ENEE640 Exam 2 Report

1. Abstract

This project aims to design the global placer that can place sets of blocks in the way to optimize the overlapping area and wire-length of placement. The global placers are based on the simulated annealing optimization algorithm, and three benchmarks have been used to evaluate the performance of global placer.

2. Problem Formulation

The objective of the global placer is to generate the optimal positions for all blocks so as to minimize both the total wire-length and overlapping area, along with the constraints that the blocks can only be placed within the rectangular region enclosed by fixed terminals.

.The optimization problem is formulated as the following:

Minimize { alpha * Overlapping_Area + (1 - alpha) * Wirelength }

s.t. (i) for benchmark requiring Overlapping_Area less than 5 %, the Overlapping_Area cannot exceed 5 %; for benchmark requiring Overlapping_Area less than 25 %, the Overlapping_Area cannot exceed 25 %;

(ii) the blocks can only be placed within the rectangular region enclosed by fixed terminals, so the constraints for the center coordinate of all blocks can be represented as:

$$\max_{1\le i\le n}(x_{b,i} + \frac{W_i}{2}) \le \max_{\forall x_p}(x_p)$$

$$\min_{1\le i\le n}(x_{b,i} - \frac{W_i}{2}) \ge \min_{\forall x_p}(x_p)$$

$$\max_{1\le i\le n}(y_{b,i} + \frac{h_i}{2}) \le \max_{\forall y_p}(y_p)$$

$$\min_{1\le i\le n}(y_{b,i} - \frac{h_i}{2}) \ge \min_{\forall y_p}(y_p)$$

alpha corresponds to the coefficient set to cost function, ranging from 0 to 1 inclusively. Overlapping_Area represents the total overlapping area between blocks. Wirelength is defined as the sum of half parameter (width plus height) of all netlists.

3. Data Structure

The information of block or terminal is stored in the self-defined data structure called **net.** Note here that the boolean flag *is_block* is used to distinguish whether the net is block or terminal.

```
1
2  public class net {
3      public double width;  //the width of the block/terminal
4      public double height;  //the height of the block/terminal
5      public String id;    // the name of the block/terminal
6      public double x;     // the x coordinate of block/terminal
7      public double y;     // the y coordinate of block/terminal
8      public double area;  // the area of block
9      public boolean is_block;     // true means it's block, otherwise it's terminal
10     public net(String id,double x,double y,boolean is_block) {
11         this.id = id;
12         this.is_block = is_block;
13         if(is_block) {  //means it is block
14             this.width = x;
15             this.height = y;
16             this.area = x * y;
17             x = 0;
18             y = 0;
19         }
20         else {   //it is terminal
21             this.x = x;
22             this.y = y;
23             this.width = 0;
24             this.height = 0;
25             this.area = 0;
26         }
27     }
28 }
```

Figure 1 The Screenshot of Data Structure net

Then nets are stored in the hashmap <id, net> called **res** where id represents the id of net and net corresponds the data structure constructed above. Note here the net is pass-by-reference, so the change of value of net will be reflected in its all related data structures.

4. Simulated Annealing

Because global placement is an NP-hard problem, the simulated-annealing algorithm is used to approximate the global optimum of a given function with a reasonable time. The pseudocode of my simulated annealing algorithm is shown as follow:

Algorithm: Simulated-annealing based global placement

Input: map<id, net> res

Output: map<id, net> res;

```
temp ← init-temp;
plan ← Rand-Generation (res);
    for k = 0  through kmax
        Cur_cost ← getCost(plan)
        New_plan ← Generate a new random position for a picked block
        New_cost ← getCost(new_plan)
        Cost_Diff ← exp(-( New_cost - Cur_cost) / temp);
        if  New_cost - Cur_cost < 0  do
            plan ← New_plan;
            Best_plan ← New_plan;
        else if Random(0,1) < Cost_Diff do
            plan ← New_plan;
        endif
        temp ← temp * cooling_rate;
    endfor
```

4.1 Rand-Generation

Random positions of the blocks are generated randomly within the fixed terminal's boundary. The algorithm is shown below:

```
for net n in res
    if n.is_block is true
        x ← random(x_min, x_max);
        y ← random(y_min, y_max);
        while x + n.width/2 > =  x_max or x - n.width/2 <=  x_min
            x ← random(x_min, x_max);
        endwhile
        while y + n.height/2 > =  y_max or y - n.height/2 <=  y_min
            y ← random(y_min, y_max);
        endwhile
    endif
endfor
```

4.2 getCost Function

The getCost function is as follows:

alpha * Overlapping_Area + (1 - alpha) * Wirelength

According to the experiments with reasonable running time, alpha ranging from 0.3 to 0.45 generates the good solution for a requirement of overlapping area is less than 25 %, and alpha ranging from 0.75 to 0.85 generates the good solution for a requirement of overlapping area is less than 5 %.

4.3 Temperature and Termination Condition

The initial temperature is set to 100 in my program, and cooling rate is set to pow(0.99,k) in which k represents the number of iterations. The program is terminated when k exceeds the k_max, where k_max is set to 450000 in general.

5. Experiments and Results

The global placement is implemented in Java, and three benchmarks are used to evaluate the performance of the global placer. The simulation is able to run under the JDK 1.8, and the results are shown in Table 1. The plots of three global placement solutions are illustrated in Figure 2.

	$OL_r \leq 5\,\%$	$OL_r \leq 25\,\%$
Benchmark 1	1.093977E+07	7.158525E+06
Benchmark 2	3.377767E+07	1.813226E+07
Benchmark 3	4.108584E+07	3.057802E+07
Average	28601093	18622935

The plots of benchmarks are shown as below:

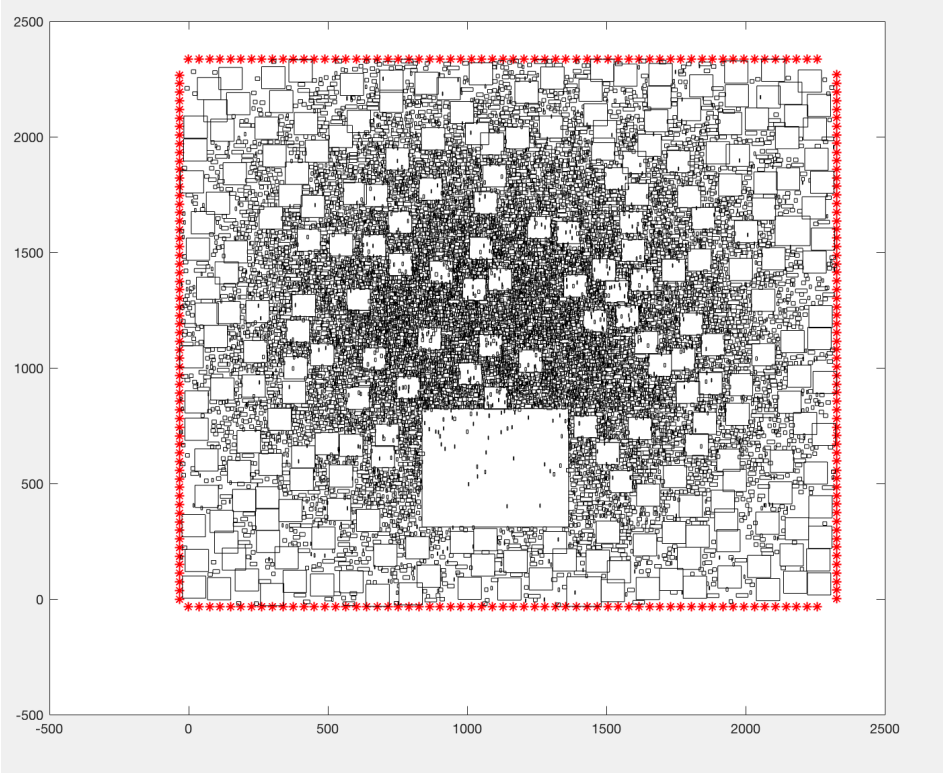

Figure 2 The Plot of Benchmark 1 with $OL_r \leq 5\,\%$

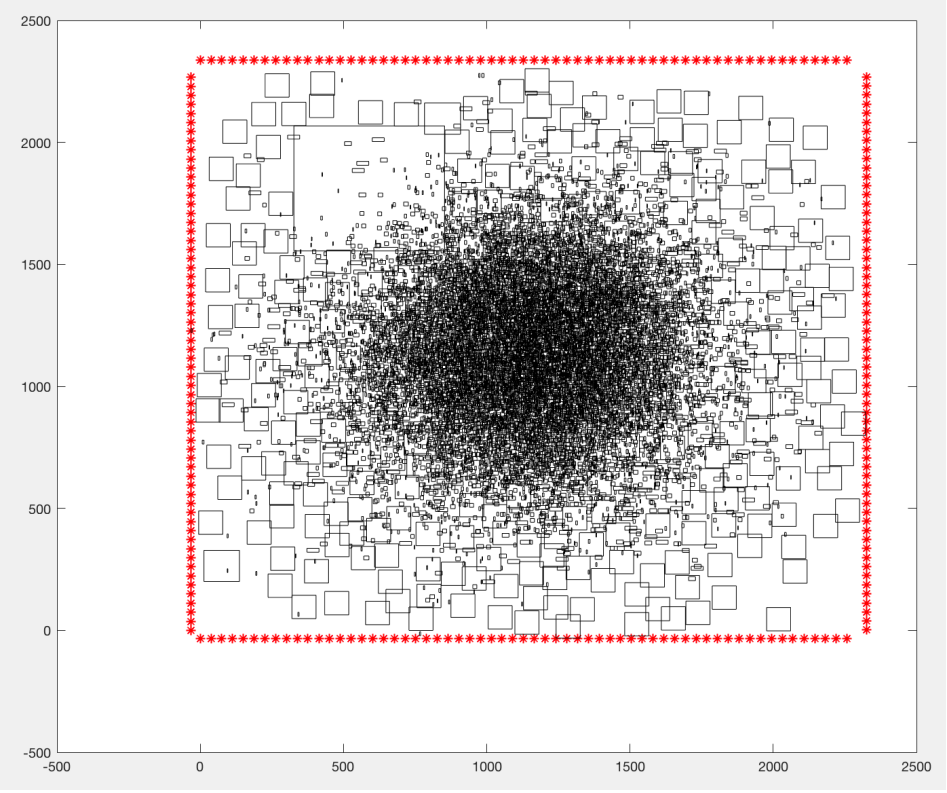

Figure 3 The Plot of Benchmark 1 with $OL_r \leq 25\,\%$

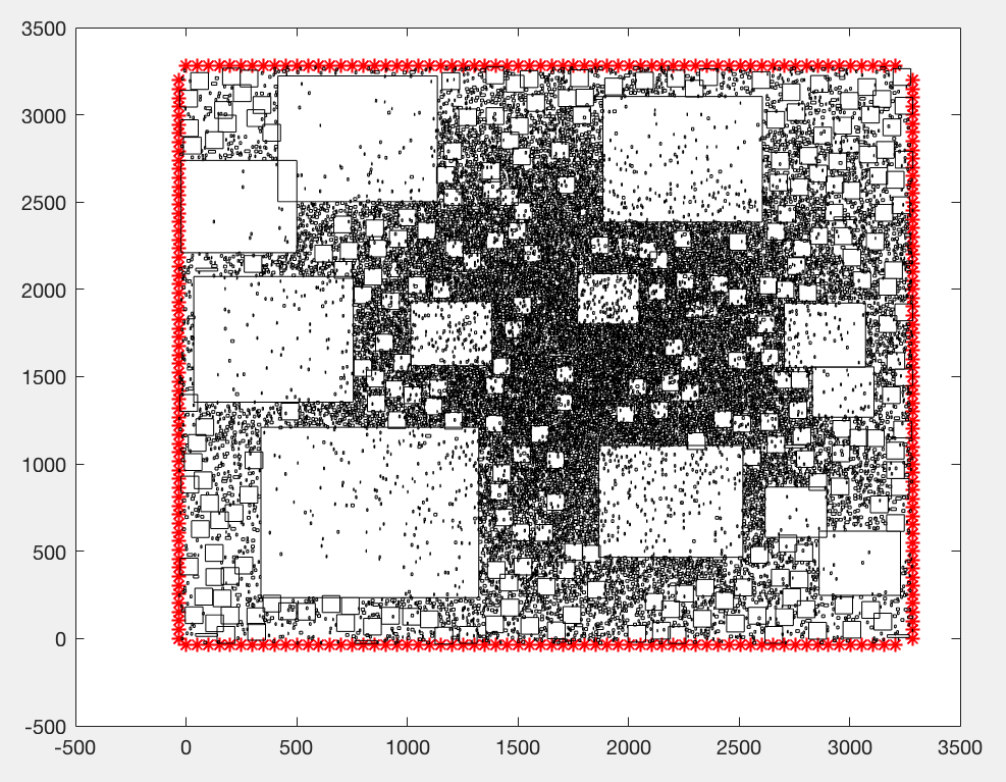

Figure 4 The Plot of Benchmark 2 with $OL_r \leq 5\,\%$

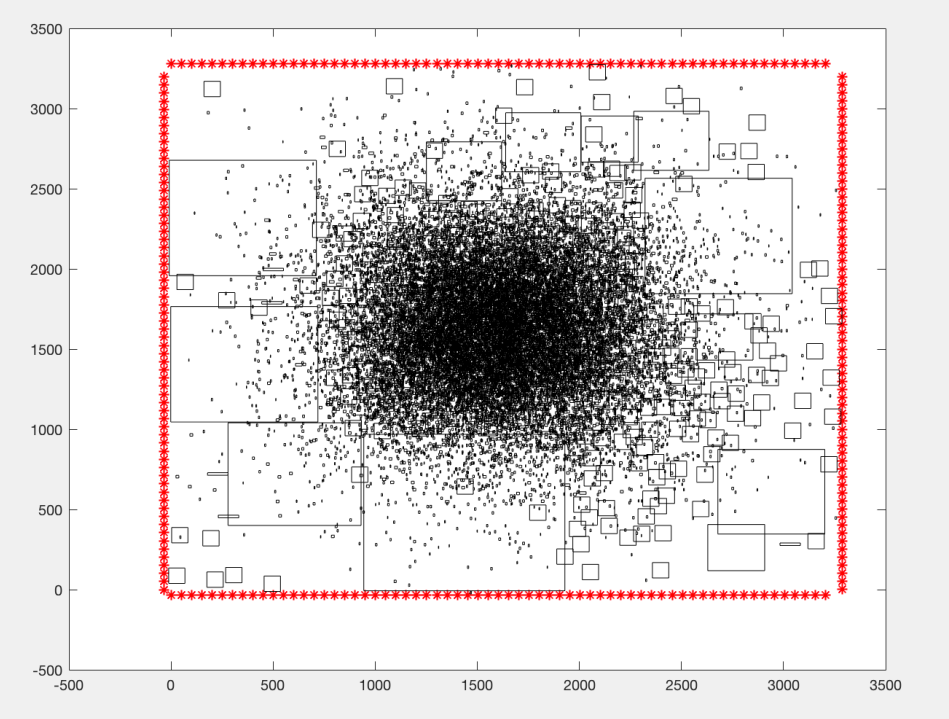

Figure 5 The Plot of Benchmark 2 with $OL_r \leq 25\,\%$

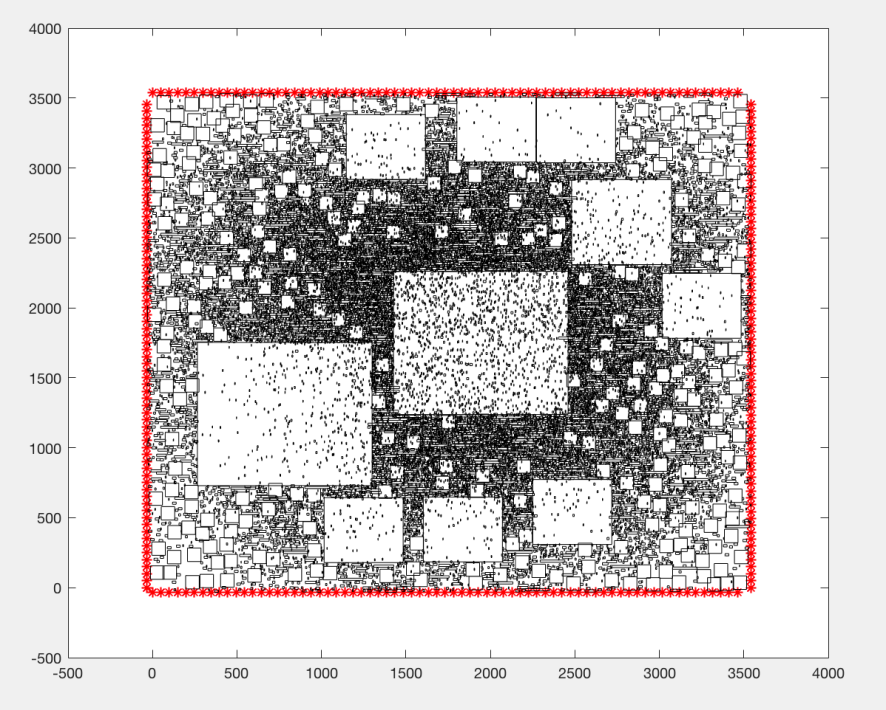

Figure 6 The Plot of Benchmark 3 with $OL_r \leq 5\,\%$

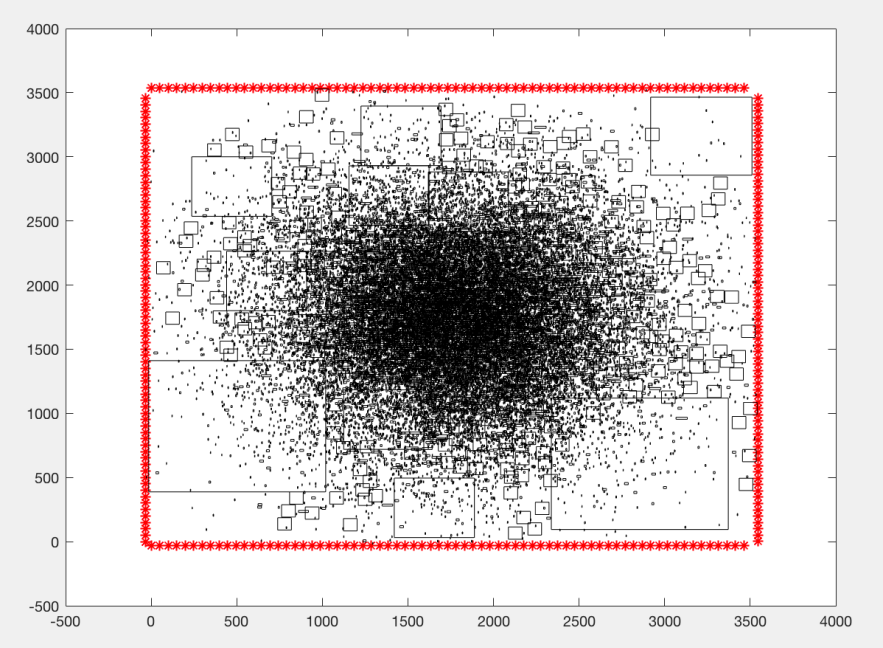

Figure 7 The Plot of Benchmark 3 with $OL_r \leq 25\,\%$

6. Discussion and Future Improvement

In general, a longer running time for simulated annealing algorithm generates a better solution. To run the simulation with reasonable running time and better result, the setting of parameters is essential to the project. The discussion of changing parameters is shown as below:

(1) initial temperature: when the initial temperature is high, the simulated annealing algorithm starts with the higher probability of accepting bad solutions. Based on my experiment the higher temperature makes the running time longer but not always generates the better solution. When temp = 100 it generates a reasonable solution.

(2) the number of iterations: the simulated annealing algorithm will run longer time and generate the better solution in general. However, it violates the property that running simulation within the reasonable time. To guarantee the running time within two hours, the number of iterations is set to 450000

(3) cooling rate: it affects acceptation rate of bad solution, because temp decreases by multiplying cooling rate (< 1) in each iteration. In order to maintain a slowly decreasing rate, I set up the cooling rate as Pow(0.99,k) where k represents the number of the current iteration

(4) alpha for cost function: For the benchmark requiring less than 25 % area, the alpha is set to the range between 0.3 to 0.4 for three benchmarks; For the benchmark requiring less than 5 % area, the alpha is set to the range between 0.75 to 0.85 for three benchmarks. The reason to do this way is that the requirement of less than 25 % overlapping area needs more time on obtaining optimal wirelength, while the requirement of less than 5 % area needs more time on obtaining optimal overlapping area.

Future improvements can be done in following aspects: (1) Adjust better parameters for current simulated annealing algorithm. Due to the time limit, I just can narrow the parameters down into certain range and try if it is optimal. (2) Since my program is written in Java, it is slower in running time compared to C/C++. So the improvements can be done in translating Java into C/C++. (3) Combining partition with simulated annealing. The partition can generate multiple random positions for multiple picked neighbors increasing the efficiency of the program.