CS6135 HW4 Global Placement Report

109062509 熊磊

I. The wirelength and the runtime of each testcase.

	IBM01	IBM02	IBM05	IBM07	IBM08	IBM09
Global HPWL	418793367	861014178	10728134	3395395426	3750901269	4007791442
Legal HPWL	479407458	974822225	10043671	3560161822	3904811905	4184176898
Detail HPWL	292446882	624234343	9962963	2173906812	2309337900	2469194779
Global Run Time (s)	24	38	0	90	104	105
Total Run Time (s)	**30**	**49**	**4**	**125**	**145**	**145**

II. The details of the algorithm

Algorithm 1: Global Placement PseudoCode

Result: A Legalizable Global Placement
Objective: Minimize $\sum_{n \in Nets} LSEWL_{n,\eta} + \lambda \times \sum_{b \in Bins} (D_b(x, y) - T_b)^2$

Parse Input Data;
if *have stored seed* **then**
  restore the seed.
else
  search the seed having minimal HPWL.
end
Do Initial Placement; ▷ *netFirst*: modules connected in the same net would be placed in closely
for *round* ***in*** *1 ... 4* **do**
  compute *LSE wirelength* and *Bin Density* ;
    LSE = $\eta \sum_{n \in N} (\ln(\sum_{v_k \in n} e^{x_k/\eta}) + \ln(\sum_{v_k \in n} e^{-x_k/\eta}) + \ln(\sum_{v_k \in n} e^{y_k/\eta}) + \ln(\sum_{v_k \in n} e^{-y_k/\eta}))$
    Bin Density = $\sum_{b \in B} \sum_{v \in V} c_v \cdot \theta_{x_{b,v}} \cdot \theta_{y_{b,v}}$ ▷ Bell Shape Smoothing

  if *round == 1* **then**
    Focus on minimizing HPWL, set $\lambda = 0$; ▷ First Round
    Compute the gradient of LSE and update the placement for *NumIteration* times;
  else
    Compute the gradient of (LSE+BinDensity) and update the placement for *NumIteration* times;
  end
  Check if any module was out of bounds and adjust them.
  Save the current placement.
  $\lambda = \lambda + 1000$;
end
Do Legalization;
Do Detail Placement;
Output Result;

III. What tricks did you do to speed up your program or to enhance your solution quality?

1) Better Initialization by *netFirst* placing. It is to place the modules connected to the common net closely, and, at the same time, maintain the bin density by specifying a upper bound. I saved the seed of randomization to reproduce the result.
2) Parameter such as *λ, η*, *NumIteration*, and *StepSizeBound* are specified based on repeated tests.
3) Most of the implementation ideas come from the slides, and "*Challenges and Solutions in Modern Circuit Placement*" by Y.-W Chang.

IV. Please compare your results with the previous top 5 students' results and show your advantage either in runtime or in solution quality. Are your results better than them?

	Wirelength			Total Runtime(s)		
Ranks	ibm01	ibm05	ibm09	ibm01	ibm05	ibm09
1	86731510	11640852	537572709	180	380	894
2	76524759	10287864	720387019	505	481	1176
3	212729459	9962963	1640571405	1	4	11
4	123499363	12490103	1558917733	59	143	322
5	315825188	28470167	2644255425	8	40	56
My	292446882	9962963	2469194779	30	4	145

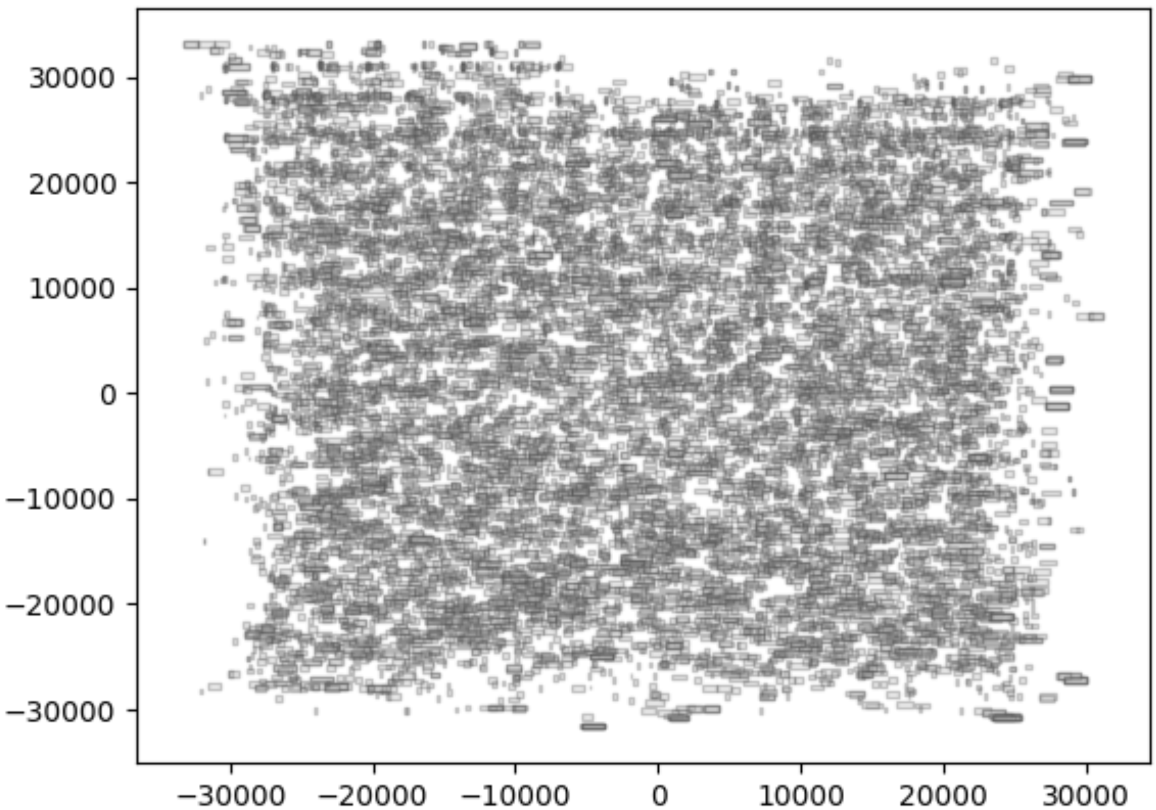
Some of my wirelength is better than, or as good as, top-5 from previous classmates. However, most results are far from the best. I think it is because that, bell-shape smoothing makes the objective function does not good enough to find global or even the local minimal.

V. Final Thoughts

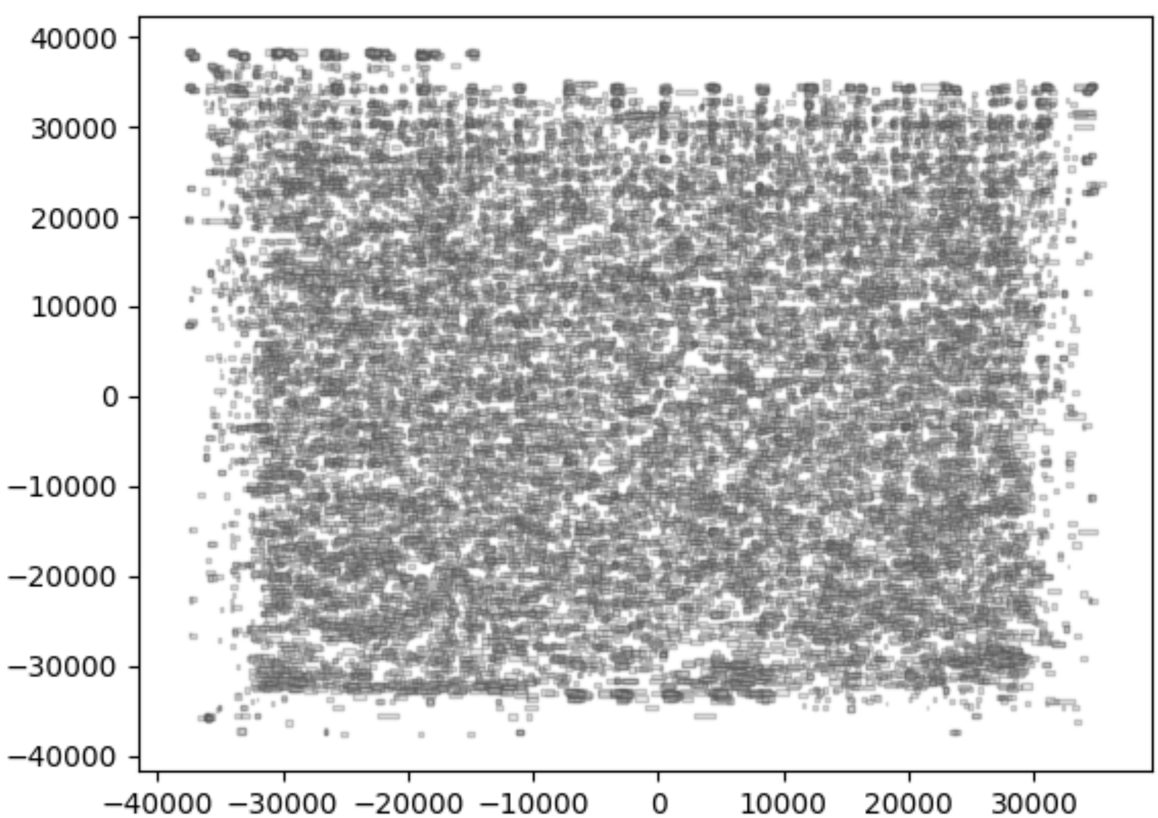
There is much less work have to be done in this homework, and the idea is simple, which is try to minimize Wirelength and avoid heavy overlapping. But, it is hard to get high quality result. Some time, the results I get are even worse than random placement, which really disappoints me. I would try to improve the performance in the future if I have time.

VI. Visualization of the result of Global Placement.

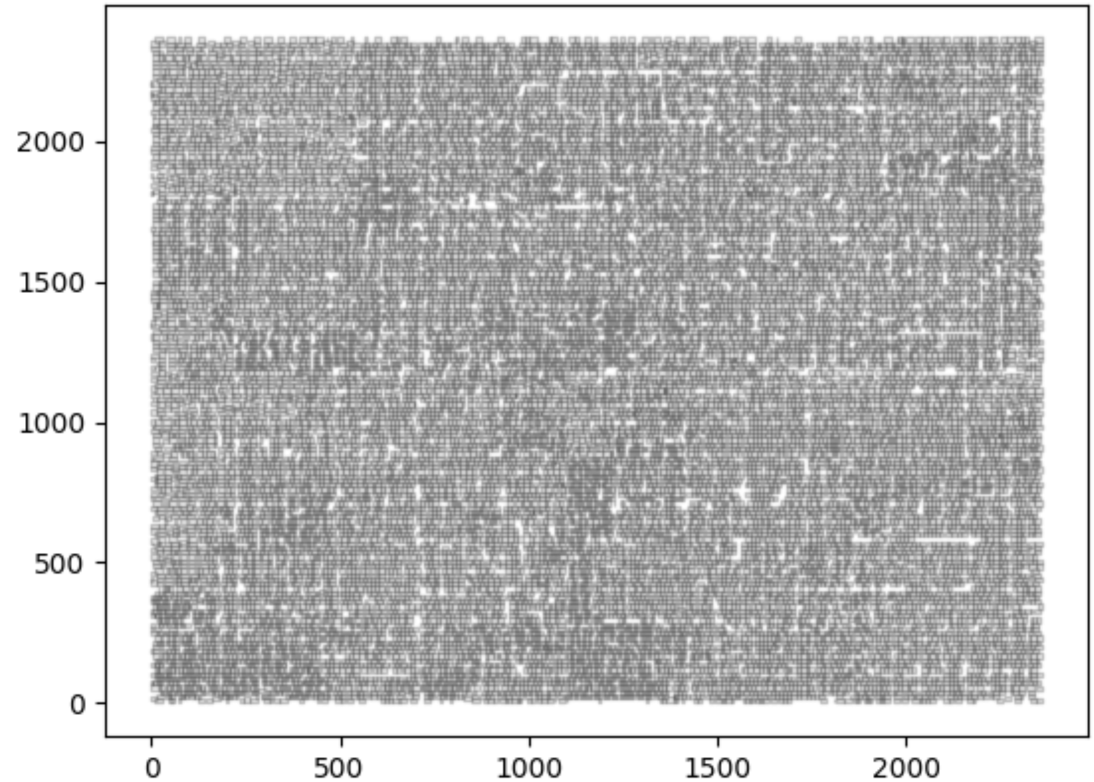
During the homework, I'm really curious about how is the global placement was been optimized, hence I visualize the results of global placement. Since I let *StepSizeBound = bounderyWidth / 20*, one can clearly see that there is a clustering effect for nearly modules.

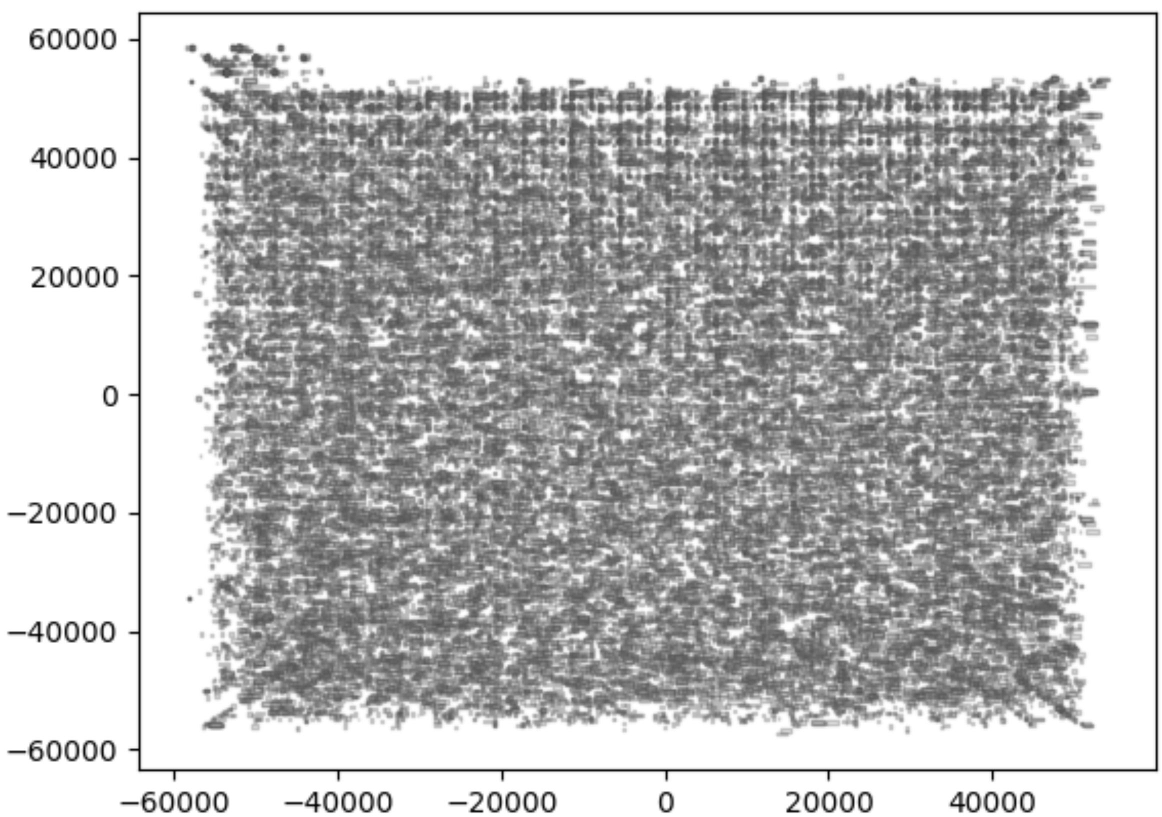

30000
20000
10000
0
−10000
−20000
−30000
−30000 −20000 −10000 0 10000 20000 30000

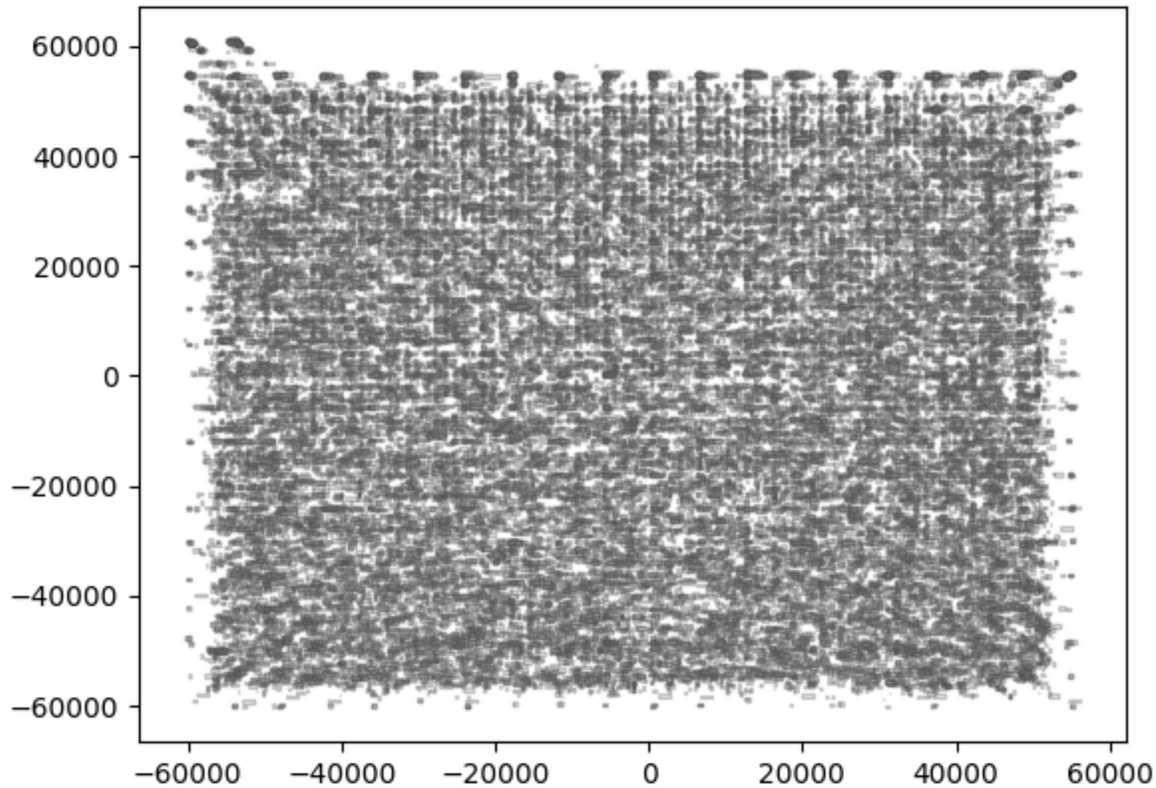
▲ ibm01


40000
30000
20000
10000
0
−10000
−20000
−30000
−40000
−40000 −30000 −20000 −10000 0 10000 20000 30000

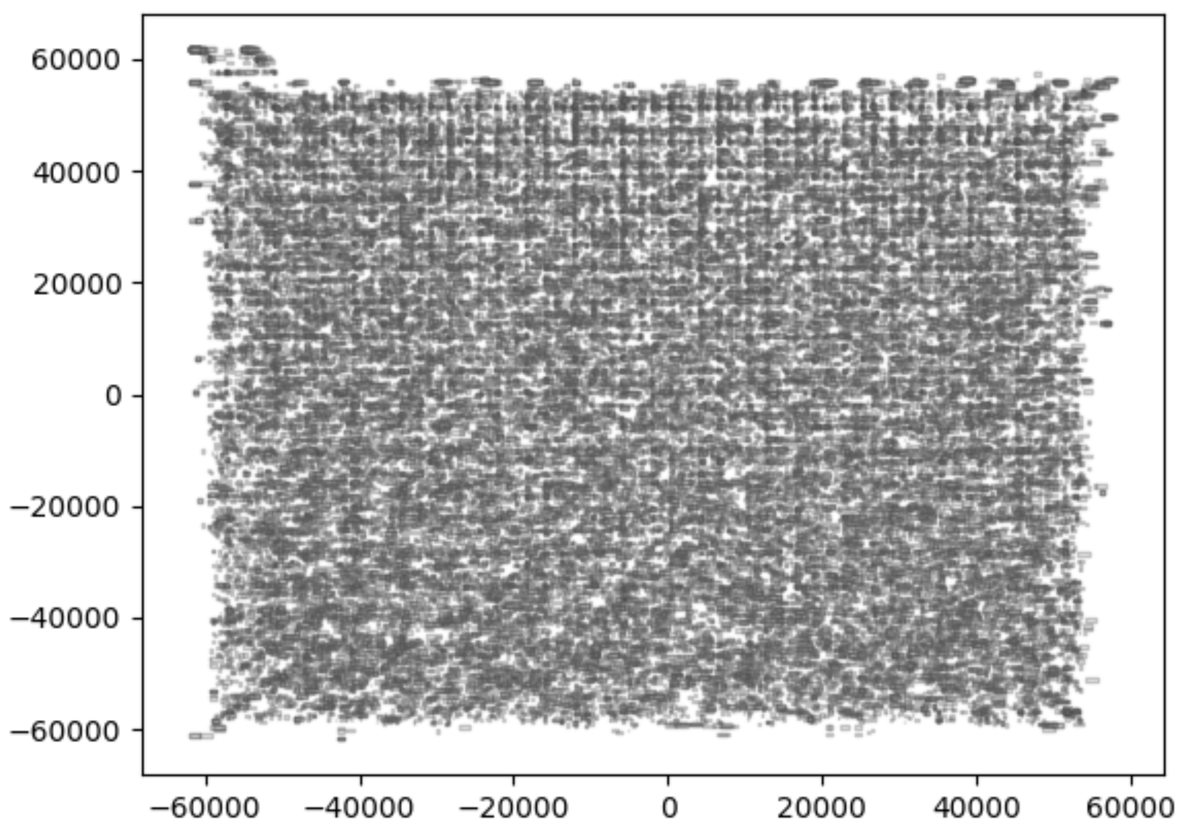
▲ ibm02


2000
1500
1000
500
0
0
500
1000
1500
2000
▲ ibm05


60000
40000
20000
0
−20000
−40000
−60000
−60000
−40000
−20000
0
20000
40000
▲ ibm07


60000
40000
20000
0
−20000
−40000
−60000
−60000
−40000
−20000
0
20000
40000
60000

▲ ibm08


60000
40000
20000
0
−20000
−40000
−60000
−60000
−40000
−20000
0
20000
40000
60000

▲ ibm09