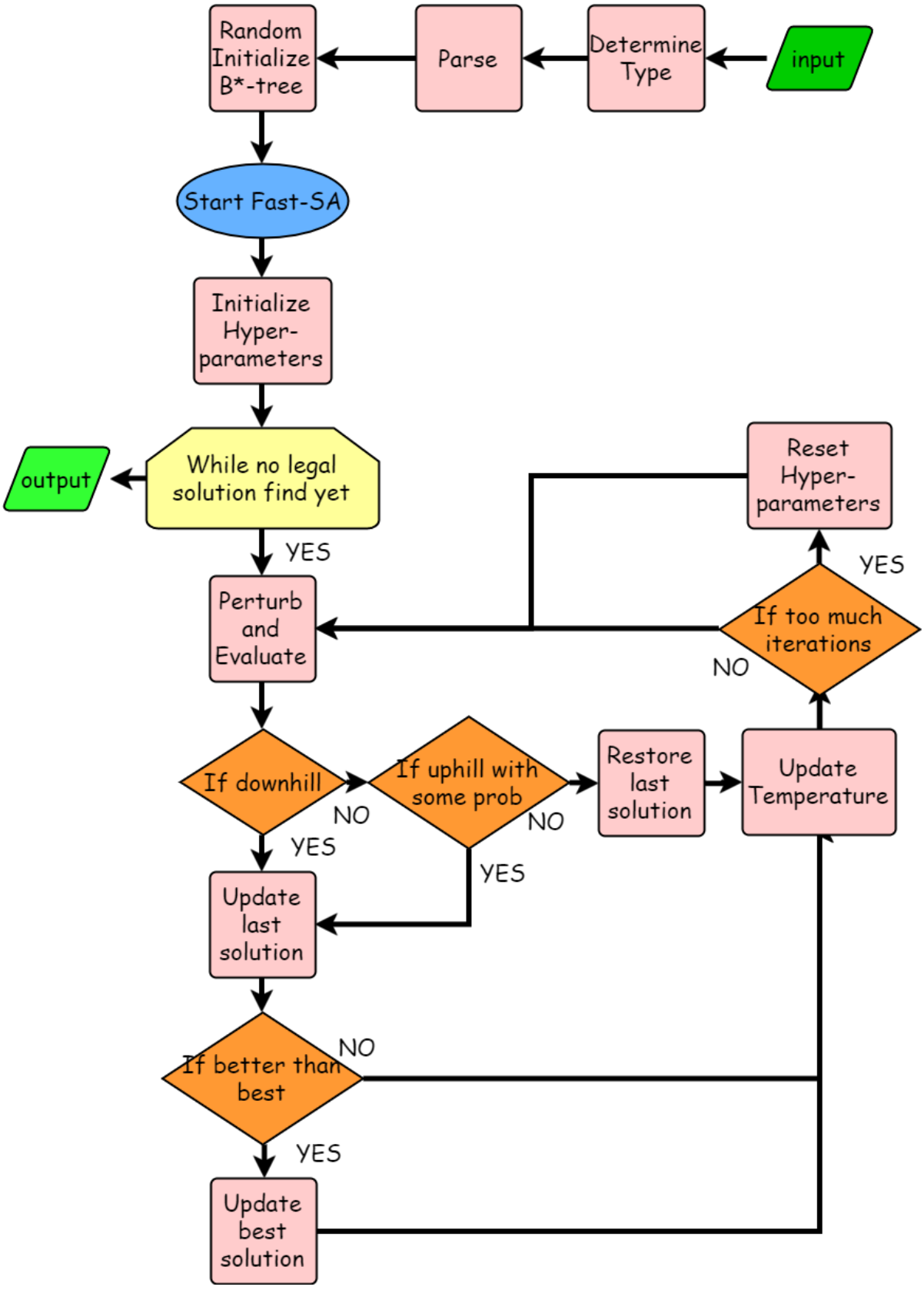
Physical Design for Nanometers ICs, PA 2

B03901056, 電機三, 孫凡耕

(1) Algorithm flow:



(2) Detailed description:

Determine Type→

Since the quantity of the blocks is pretty small and the size of the outline is not big. My program will dynamically choose the data types to store the data. For data storing IDs, the type will be either unsigned char or unsigned short. For data storing LENs (x, y, w, h), the type will be either unsigned short or unsigned int.

Initialize Hyper-parameters→

P = 0.9, alpha_base = 0.5, k = max(2, num_blocks/11), perturb per iter = 2*num_blocks+20

c = max(100-num_blocks, 10).

First perturb num_blocks times to obtain the average hpwl, area, aspect ratio and true cost. Here the true cost represents the cost for the output.

The cost function I used is the same as the Fast-SA which every term is normalized according to the value computed above.

Temperature Update →

According to the Fast-SA:

$$T_n = \begin{cases} \frac{\Delta_{avg}}{\ln P} & n = 1 \\ \frac{T_1\langle\Delta_{cost}\rangle}{nc} & 2 \le n \le k \\ \frac{T_1\langle\Delta_{cost}\rangle}{n} & n > k. \end{cases}$$

(3) Data Structure:

FLOOR_PLAN class stores all information about the blocks and nets along with a TREE class representing the current relative position of each block.

BLOCK struct stores the height, width, x-position, y-position, also with a bool to indicate this block has been rotated or not. Both blocks and terminals in input.block file are all stored in the form of BLOCK struct. The difference is that the height and width of terminals are always zero.

NET struct stores which BLOCKS (including blocks and terminals) associated with this net.

TREE class stores the b*-tree that represents the current state of each block. There are same amounts of NODE struct in TREE to store the position of the corresponding block in the b*-tree.

NODE struct stores the parent, left and right child and whether of not the associated block are rotated or not. All of this information are compression into a 32-bit integer, where 10*3 bit are used to store parent, left and right child and 1 bit are used to store the rest. As show below:

```
constexpr uint msk_l = ((1<<10)-1)<<1;
constexpr uint msk_r = ((1<<10)-1)<<11;
constexpr uint msk_p = ((1<<10)-1)<<21;
constexpr uint rmsk_l = ~msk_l;
constexpr uint rmsk_r = ~msk_r;
constexpr uint rmsk_p = ~msk_p;
template<typename ID, typename LEN> struct FLOOR_PLAN<ID, LEN>::NODE {
  NODE() : x(0) {};
  ID _l() const { return (x>>01)&((1<<10)-1); }
  ID _r() const { return (x>>11)&((1<<10)-1); }
  ID _p() const { return (x>>21)&((1<<10)-1); }
  bool _rot() const { return x&1; }
  void s_l(const uint& i) { x = ((x&rmsk_l) | ((i<<01)&msk_l)); }
  void s_r(const uint& i) { x = ((x&rmsk_r) | ((i<<11)&msk_r)); }
  void s_p(const uint& i) { x = ((x&rmsk_p) | ((i<<21)&msk_p)); }
  void s_rot() { x ^= 1; }
  int x;
};
```

(4) Problem and Discussion:

- How to make sure that at least one feasible solution are found and outputted?

I implemented two different methods to achieve this goal efficiently.

First:

Since the initial solution are always random generated, there is always some probability that the Fast-SA will not produce legal solution at last. At first there are two ways to tackle this problem, either let the temperature drops slower or perturb more times in a single iteration. However, it turns out that both problems only mitigate the problem without completely solving it. Furthermore, the program runs considerably slower since it wastes too much time in the low-temperature phase. So, I turned to another way: simply let the program reset the temperature after the iterations are too much. I conjecture that starting from an ordinary initial solution is better that trying to solve a bad initialized solution which probably are already trapped inside a local minimum. However, there are a small disadvantage of this method: there may be cases that simply needs many iterations to converge. If my program simply reset the temperature every k iterations for every time, my program may never solve this case. To avoid this, I increment k and the number of perturb in a iterations after every reset.

Second:

Usually the SA stores the best solution when the current cost is lower that the best one. This doesn't implies that the stores solution are feasible. Thus, I changed this so the computer will only store the best solution when the cost are lower and the solution are feasible. All best cost always correspond to a particular feasible solution, indeed.

- How to improve the actual cost since my program use a different cost function?

During Fast-SA, I used the cost function as suggested in the paper which has an additional aspect ratio term to guide the SA process toward fitting the outline. However, this is different from our ultimate cost function. Without considering this issue, we will have no control over the quality of the generated solution. I tried to solve this problem by running a second stage Fast-SA which is the same as the one-stage Fast-SA only that the cost function are the true cost function normalized altogether:

$$\frac{\alpha A + (1-\alpha)\mathrm{HPWL}}{\mathrm{avg(total\ cost)}}$$

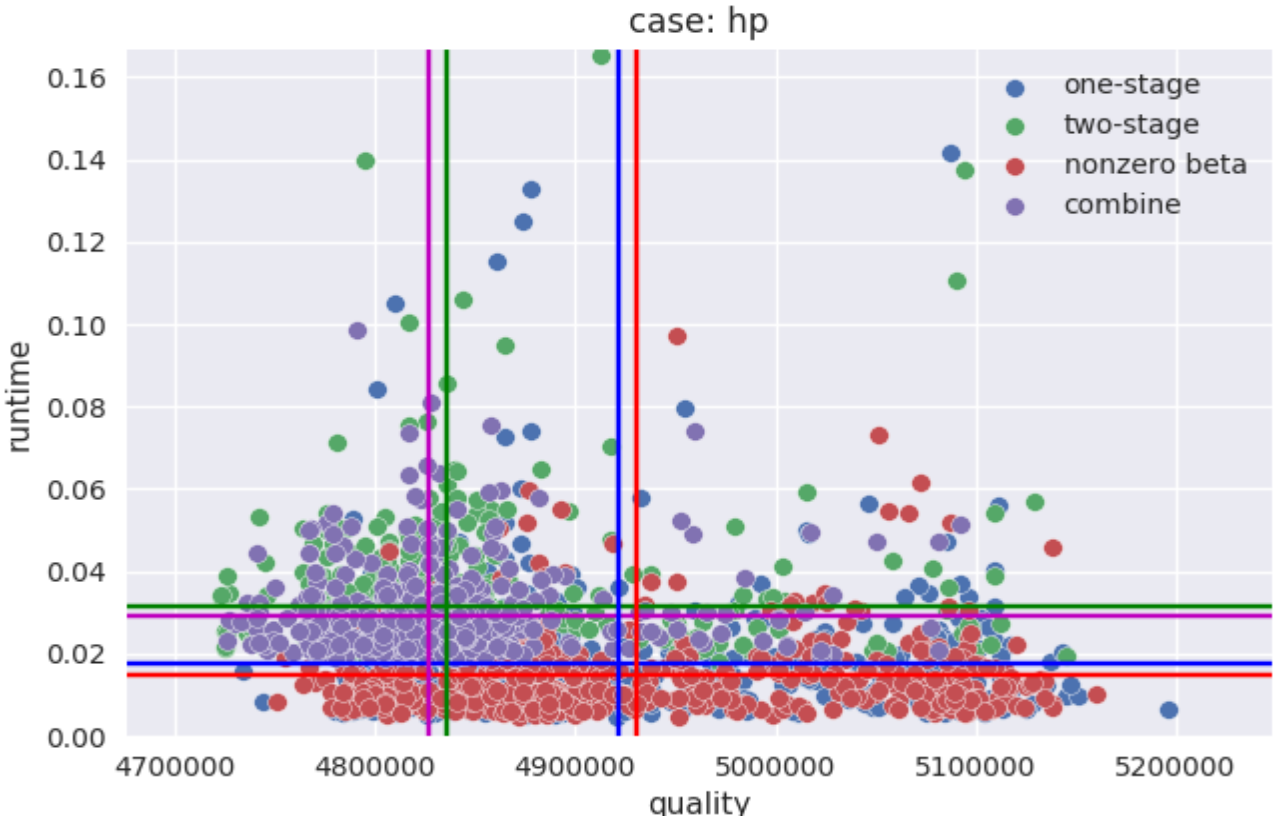
Where average total cost can be approximated before the process begin. Also, if we set the temperature the same as the Fast-SA, the high temperature will essentially break down the feasible solution found in the first stage and will eventually be pretty difficult to find a new solution with this new cost function. So the initial temperature in stage two will be 1/50 of the initial temperature in stage one to preserve the discovered legal solution in stage one. To prove that this method effectively improves the final solution, I conducted the following experiment:

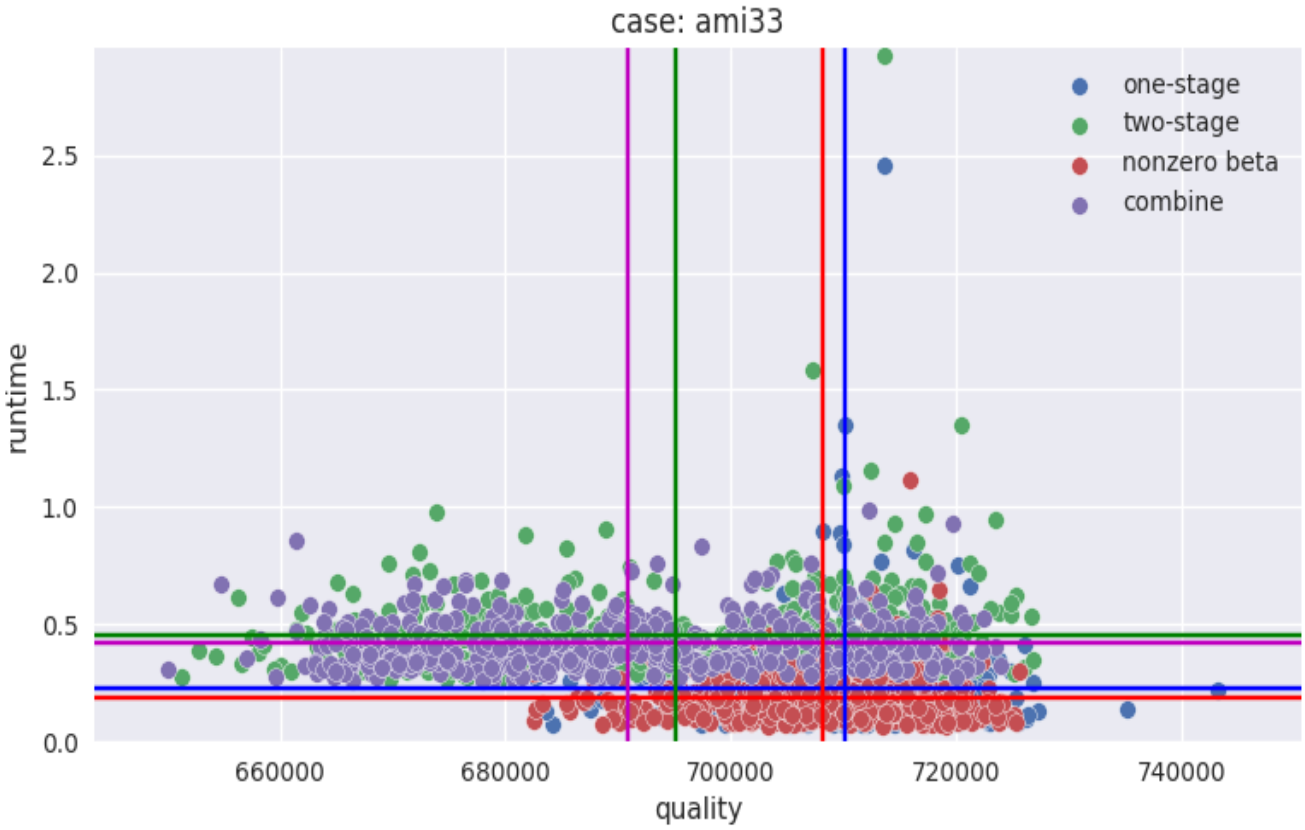
1. Compare four methods on all benchmark:
   one-stage Fast-SA with beta = 0 (one-stage, blue),
   two-stage Fast-SA with beta = 0 (two-stage, green),
   one-stage Fast-SA with beta = 0.1 (nonzero beta, red),

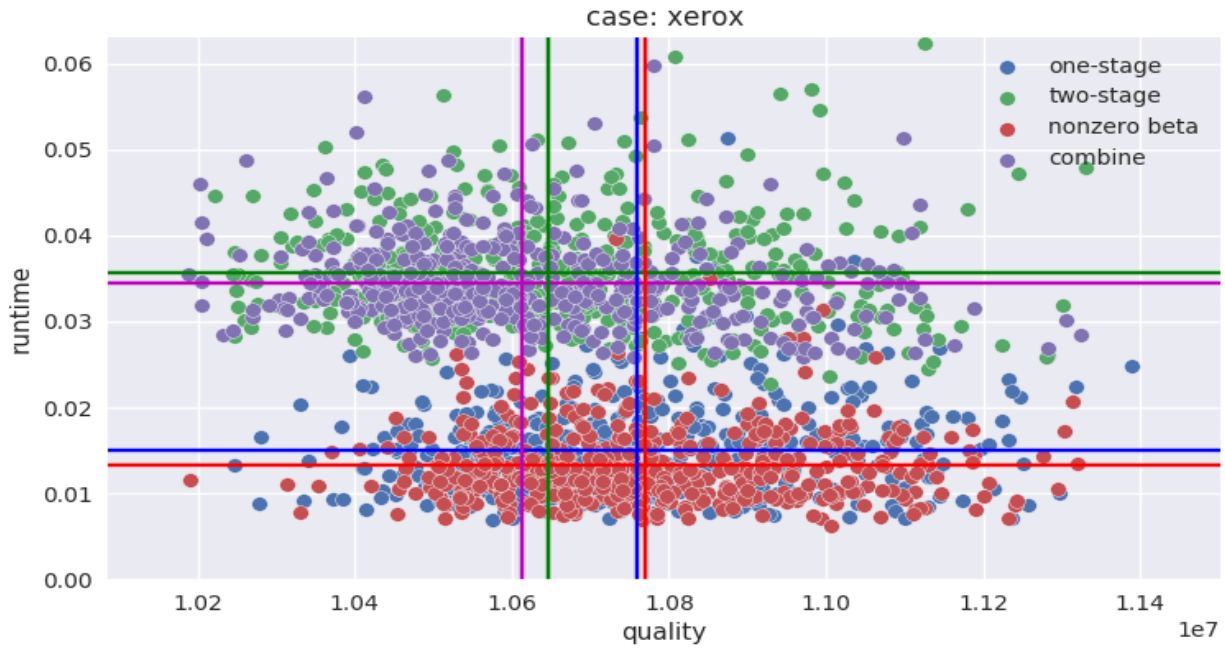
two-stage Fast-SA with beta = 0.1 (combine, purple)

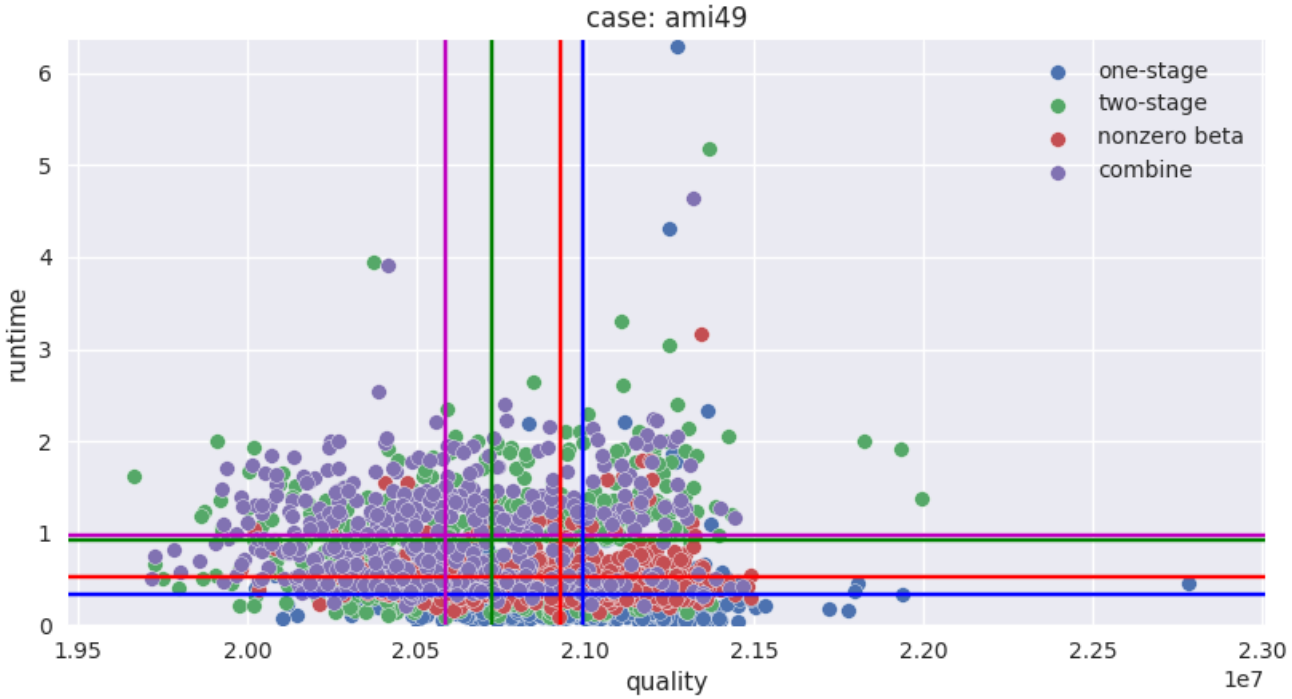
2. Run 500 times where all four cases have the same 500 random seeds. Every time the program run two epochs of the algorithm and select the best one.
3. The unit of runtime is seconds.
4. Alpha = 0.5 in all cases.

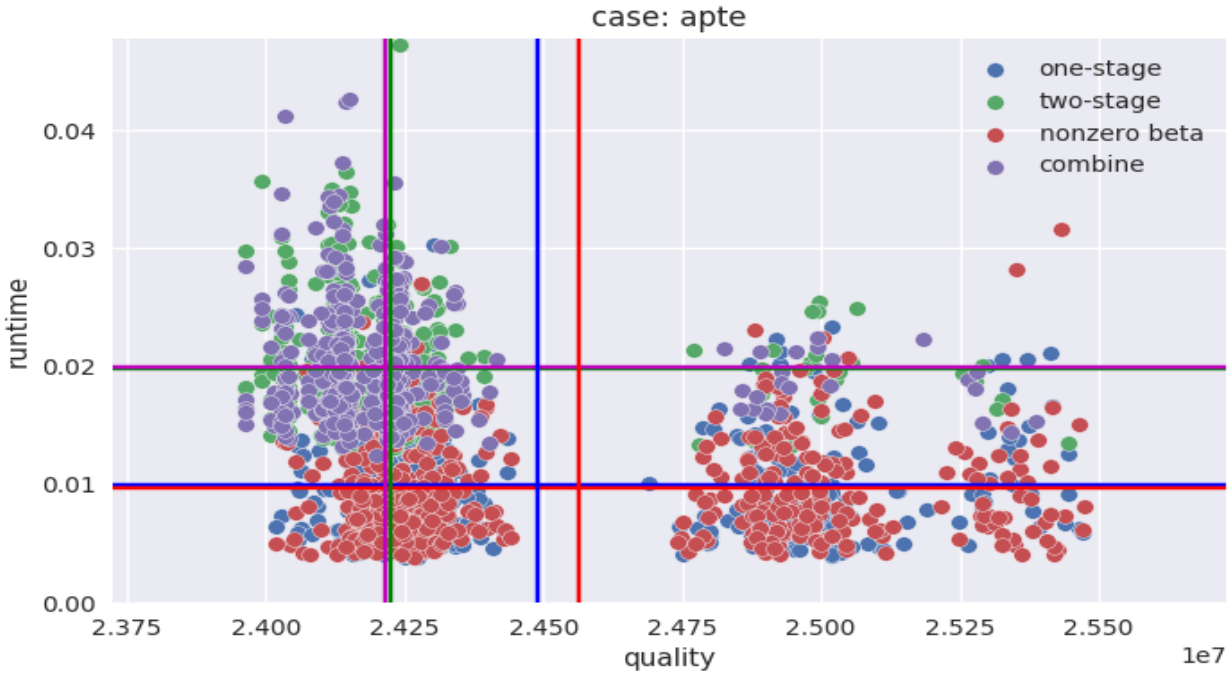
Results are shown below: (line indicate the average value)






case: xerox
one-stage
two-stage
nonzero beta
combine
runtime
quality
1e7
0.00
0.01
0.02
0.03
0.04
0.05
0.06
1.02
1.04
1.06
1.08
1.10
1.12
1.14


case: ami49
one-stage
two-stage
nonzero beta
combine
runtime
quality
1e7
0
1
2
3
4
5
6
1.95
2.00
2.05
2.10
2.15
2.20
2.25
2.30


case: apte
one-stage
two-stage
nonzero beta
combine
runtime
quality
1e7
0.00
0.01
0.02
0.03
0.04
2.375
2.400
2.425
2.450
2.475
2.500
2.525
2.550

Discussion:

We can see that in all cases, the second stage did improve the average solution quality about 2%. The tradeoff is the twice longer running time. One interesting thing is that adding the beta value without the two-stage process doesn't necessary produce better results (advance in two cases but deteriorate in three cases). This is expected since there is a bias between the true cost and the one used in Fast-SA, especially when one of the factor, normally the area cost, dominates. However, the combined result of the two-stage Fast-SA and nonzero beta value performs considerably well within basically the same runtime. This phenomenon is more apparent in cases with more blocks such as ami33 or ami49.

(5) Visualization:

I used to use gnuplot as my visualization tool. But one main disadvantage of it is that can only be executed outside of a c++ program so that it is difficult to use it efficiently and effectively. I tried to find alternative one and then I found the gnuplot-iostream which can be easily used almost the same as the original gnuplot but can be embedded within my codes. To show the temperatures, history of best costs and the final output, simply add "--plot" as the final argument to my executable file.

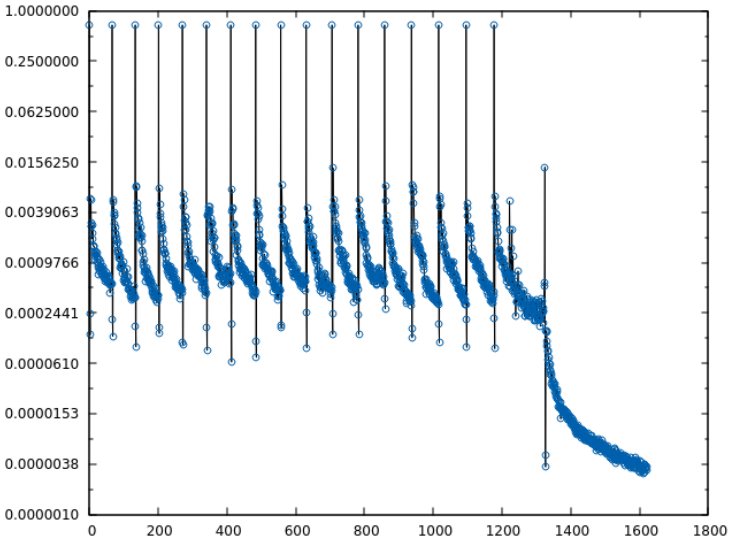
One result of ami33 and ami49 are shown below, where the temperature reset, the lower temperature in the second stage, the local greedy search stage in Fast-SA, the decrease of cost and the final output placement is observable:

p.s.

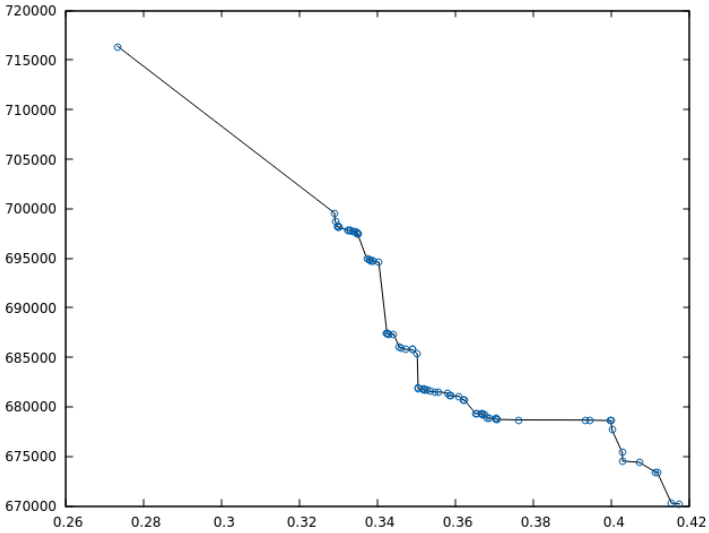
the x-axis of temperature is number of iterations, the x-axis of history of best costs is time in seconds, the blue circles outside of the outline is the fixed terminals.

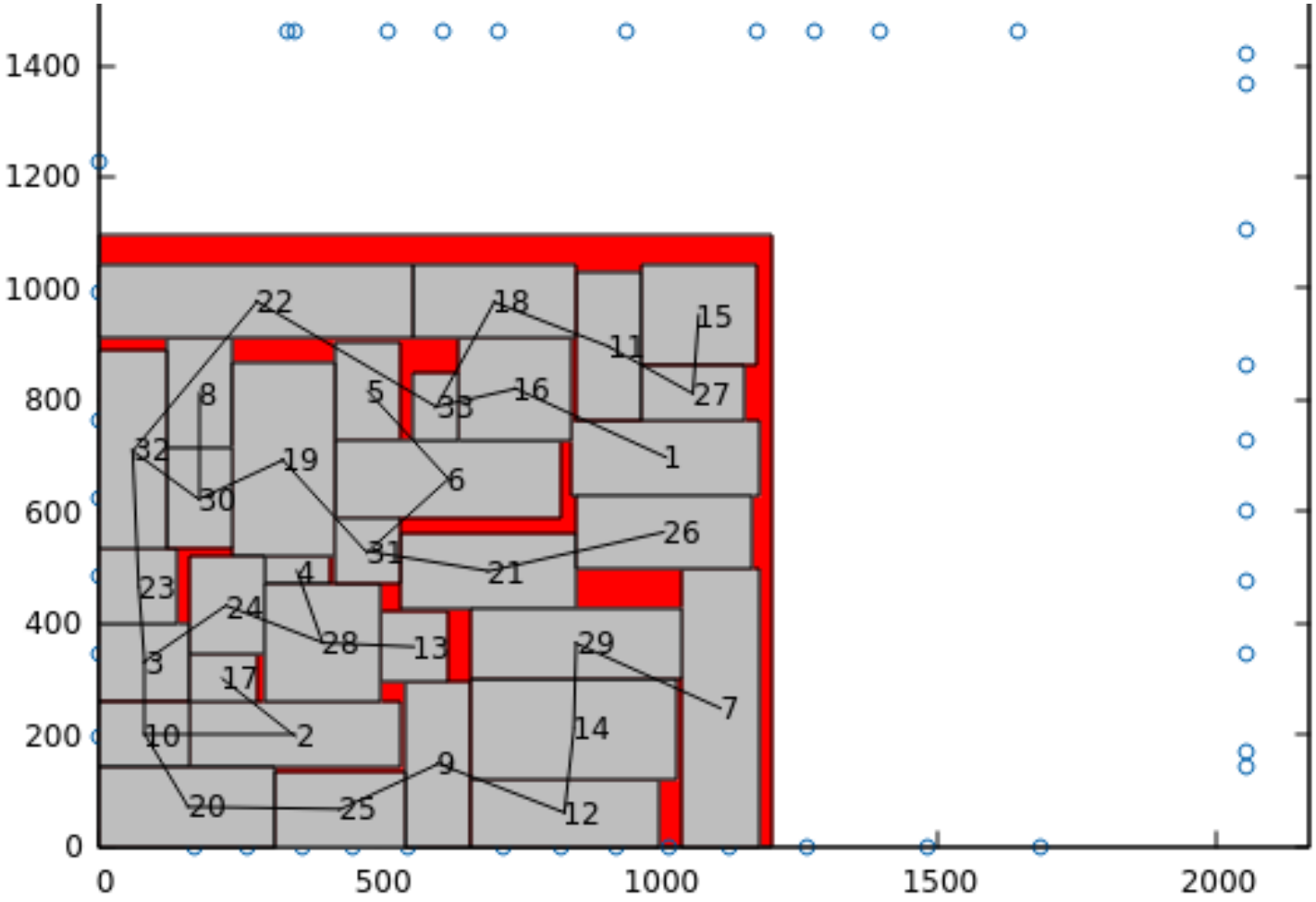
ami33:

Temperature:



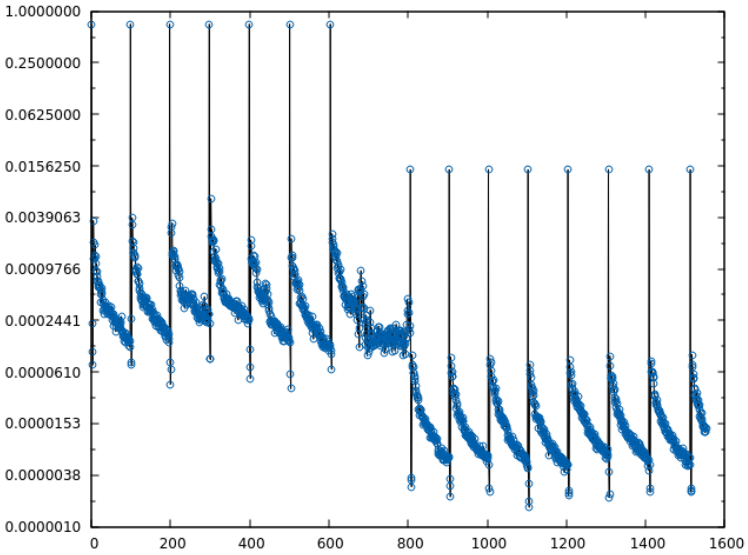
History of best costs:



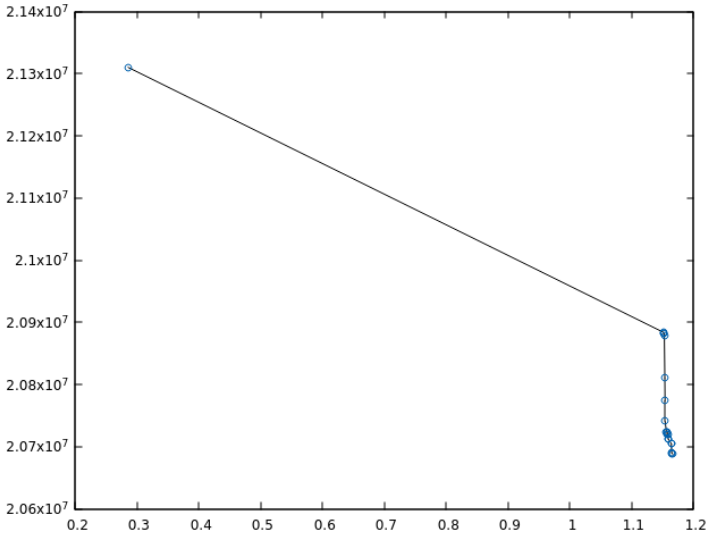


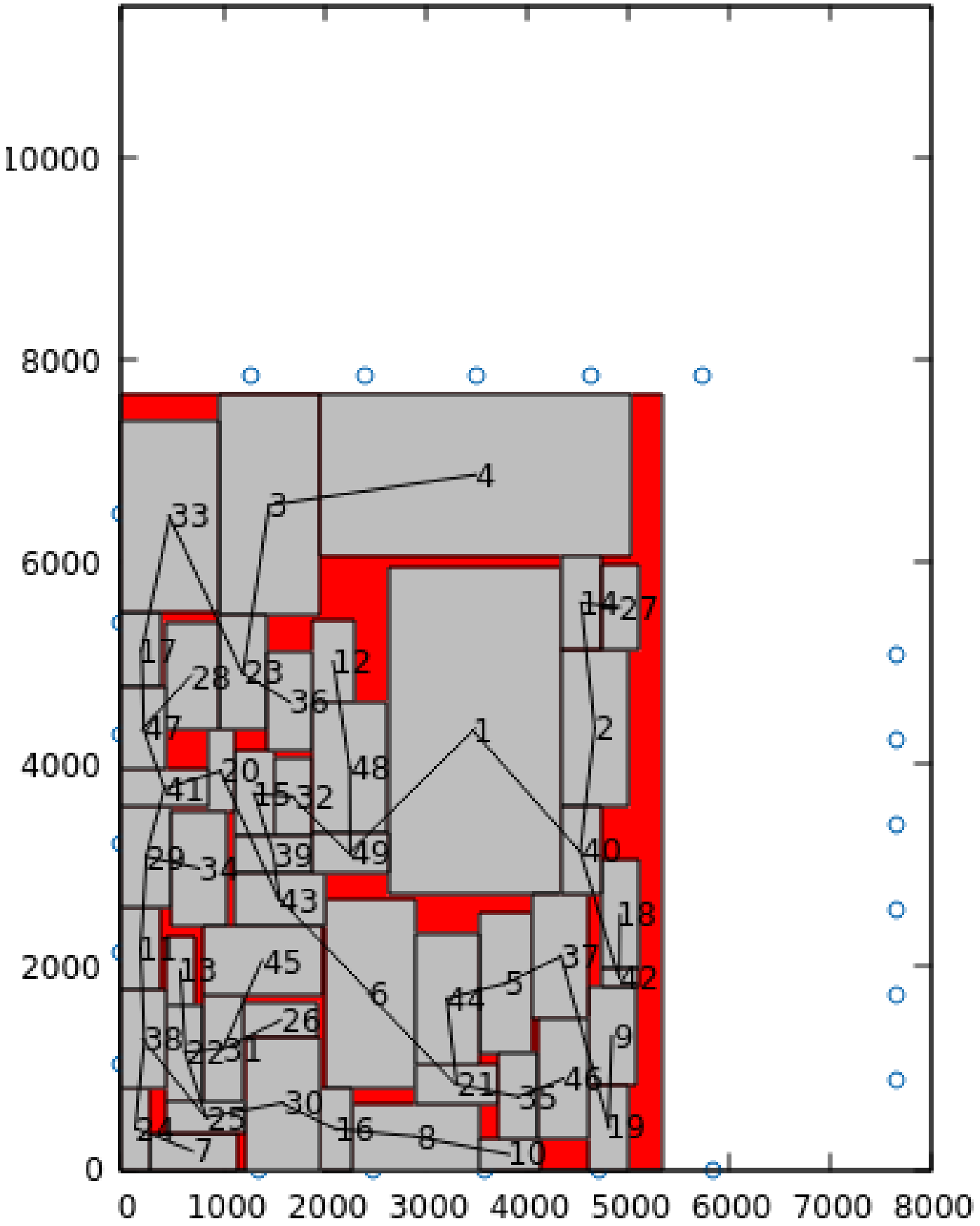
ami49:

temperatures:



history of best costs:




10000
8000
6000
4000
2000
0
0
1000
2000
3000
4000
5000
6000
7000
8000