev=target_new;
            delete c_backup;
          }
          for(auto it_net:netlist){
            (it_net.second)->uniform();
          }

          target_row.clear();
          cluster* t_temp=cluster_v[y];
          while(t_temp!=nullptr){
            for (auto cell_ptr:t_temp->celllist){
              target_row[cell_ptr->x_pos]=cell_ptr;
            }
            t_temp=t_temp->next;
          }
//          cluster* t_temp=cluster_v[y];
//          auto row_it=target_row.begin();
//          while(t_temp!=nullptr){
//            for (auto cell_ptr:t_temp->celllist){
//              target_row.change_key(row_it,cell_ptr->x_pos);
//              ++row_it;
//            }
//            t_temp=t_temp->next;
```

```
//              }
//


        }
        else{
            if(samecluster){
                for(auto it_net:c_backup->celllist){
                    it_net->reverse();
                }
                delete target_new;
            }

        }
    }

    int swap(int x, int y, cell* c){
        ++swapcount;
                        if(swapcount % 10000 == 0)
                                cout<<"swapcount is "<<swapcount<<endl;
        int cell_y=c->y_pos;
        int cell_x=c->x_pos;
     bool whitespace;
        cell* t=nullptr;
        map<int,cell*>& target_row=grid[y];
        map<int,cell*>& cell_row=grid[c->y_pos];
        unordered_map<string,net*> netlist;

        c_head=nullptr;
        c_tail=nullptr;
        t_head=nullptr;
        t_tail=nullptr;
        /**** whether to split or not ***/
        bool c_split=false;
        bool t_split=false;

        auto b=target_row.begin();
        auto e=target_row.end();

        cluster* c_backup=c->ptr_cluster;
        cluster* t_backup=nullptr;

        /** iterator to indicate where the cell and target are in their cluster ***/
        list<cell*>::iterator cell_temp, target_temp;

        /**** new clusters to be passed to double_collapse and modfied ****/
        cluster* cluster_new=nullptr;  // cluster_new is the cluster to replace in cell_row
        cluster* target_new; //target_new is the cluster to replace in the target_row




        /******************** get all the nets involved  *****************/
        while(b!=e){
            for(net* n:((*b).second)->list_net){
                if(netlist.find(n->net_name)==netlist.end()) netlist[n->net_name]=n;
```

```
        }
        ++b;
    }
    b=cell_row.begin();
    e=cell_row.end();
    while(b!=e){
        for(net* n:((*b).second)->list_net){
            if(netlist.find(n->net_name)==netlist.end()) netlist[n->net_name]=n;
        }
        ++b;
    }
    /******** calculate the HPWL of the netlist before swap ************/
    int old_hpwl=0;
    for(auto i:netlist){
        old_hpwl+=(i.second)->get_hpwl_now();
    }
    /******************** back up the old position of the cell *****************/
    c->x_pos_old=c->x_pos;
    c->y_pos_old=c->y_pos;
    c->x_pos=x;
    c->y_pos=y;
    cluster* target_cluster=cluster_v[y];
    cluster* cell_cluster=cluster_v[cell_y];

    bool legalswap=true;

  //    if(target_row.find(x)==target_row.end()){  //target is a whitespace
    if(target_row.upper_bound(x)==target_row.begin()||
      ((*(--target_row.upper_bound(x))).second)->x_pos+((*(--target_row.upper_bound(x))).second)-
>width<=x)
    {
        whitespace=true;
        /***** put a nullptr in the cell position of c_backup ****/
        cell_temp=std::find(c_backup->celllist.begin(), c_backup->celllist.end(),c);
        *cell_temp=nullptr;   // when target and cell are in the same cluster, or probably in the same row,
could be a problem

        /****** for the cell row *****/
        c_head=nullptr;
        c_tail=nullptr;

        /***** new a cluster which is the cell itself *****/
        target_new=new cluster();
        target_new->x1=x;
        target_new->y_pos=y;
        target_new->e=1;
        target_new->width=c->width;
        target_new->q=x-c->width;
        target_new->celllist.push_back(c);
        /****** find the previous cluster and next cluster in the new position ****/

        cluster* next_cluster;
        cluster* prev_cluster;

        if(target_row.upper_bound(x)!=target_row.end()){
            next_cluster=(*(target_row.upper_bound(x))).second->ptr_cluster;
```

```
            prev_cluster=next_cluster->prev;
        }else{
            next_cluster=nullptr;
            prev_cluster=cluster_v[y];
            while(prev_cluster->next!=nullptr){
                prev_cluster=prev_cluster->next;
            }
        }

        target_new->next=next_cluster;
        target_new->prev=prev_cluster;
        t_head=nullptr;
        t_tail=nullptr;
        //only deal with the target row
        if(!double_collapse_t(target_new)) legalswap=false;

        /******* for the cellrow simply split the original cluster, which is to do nothing, for now ******/

    }
  else{
    whitespace=false;

        t=(*(--target_row.upper_bound(x))).second;
    if(t->fixed==true) {
        c->x_pos=cell_x;
        c->y_pos=cell_y;
        return 1;
    }
        t_backup=t->ptr_cluster;
        c->x_pos=t->x_pos;
        t->x_pos_old=t->x_pos;
        t->y_pos_old=t->y_pos;
        t->y_pos=cell_y;
        t->x_pos=cell_x;
        /**** new clusters to be passed to double_collapse and modfied ****/
        cluster_new=new cluster(*c_backup);
        target_new=new cluster(*t_backup);
        /******* change the cellist of the new cluster, i.e swap c and t to corresponding cluster  ******/
        cell_temp=std::find(cluster_new->celllist.begin(), cluster_new->celllist.end(), c);
        target_temp=std::find(target_new->celllist.begin(), target_new->celllist.end(), t);
        *cell_temp=t;
        *target_temp=c;
        /******* modify the new clusters and collapse them *****/

    c_head=c_backup;
    c_tail=c_backup;
    if(t->width>=c->width){
        cluster_new->width=0;
        cluster_new->e=0;
        cluster_new->q=0;
        cluster_new->x1=0;
        for(auto c_it:cluster_new->celllist){
            ++cluster_new->e;
            cluster_new->q+=c_it->x_pos-c_it->width;
            cluster_new->width+=c_it->width;
        }
```

```
        cluster_new->x1=cluster_new->q/cluster_new->e;

        if(cluster_new->x1<0)
           cluster_new->x1=0;
        if(cluster_new->x1+cluster_new->width>x_size)
           cluster_new->x1=x_size-cluster_new->width;

        if(!double_collapse_c(cluster_new))  {legalswap=false;}
      }else{
        c_split=true;
      }
      /*** modify the target clusters and collapse them  ****/

        t_head=t_backup;
        t_tail=t_backup;

        if(c->width>=t->width){
           target_new->width=0;
           target_new->q=0;
           target_new->e=0;
           for(auto c_it:target_new->celllist){
           ++target_new->e;
           target_new->q+=c_it->x_pos-c_it->width;
           target_new->width+=c_it->width;
        }
           target_new->x1=target_new->q/target_new->e;
           if(target_new->x1<0)
              target_new->x1=0;
           if(target_new->x1+target_new->width>x_size)
              target_new->x1=x_size-target_new->width;

           if(!double_collapse_t(target_new)) legalswap=false;
        }else{
           t_split=true;
        }

   }
      int new_hpwl=0;
      /***  update the cell position in the cluster ***/
      if(!legalswap) {
//          cout<<""<<endl;
;
      }
      if(legalswap){
        cluster* clup=cluster_new;
        if((!whitespace)&&(!c_split)){
           if(clup->update_cell()==false) {   legalswap=false;}
        }
      }

      if(legalswap){
        cluster* clup=target_new;
        if(!t_split)
        if(clup->update_cell()==false) {   legalswap=false;}
      }
```

```
// if(!legalswap) ++displacement;

/**** get the new hpwl from the netlist ****/
if(legalswap){
   for(auto c_it:netlist){
      new_hpwl+=(c_it.second)->get_hpwl_now();
   }
   if(new_hpwl>=old_hpwl+thresh) {
      legalswap=false;
      ++increase_count;
   }
   else{
      ++decrease_count;
   }
}



/******* the swap decreases the hpwl ***/
if(legalswap){
   /******* update cell row cluster ********/

   for(auto it_net:netlist){
      (it_net.second)->uniform();
   }


   /***** delete the original clusters in cell row *****/
   cluster* dtemp=c_head;
   if((!whitespace)&&(!c_split)){
      /*** points the other cluster to cluster_new  **/
      if(c_head->prev!=nullptr) c_head->prev->next=cluster_new;
      else cluster_v[cell_y]=cluster_new;
      if(c_tail->next!=nullptr) c_tail->next->prev=cluster_new;

      /**** update the ptr_cluster in the cellist ***/
      for(auto it: cluster_new->celllist){
         if(cluster_new->celllist.size()!=0)
         it->ptr_cluster=cluster_new;
         else{
            cout<<"bug found in line 601 "<<endl;
         }
      }
      while(dtemp!=c_tail){
         cluster* dnext=dtemp->next;
         delete dtemp;
         dtemp=dnext;
      }
      delete dtemp;
   }else if(whitespace){
      cluster* c_front=nullptr;
      cluster* c_back=nullptr;
      /****** if the target is white space, split the original cluster into two parts ****/
      auto it=c_backup->celllist.begin();
      if(c_backup->celllist.size()==1&&it==cell_temp){
       //   cout<<"here we got a problem in swapcount "<<swapcount<<endl;
```

```
                if(c_backup->prev==nullptr) { cluster_v[cell_y]=c_backup->next; c_backup->next->prev=nullptr; }
                else{
                    c_backup->prev->next=c_backup->next;
                }
                if(c_backup->next!=nullptr) c_backup->next->prev=c_backup->prev;
                delete c_backup;
            }else if(it==cell_temp||cell_temp==(--c_backup->celllist.end())){
                c_backup->celllist.erase(cell_temp);
                c_backup->width=0;
                c_backup->e=0;
                c_backup->q=0;
                int c_x1=(*(c_backup->celllist.begin()))->x_pos;
                for(auto cell_it:c_backup->celllist){
                    ++c_backup->e;
                    c_backup->q+=cell_it->x_pos-cell_it->width;
                    c_backup->width+=cell_it->width;
                }
                c_backup->x1=c_x1;
            }else{
                c_front=new cluster();
                c_front->y_pos=c_backup->y_pos;
                while(it!=cell_temp){
                    c_front->celllist.push_back(*it);
                    ++c_front->e;
                    c_front->q+=(*it)->x_pos-(*it)->width;
                    c_front->width+=(*it)->width;
                    ++it;
                }
                c_front->x1=c_backup->x1;
                for(auto cellit:c_front->celllist){
                    if(c_front->celllist.size()!=0)
                    cellit->ptr_cluster=c_front;
                    else{
                        cout<<"bug found in line 648"<<endl;
                    }
                }
                ++it;
                if(it!=c_backup->celllist.end()) {
                    int x1temp=(*it)->x_pos;
                    c_back=new cluster();
                    c_back->y_pos=c_backup->y_pos;
                    while (it!=c_backup->celllist.end()) {
                        c_back->celllist.push_back(*it);
                        ++c_back->e;
                        c_back->q+=(*it)->x_pos-(*it)->width;
                        c_back->width+=(*it)->width;
                        ++it;
                    }
                    c_back->x1=x1temp;
                    for(auto cellit:c_back->celllist){
                        if(c_back->celllist.size()!=0)
                        cellit->ptr_cluster=c_back;
                        else
                        cout<<"bug found in line 648"<<endl;
```

```
            }
            if(c_backup->prev!=nullptr) c_backup->prev->next=c_front; else cluster_v[cell_y]=c_front;
            if(c_backup->next!=nullptr) c_backup->next->prev=c_back;
            c_front->prev=c_backup->prev;
            c_back->next=c_backup->next;
            c_back->prev=c_front;
            c_front->next=c_back;
            delete c_backup;
          }
          else{
            if(c_backup->prev!=nullptr) c_backup->prev->next=c_front; else cluster_v[cell_y]=c_front;
            if(c_backup->next!=nullptr) c_backup->next->prev=c_front;
            c_front->prev=c_backup->prev;
            c_front->next=c_backup->next;
            delete c_backup;
          }
        }
        }else{
          delete c_backup;
          cluster* c_front=nullptr;
          cluster* c_back=nullptr;
          /****** if the target is white space, split the original cluster into two parts ****/
          auto it=cluster_new->celllist.begin();
         if(cell_temp==(--cluster_new->celllist.end())){
            cluster_new->width=0;
            cluster_new->q=0;
            cluster_new->e=0;
            for(auto c_it:cluster_new->celllist){
              ++cluster_new->e;
              cluster_new->q+=c_it->x_pos-c_it->width;
              cluster_new->width+=c_it->width;
            }
            if(cluster_new->prev!=nullptr) cluster_new->prev->next=cluster_new;
            else cluster_v[cell_y]=cluster_new;
            if(cluster_new->next!=nullptr) cluster_new->next->prev=cluster_new;

            for(auto c_it:cluster_new->celllist){
              c_it->ptr_cluster=cluster_new;
            }

          }else{
          /*** get c_front  ***/
            c_front=new cluster();
            c_front->y_pos=cluster_new->y_pos;
            while(it!=cell_temp){
              c_front->celllist.push_back(*it);
              ++c_front->e;
              c_front->q+=(*it)->x_pos-(*it)->width;
              c_front->width+=(*it)->width;
              ++it;
            }
            c_front->celllist.push_back(*it);
            ++c_front->e;
            c_front->q+=(*it)->x_pos-(*it)->width;
            c_front->width+=(*it)->width;
            ++it;
```

```
                c_front->x1=cluster_new->x1;
                for(auto cellit:c_front->celllist){
                      cellit->ptr_cluster=c_front;
                }
             /*** get c_back ***/
                int x1temp=(*it)->x_pos;
                c_back=new cluster();
                c_back->y_pos=cluster_new->y_pos;
                while (it!=cluster_new->celllist.end()) {
                   c_back->celllist.push_back(*it);
                   ++c_back->e;
                   c_back->q+=(*it)->x_pos-(*it)->width;
                   c_back->width+=(*it)->width;
                   ++it;
                }
                   c_back->x1=x1temp;
                   for(auto cellit:c_back->celllist){
                         cellit->ptr_cluster=c_back;
                   }
                   if(cluster_new->prev!=nullptr) cluster_new->prev->next=c_front; else
cluster_v[cell_y]=c_front;
                   if(cluster_new->next!=nullptr) cluster_new->next->prev=c_back;
                   c_front->prev=cluster_new->prev;
                   c_back->next=cluster_new->next;
                   c_back->prev=c_front;
                   c_front->next=c_back;
                   delete cluster_new;
             }
          }


          if(!t_split){
             /**** update the ptr_cluster in the cellist ***/
             for(auto it: target_new->celllist){
                if(target_new->celllist.size()!=0)
                   it->ptr_cluster=target_new;
                else
                   cout<<"bug found in line 695"<<endl;

             }
             /****** update target row cluster ****/
             if(t_head!=nullptr){
                if(t_head->prev!=nullptr) t_head->prev->next=target_new;
                else cluster_v[y]=target_new;
                if(t_tail->next!=nullptr) t_tail->next->prev=target_new;
             }else{
                if(target_new->prev!=nullptr) target_new->prev->next=target_new;
                else cluster_v[y]=target_new;
                if(target_new->next!=nullptr) target_new->next->prev=target_new;
             }

             /***** delete the original clusters *****/
             dtemp=t_head;
             while(dtemp!=t_tail){
                cluster* dnext=dtemp->next;
                delete dtemp;
```

```
            dtemp=dnext;
        }
        if(dtemp!=nullptr) delete dtemp;
    }else{  /***** for t_split  ****/
        delete t_backup;
        cluster* t_front=nullptr;
        cluster* t_back=nullptr;

        auto it=target_new->celllist.begin();

        if(target_temp==(--target_new->celllist.end())){
            target_new->width=target_new->width-t->width+c->width;

            if(target_new->prev!=nullptr) target_new->prev->next=target_new;
            else cluster_v[y]=target_new;
            if(target_new->next!=nullptr) target_new->next->prev=target_new;

            for(auto c_it:target_new->celllist){
                c_it->ptr_cluster=target_new;
            }

        }else{
            /*** get t_front  ***/
            t_front=new cluster();
            t_front->y_pos=target_new->y_pos;
            while(it!=target_temp){
                t_front->celllist.push_back(*it);
                ++t_front->e;
                t_front->q+=(*it)->x_pos-(*it)->width;
                t_front->width+=(*it)->width;
                ++it;
            }
            t_front->celllist.push_back(*it);
            ++t_front->e;
            t_front->q+=(*it)->x_pos-(*it)->width;
            t_front->width+=(*it)->width;
            ++it;
            t_front->x1=target_new->x1;
            for(auto cellit:t_front->celllist){
                cellit->ptr_cluster=t_front;
            }
            /*** get t_back ***/
            int x1temp=(*it)->x_pos;
            t_back=new cluster();
            t_back->y_pos=target_new->y_pos;
            while (it!=target_new->celllist.end()) {
                t_back->celllist.push_back(*it);
                ++t_back->e;
                t_back->q+=(*it)->x_pos-(*it)->width;
                t_back->width+=(*it)->width;
                ++it;
            }
            t_back->x1=x1temp;
            for(auto cellit:t_back->celllist){
                cellit->ptr_cluster=t_back;
            }
```

```
            if(target_new->prev!=nullptr) target_new->prev->next=t_front; else cluster_v[y]=t_front;
            if(target_new->next!=nullptr) target_new->next->prev=t_back;
            t_front->prev=target_new->prev;
            t_back->next=target_new->next;
            t_back->prev=t_front;
            t_front->next=t_back;
            delete target_new;
         }
      }



      // update row map
      target_row.clear();
      cluster* t_temp=cluster_v[y];
      while(t_temp!=nullptr){
         for (auto cell_ptr:t_temp->celllist){
            target_row[cell_ptr->x_pos]=cell_ptr;
         }
         t_temp=t_temp->next;
      }
      cell_row.clear();
      cluster* c_temp=cluster_v[cell_y];
      while(c_temp!=nullptr){
         for (auto cell_ptr:c_temp->celllist){
            cell_row[cell_ptr->x_pos]=cell_ptr;
         }
         c_temp=c_temp->next;
      }

   }else{  //the swap increases the hpwl, need to reverse
      //reverse cell x_pos, y_pos
      for(auto it_net:netlist){
         (it_net.second)->reverse();
      }
      //reverse cluster
      if(c==nullptr) cout<<"bug in line 765"<<endl;
      if(whitespace) {
         //restore the nullptr in c_backup
         *cell_temp=c;
      }
      else{
         //restore the ptr in c_backup
         *cell_temp=c;
         delete cluster_new;
      }
            //resotre the ptr in t_backup, then delete target_new
      if(!whitespace) { if(t==nullptr) cout<<"bug in line 787"<<endl; *target_temp=t; }
      delete target_new;
      // row map untouched, no need to reverse!!
            }
   if(legalswap)
   return new_hpwl-old_hpwl;
   else
      return 1;
}
```

```
   void addcluster(cluster* prevc, cluster* c){
      prevc->width+=c->width;
      prevc->e+=c->e;
      prevc->q+=c->q-prevc->e*c->width;
      prevc->x1=prevc->q/prevc->e;
      if(prevc->x1<0) prevc->x1=0;
      if(prevc->x1>x_size*sitewidth-prevc->width) prevc->x1=x_size*sitewidth-prevc->width;
   }

   bool collapse(list<cluster*>& clist, cluster* clu_p){
      clu_p->x1=clu_p->q/clu_p->e;
      if(clu_p->x1<0) clu_p->x1=0;
      else if(clu_p->x1>x_size-clu_p->width) clu_p->x1=x_size-clu_p->width;
      else{
         auto it=clist.end();
         --it;
         if(it==clist.begin()) return true;
         --it;
         if((*it)->fixed==1) return false;
         if((*it)->x1+(*it)->width>clu_p->x1){
            addcluster(*it,clu_p);
            clist.pop_back();
            return collapse(clist, *it);
         }

      }
      return true;
   }
};


#endif /* PLACEROW_H_ */
```

E. main.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#ifndef WIN32
# include <unistd.h>
#endif /* not WIN32 */
#include "defrReader.hpp"
#include "defiAlias.hpp"

/*****************************************************************************************
**********/
#include "class_define.h"
#include "get_target.h"
#include "result_func.h"
#include "placerow.h"

using namespace std;

auto hash_cell = new unordered_map<string, cell*>;
auto hash_gate = new unordered_map<string, double>;
auto hash_gate_height = new unordered_map<string, double>;
/*****************************************************************************************
*************/
char defaultName[64];
char defaultOut[64];

// Global variables
FILE* fout;
int userData;
int numObjs;
int isSumSet; // to keep track if within SUM
int isProp = 0; // for PROPERTYDEFINITIONS
int begOperand; // to keep track for constraint, to print - as the 1st char
//static double curVer = 0;
static int setSNetWireCbk = 0;



// TX_DIR:TRANSLATION ON

void myLogFunction(const char* errMsg){
   fprintf(fout, "ERROR: found error: %s\n", errMsg);
}

void myWarningLogFunction(const char* errMsg){
   fprintf(fout, "WARNING: found error: %s\n", errMsg);
}

void dataError() {
   //fprintf(fout, "ERROR: returned user data is not correct!\n");
   ;
}
```

```
void checkType(defrCallbackType_e c) {
   if (c >= 0 && c <= defrDesignEndCbkType) {
      // OK
   } else {
      fprintf(fout, "ERROR: callback type is out of bounds!\n");
   }
}

int done(defrCallbackType_e c, void* dummy, defiUserData ud) {
   checkType(c);
   if ((long)ud != userData) dataError();
   fprintf(fout, "END DESIGN\n");
   return 0;
}

int endfunc(defrCallbackType_e c, void* dummy, defiUserData ud) {
   checkType(c);
   if ((long)ud != userData) dataError();
   return 0;
}

char* orientStr(int orient) {
   switch (orient) {
      case 0: return ((char*)"N");
      case 1: return ((char*)"W");
      case 2: return ((char*)"S");
      case 3: return ((char*)"E");
      case 4: return ((char*)"FN");
      case 5: return ((char*)"FW");
      case 6: return ((char*)"FS");
      case 7: return ((char*)"FE");
   };
   return ((char*)"BOGUS");
}

int cs(defrCallbackType_e c, int num, defiUserData ud) {
   string name;
   checkType(c);
   if ((long)ud != userData) dataError();
   switch (c) {
      case defrComponentStartCbkType : name="COMPONENTS"; component_num = num; break;
      case defrNetStartCbkType : name="NETS"; net_num = num; break;
      case defrStartPinsCbkType : name="PINS"; pin_num = num; break;
      default : name="NO MATCH!"; return 1;
   }
   return 0;
}

int cls(defrCallbackType_e c, void* cl, defiUserData ud) {
   /**********addition because of net & component begin**********/
   defiNet* net1;
   defiComponent* co;
   int count = 0;
   /**********addition because of net & component end**********/
```

```
    defiSite* site; // Site and Canplace and CannotOccupy
    defiBox* box; // DieArea and
    defiPinCap* pc;
    defiPin* pin;
    int i, j;
    defiRow* row;
    defiTrack* track;
    defiGcellGrid* gcg;
    defiVia* via;
    defiRegion* re;
    defiGroup* group;
    defiScanchain* sc;
    defiIOTiming* iot;
    defiFPC* fpc;
    defiTimingDisable* td;
    defiPartition* part;
    defiPinProp* pprop;
    defiBlockage* block;
    defiSlot* slots;
    defiFill* fills;
    defiStyles* styles;
    int xl, yl, xh, yh;
    char *name, *a1, *b1;
    char **inst, **inPin, **outPin;
    int *bits;
    int size;
    int corner, typ;
    const char *itemT;
    char dir;
    defiPinAntennaModel* aModel;
    struct defiPoints points;
    checkType(c);
    if ((long)ud != userData) dataError();
    /****************************这里 case 选取了 DIEAERA/ ROW/ PIN 有关的, 又加上 NET/
COMPONENT************************/
    switch (c) {
       case defrNetCbkType:{
          net1 = (defiNet*)cl;
          string net_name = net1->defiNet::name();
          net* ptr_net = vec_net + vec_net_counter;
          new (ptr_net) net(net_name);

          for (i = 0; i < net1->defiNet::numConnections(); i++) {
             string cell_name = net1->defiNet::instance(i);
             cell* ptr_cell = (*hash_cell)[cell_name];
             if(ptr_cell == nullptr){
                cell_name = net1->defiNet::pin(i);
                ptr_cell = (*hash_cell)[cell_name];
             }
             (ptr_net->list_cell).push_back(ptr_cell);
             (ptr_cell->list_net).push_back(ptr_net);
          }
          ++vec_net_counter;
          break;
       }
```

```
        case defrComponentCbkType:{
            co = (defiComponent*)cl;
            string cell_name = co->defiComponent::id();
            string cell_type = co->defiComponent::name();
            int x_pos, y_pos;
            bool fixed, direction;
            bool is_pin = false;
            double gate_width_d = (*hash_gate)[cell_type];
            double gate_height_d = (*hash_gate_height)[cell_type];
            int gate_width = (gate_width_d + 0.99);
            int gate_height = (gate_height_d + 0.99);

            if (co->defiComponent::isFixed()){
                x_pos = co->defiComponent::placementX() / x_step;
                y_pos = co->defiComponent::placementY() / y_step;
                fixed = true;
                direction = true;
            }

            if (co->defiComponent::isPlaced()){
                x_pos = co->defiComponent::placementX() / x_step;
                y_pos = co->defiComponent::placementY() / y_step;
                fixed = false;
                direction = true;
            }

            new (vec_cell + vec_cell_counter) cell{cell_name, x_pos, y_pos, gate_width, gate_width_d,
gate_height, gate_height_d, fixed, false, is_pin};
            (*hash_cell)[cell_name] = (vec_cell + vec_cell_counter);
            subrow[y_pos][x_pos] = vec_cell + vec_cell_counter;
            ++vec_cell_counter;

            for(int i = 1 ;i < gate_height; ++i){
              new (vec_cell + vec_cell_counter) cell{"not used" + cell_name, x_pos, y_pos + i, gate_width,
gate_width_d, 1, 1, fixed, true, is_pin};
              subrow[y_pos+i][x_pos] = vec_cell + vec_cell_counter;
              ++vec_cell_counter;
            }

            break;
        }

        case defrDieAreaCbkType :{
            box = (defiBox*)cl;
            x_length = box->defiBox::xh() - box->defiBox::xl();
            y_length = box->defiBox::yh() - box->defiBox::yl();
            break;
        }

        case defrRowCbkType :{
            ++row_counter;
            row = (defiRow*)cl;
            if (row->defiRow::hasDo()) {
                column_num = row->defiRow::xNum();
                x_step = row->defiRow::xStep();
            }
```

```
            break;
        }

        case defrPinCbkType :{
            pin = (defiPin*)cl;

            string cell_name = pin->defiPin::pinName();
            string cell_type = "PIN";
            int x_pos, y_pos;
            bool fixed, direction;
            bool is_pin = true;
            char* ss = "INPUT";

            if (pin->defiPin::hasDirection()){
                if(!strcmp(pin->defiPin::direction(), ss))
                    direction = true;
                else
                    direction = false;
            }
            fixed = true;

            if (pin->defiPin::hasPlacement()) {
                if (pin->defiPin::isFixed()) {
                    x_pos = pin->defiPin::placementX() / x_step;
                    y_pos = pin->defiPin::placementY() / y_step;
                }
            }

            new (vec_cell + vec_cell_counter) cell{cell_name, x_pos, y_pos, 0, 0, 0, 0, fixed, true, is_pin};
            (*hash_cell)[cell_name] = (vec_cell + vec_cell_counter);
            //subrow[y_pos][x_pos] = vec_cell + vec_cell_counter;
            ++vec_cell_counter;

            break;
        }
    }
    return 0;
}

int units(defrCallbackType_e c, double d, defiUserData ud) {
    checkType(c);
    unit_distance_micro = d;
    if ((long)ud != userData) dataError();
    return 0;
}
/*****************************************************************************************
******************/

int main(int argc, char** argv) {
    int num = 1734;
    char* inFile[6];
    char* outFile;
    FILE* f;
    int res;
    int noCalls = 0;
    int retStr = 0;
```

```
int numInFile = 0;
int fileCt = 0;

strcpy(defaultName, "def.in");
strcpy(defaultOut, "list");
inFile[0] = defaultName;
outFile = defaultOut;
fout = stdout;
userData = 0x01020304;

argc--;
argv++;
while (argc--) {
   if (strcmp(*argv, "-d") == 0) {
      argv++;
      argc--;
      sscanf(*argv, "%d", &num);
      defiSetDebug(num, 1);
   } else if (strcmp(*argv, "-nc") == 0) {
      noCalls = 1;
   } else if (strcmp(*argv, "-o") == 0) {
      argv++;
      argc--;
      outFile = *argv;
      if ((fout = fopen(outFile, "w")) == 0) {
         fprintf(stderr, "ERROR: could not open output file\n");
         return 2;
      }
   } else if (strcmp(*argv, "-verStr") == 0) {
      /* New to set the version callback routine to return a string */
      /* instead of double. */
      retStr = 1;
   } else if (argv[0][0] != '-') {
      if (numInFile >= 6) {
         fprintf(stderr, "ERROR: too many input files, max = 6.\n");
         return 2;
      }
      inFile[numInFile++] = *argv;
   } else if (strcmp(*argv, "-h") == 0) {
      fprintf(stderr, "Usage: defrw [<defFilename>] [-o <outputFilename>]\n");
      return 2;
   } else if (strcmp(*argv, "-setSNetWireCbk") == 0) {
      setSNetWireCbk = 1;
   } else {
      fprintf(stderr, "ERROR: Illegal command line option: '%s'\n", *argv);
      return 2;
   }

   argv++;
}

if (noCalls == 0) {
   /***********************************************************/
   defrSetUnitsCbk(units);
   defrSetRowCbk((defrRowCbkFnType)cls);
   defrSetDieAreaCbk((defrBoxCbkFnType)cls);
```

```
    defrSetComponentStartCbk(cs);
    defrSetStartPinsCbk(cs);
    defrSetNetStartCbk(cs);
    /***********************************************************/
  }

  defrInit();

  for (fileCt = 0; fileCt < numInFile; fileCt++) {
    defrReset();
    if ((f = fopen(inFile[fileCt],"r")) == 0) {
      fprintf(stderr,"Couldn't open input file '%s'\n", inFile[fileCt]);
      return(2);
    }
    // Set case sensitive to 0 to start with, in History & PropertyDefinition
    // reset it to 1.
    res = defrRead(f, inFile[fileCt], (void*)userData, 1);

    if (res)
      fprintf(stderr, "Reader returns bad status.\n", inFile[fileCt]);

    (void)defrPrintUnusedCallbacks(fout);
    (void)defrReleaseNResetMemory();
  }

  row_num = row_counter;
  x_step = (double)x_length / column_num;
  y_step = (double)y_length / row_num;

  vec_cell = (cell*) operator new(sizeof(cell) * (component_num + pin_num + 10000));
  vec_net = (net*) operator new(sizeof(net) * net_num);
  subrow.resize(row_num + 1);                  //+1 because of pin

  /*********************************read gate_size*************************************/
  ifstream iifile;
  string strtmp;
  iifile.open("gate_size_vga");
  while(getline(iifile, strtmp, '\n')){
    string gate_name = strtmp;
    getline(iifile, strtmp, '\n');
    double gate_size = stod(strtmp);
    getline(iifile, strtmp, '\n');
    double gate_height = stod(strtmp);
    (*hash_gate)[gate_name] = gate_size * unit_distance_micro / x_step;
    (*hash_gate_height)[gate_name] = gate_height * unit_distance_micro / y_step;
  }

  /*********************************read gate_size*************************************/
  fprintf(stderr,"******x_step: %d****y_step: %d*********************************\n", x_step,
y_step);

  fprintf(stderr,"row_num: %d  pin_num: %d  component_num: %d  net_num: %d  column_num: %d\n",
row_num, pin_num, component_num, net_num, column_num);
```

```
/*****************************************************************************************
****/
   if (noCalls == 0) {
      defrSetComponentCbk((defrComponentCbkFnType)cls);
      defrSetNetCbk((defrNetCbkFnType)cls);
      defrSetPinCbk((defrPinCbkFnType)cls);
   }

   defrInit();

   for (fileCt = 0; fileCt < numInFile; fileCt++) {
      defrReset();
      if ((f = fopen(inFile[fileCt],"r")) == 0) {
         fprintf(stderr,"Couldn't open input file '%s'\n", inFile[fileCt]);
         return(2);
      }
      // Set case sensitive to 0 to start with, in History & PropertyDefinition
      // reset it to 1.
      res = defrRead(f, inFile[fileCt], (void*)userData, 1);

      if (res)
         fprintf(stderr, "Reader returns bad status.\n", inFile[fileCt]);

      (void)defrPrintUnusedCallbacks(fout);
      (void)defrReleaseNResetMemory();
   }


   delete hash_cell;
   delete hash_gate;
   delete hash_gate_height;
   /**********************************just for test
begin*****************************************/
   cout<<endl<<endl;
   cout.precision(12);
   bool legal_before = check_legal();
   double hpwl_before = total_hpwl();
   time_t t=time(0);
   struct tm* now=localtime(&t);
   int old_hour = now->tm_hour;
   int old_min = now->tm_min;
   int old_sec = now->tm_sec;
   cout<<"start time: "<<old_hour<<":"<<old_min<<":"<<old_sec<<endl;
   cout<<"before:\t 1 is legal, 0 is illegal: "<<legal_before<<endl;
   cout<<"before:\t total hpwl: "<<hpwl_before<<endl;

   ffout.open("for_para");
   int inc_count = 0;
   int dec_count = 0;
   auto placer=row_placer(subrow,column_num,row_num-1,x_step);////////////////////////////////////

   int need_swap=0;
   for(int xxx = 0; xxx < 3; ++xxx){//////////////////////////////////////////////////////
           thresh = 20;
                   for(int i = 0; i < vec_cell_counter; ++i){
                           if(!(vec_cell + i)->fixed){
```

```
                                auto vec_target = get_target(vec_cell + i);
                                if(vec_target.size()!=0){
                                        if(vec_target[0].y!=(vec_cell+i)->y_pos){
                                                int result=placer.swap(vec_target[0].x,
vec_target[0].y, vec_cell+i);
                                        }
                                        else{
                                                placer.same_row_swap(vec_target[0].x,
vec_target[0].y, vec_cell+i);
                                        }
                                        ++need_swap;
                                }
                        }
                }
                cout<<"**************************************"<<endl;
                cout<<"big loop #"<<xxx+1<<"  after strategy 1: "<<endl;
                display_time();
                cout<<"hpwl: "<<total_hpwl()<<endl;
                cout<<"**************************************"<<endl;

                thresh = 200;
                for(int xxxx = 0; xxxx < 1; ++xxxx){//////////////////////////////////////
                        for(int i = 0; i < vec_cell_counter; ++i){
                                if(!(vec_cell + i)->fixed){
                                        auto vec_target = get_target_s3(vec_cell + i);
                                        if(vec_target.size()!=0){
                                                if(vec_target[0].y!=(vec_cell+i)-
>y_pos){
                                                        int
result=placer.swap(vec_target[0].x, vec_target[0].y, vec_cell+i);
                                                }
                                                else{

        placer.same_row_swap(vec_target[0].x, vec_target[0].y, vec_cell+i);
                                                }
                                                ++need_swap;
                                        }
                                }
                        }
                }

                cout<<"**************************************"<<endl;
                cout<<"big loop #"<<xxx+1<<"  after strategy 2: "<<endl;
                display_time();
                cout<<"hpwl: "<<total_hpwl()<<endl;
                cout<<"**************************************"<<endl<<endl;
   }

                thresh = 0;
                for(int i = 0; i < vec_cell_counter; ++i){
                        if(!(vec_cell + i)->fixed){
                                auto vec_target = get_target(vec_cell + i);
                                if(vec_target.size()!=0){
                                        if(vec_target[0].y!=(vec_cell+i)->y_pos){
                                                int
result=placer.swap(vec_target[0].x, vec_target[0].y, vec_cell+i);
```

```
                                                        }
                                                        else{

            placer.same_row_swap(vec_target[0].x, vec_target[0].y, vec_cell+i);
                                                        }
                                                        ++need_swap;
                                                }
                                        }
                                }
    cout<<"************************************"<<endl;
    cout<<"final:"<<endl<<endl;
            cout<<"whitespaceswap count "<<whitespaceswap<<endl;
    cout<<"decrease count "<<placer.decrease_count<<endl;
    cout<<"increase count "<<placer.increase_count<<endl<<endl;

    time_t tt=time(0);
    now=localtime(&tt);
    cout<<"start time: "<<old_hour<<":"<<old_min<<":"<<old_sec<<endl;
    cout<<"end   time: "<<now->tm_hour<<":"<<now->tm_min<<":"<<now->tm_sec<<endl<<endl;


    cout<<"before:\t 1 is legal, 0 is illegal: "<<legal_before<<endl;
    cout<<"before:\t total hpwl: "<<hpwl_before<<endl;

    cout<<"after:\t 1 is legal, 0 is illegal: "<<check_legal()<<endl;
    cout<<"after:\t total hpwl: "<<total_hpwl()<<endl;
    cout<<"************************************"<<endl;

    fclose(fout);
    return res;
}
```