Algorithm - Final Project Report

Topic: Color Balancing for Double Patterning (ICCAD Contest Problem E)

Team #: 24

Member & Assignment:

B99303103張博凱 0939098869

Colorable check (basically files other than shape.cpp)

Coloring algorithm

Project report

R03943141游 謙 0972164686

Read input file (basically shape.cpp)

Color difference calculation

Project report

Overview debug (basically GDalgorithm.cpp and shape.cpp are under his debugging)

B01504116陳則彣 0973191877

other algorithms(but not implemented this time)

B01502132黃璿諭 0932735583

Project report

Power point

other algorithms(but not implemented this time)

All components are implemented by us.

1. Abstract

In the Integrated Circuit Computer aided design Contest (ICCAD), there are five problems provided which include 3D-ICON: 3D Interlayer Cooling Optimized Network (problem A), Large-Scale Equivalence Checking and Function Correction (problem set B), Incremental Timing-driven Placement (problem set C), Routability-driven macro placement (problem set D, for promotion), Color Balancing for Double Patterning (problem set E, for promotion). Each problem deal with corresponding modern electronic design automation (EDA), and is important for electrical engineering students to think deeply and to have a glimpse at future electrical engineering and computer science task and potential solution.

Our group contains students from two different academic discipline including three electrical engineering students and one economics student who has great interesting at computer science and information engineering. From above 5 problem sets, we have decided to to choose problem set E as our final project goal which is much friendly vis-à-vis the other problem sets and can be properly applied our programming assignment three during algorithm forming to help ease the effort of designing and maintain efficient data structure.

In following section, we will introduce our algorithm in detail. To briefly talk about our algorithm implementation, we decompose our algorithm into some components, for example, reading input file, colorable shape checking, and main coloring algorithm. Next, shortly have a sight of some algorithms that was not adopted finally which was such a pity. They were provided by Huang and Chen separately, both EE students. Due to some correctness, timing, memory, and algorithm efficiency issues, we did not make one of them our final project solution. However, it is still worthy to present them in the report concisely to know how them function.

Note that the above algorithm introduction does not just show how to do the code implementation, but includes the time complexity and memory usage analysis.

Finally, we conclude our paper report with experimental (simulation) result and future work. In the future work, we consider many aspects of our algorithm. For instance, good heuristic can lead to better run time, reinterpret the input data given by ICCAD congress can have some insight of circuit configuration and input and help predict the large case trend apart from extreme cases, reconstruct the data structure to give a more efficient and powerful algorithm with smaller leading coefficient, especially for large input cases, and with little constant left for medium-scale input cases.

1. Algorithm implementation
   1. Input file reading

This part of the program reads input file and computes which two shapes should be edged together. The time complexity of this part is .

Corresponding function calls:

In shape.cpp:

ProtocolAndGroup Classifier::classify(int arg\_num, char\* argvector[]) 🡪 read the input file and classify each shape to a group

void Classifier::help\_you\_class(int group\_Num, int first\_node, int second\_node, int first\_time) and void Classifier::help\_you\_ending(int total\_group\_Num) 🡪 used to write a gn#.dot file (# represents the number of the group) so that the program can read from those files (use the function calls we’re familiar with in HW3) for the following B-part.

* 1. Colorable shape checking

For the followings, we denote colorA and colorB (mentioned in the page: http://cad-contest.el.cycu.edu.tw/problem\_E/default.htm) by 1 and -1. We use breath first search (BFS) to search each group whether the group contains more than one shape. The program also alternately color the visited shape with feasible color.

Corresponding function calls:

void reRead(int argc, char argv[\_ParsingNum\_][\_BUFFSIZE\_], Graph\* ourGraph) 🡪 read and formulate a graph

bool detect\_2coloring(int argc, char argv[\_ParsingNum\_][\_BUFFSIZE\_], Graph\* ourGraph) 🡪 used to determine the Graph\* ourGraph (the target group) is two colorable or not (true <=> two-colorable)

Then if a group passes the above test, such colorable group will be marked as colorable and initially colored as either 1 or -1, which is randomly selected. The time complexity of this part is

.

For the “n” in the big-O notation, note that for a single-shape group, this program can detect/judge that it’s two-colorable in constant time. Thus this program spends at most O(n) for those single-shape groups.

For those groups with at least two shapes, let’s divide the group into two varieties called odd cycle groups and non-odd cycle groups. The former one is not two-colorable which can be detected during the BFS and once detecting an odd cycle, the function will stop traversing the other nodes (shapes) in this group, and the latter one is two colorable which will be used later. Plus, the time needed for BFS traversal for a given group with V vertices and E edges is O(V + E). So, we have the above formula.

* 1. Main coloring algorithm

Note that right after part B, the program maintains:

For each group:

the vertices (indexed already) it has, and its corresponding (x1,y1) (x2, y2) values

In the followings, we denote each window by win(1, 1), win(2, 1) …., win(1, 2), win(2, 2)…win(1, 3) corresponding from the lower ones to higher ones and left ones to right ones. In addition, we say that a shape has contribution to a window if the shape has partial positive area contained in that window. Furthermore, a group has contribution to a window if there exists a shape in that group that has contribution to that window.

Thus, this program first search each window and find groups that contribute high color difference. This part is done by (actually it can be done by the way in partB; however we just implement it in another module):

For each group:

For each vertex in the group:

1. Check whether it’s of type1 or type2 or type3 or type4 ---- (note1)

2. Due to its type, calculate its contribution to windows ---- (note2)

3. To each window receiving contribution from this group:

Insert the contribution-difference and the contributor (the shape’s index) to the contributor list held by each window ---- (note4)

(Note that each group is initially colored by some color, e.g. color 1 for some group, and then coloring the whole group for the remaining shapes by BFS traversal; for the above group, we call its group-color is 1)

For each window (from win(1, 1) to the last window):

while (there’s a group such that: flipping its group-color reduces the most color density currently)

then flips/changes its group-color

lock the group so that there will be no consideration for flipping this group’s group-color thereafter

lock all groups that have contribution to this window

Thus, whenever there’s no such group in the wile loop mentioned above, the program continues to next window for the same iteration. After each window is computed, we will check the score.

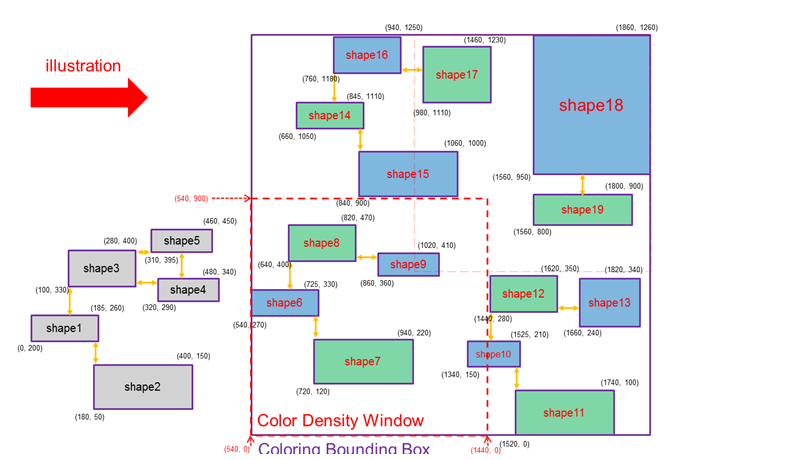
(note1):

A shape is of type1 means the shape is totally contained in the window with no area contained in any other window. e.g. shape7 in Pic\_00 is of type1

A shape is of type2 means the shape has exactly crossed vertically for at least two windows. e.g. shape8 is of type2 since its area is exactly (partially) counted in win(1, 1) and win(1, 2)

A shape is of type3 means the shape has exactly crossed horizontally for at least two windows. e.g. shape10 is of type3 since it has contribution to win(1 ,1) and win(2, 1) exactly

A shape is of type4 means the shape has crossed horizontally and vertically for at least 4 windows. e.g. shape9, very tricky, is of type4 since it has contribution to win(1 ,1) by (1020 - 860) \* (410 - 360), to win(1, 2) by (1020 – (1860 - 900)) \* (410 - 360), to win(2, 1) by (1020 - 860) \*(410 – (1260 - 900)), and to win(2, 2) by (1020 – (1860 - 900)) \* (410 – (1260 - 900)).



Pic\_00

(note2):

Since it’s very hard to determine the type of, say shape like shape9 originally, the implemented codes actually divide shapes into “seemly typei” (i from 1 to 4), where some grids are implemented to “roughly” divide the types. Take Pic\_00 as an example. There are some functions implemented in GDalgorithm.cpp to calculate the grids of the given bounding box:

Corresponding function calls:

int\*\* Color\_2coloring::CalGrid(int\* boxbounding, int omega) 🡪 returning grid (where grid[0][0] means the total grids in the horizontal direction and grid[1][0] means the total grids in the vertical direction; grid[0][i] is the value of the i-th horizontal grid and grid[1][j] is the value of the j-th horizontal grid; grid number is counted from 1) ---- (note3)

And then, since this program uses this kind of grid in determining the type of a shape, there exist some exceptions:

e.g. shape9 is classified as type1 – although it’s of course of type4

Plus, (note1) and (note2) are dealt by (in GDalgorithm.cpp)

void each\_Area(int\*\*\* eachArea, int\*\* grid, Shape\* Data, Graph\* ourGraph, int\* boxbounding, int omega, int first\_assignment, int\*\*\* Contributor\_list, int\*\*\* Area\_list, int which\_gp) 🡪 used to roughly determined the type of a given shape and calculate its corresponding contribution in each window

int\*\* test\_InBound(int\*\* grid, int omega, int which\_node, int type, Shape\* Data, Graph\* ourGraph, int leftest\_grid, int rightest\_grid, int toppest\_grid, int lowest\_grid) 🡪 each\_Area calls this function call to exactly give the type of the given shape and calculate its contribution for those windows omitted originally in each\_Area()

e.g. shape9’s contribution to win(2, 1), win(1, 2), and win(2, 2) will be calculated here.

In a word, test\_InBound() deals with those windows omitted (but should receive contribution from the given window) in each\_Area() and calculate the contribution of the given shape to them.

(note3):

Take Pic\_00 as an example. The grid returned:

grid[0][1] == 540, grid[0][2] == 1440, (thus) grid[0][0] == 2;

grid[1][1] == 0, grid[1][2] == 900, (thus) grid[1][0] == 2;

(note4):

This part is implemented in

“void Add\_area\_perShape(vector<int>& which\_window\_x, vector<int>& which\_window\_y, vector<int>& area\_sumShape\_neg, vector<int>& area\_sumShape\_pos, int additional\_area, int leftest\_grid, int lowest\_grid, int ind)” this function call (still in GDalgorithm.cpp), which will be called in each\_Area(). In our settings, each window basically keep 2 lists – one is the top-10 contributors, and another is their corresponding contribution-difference.

The contribution difference of a group G to a given window is calculated as:

Color difference contributed by G

e.g. to win(1, 1) in Pic\_00, the contribution difference of the group containing shape12 (from shape10 to shape13) is (0-(1440 - 1340) \* (210-150)) + 0 + 0 + 0

Since the program, for each window, only maintains a top-10 contributor list and its corresponding color-difference contribution, thus the insertion can be viewed as constant time.

The implementation of the insertion:

First, the “top-10” means the top -10 color-difference contributors in absolute value. For instance, say, to some win(i, j) it may has

<5, 4, 11, 1, 99, 15, 98, 100, 22, 518> as its top -10 color-difference contributors and <100, -90, 85, -80, 70, 60, -50, -40, -30, 20> as their corresponding contribution difference. And this part (insertion) is implemented by

void Cal\_contribution(int\*\*\* Contributor\_list, int\*\*\* Area\_list, int x\_grid, int y\_grid, int gp\_id, int diff) 🡪 the gp\_id means the group for inserting the contribution-difference to other windows (if it can be listed on top-10)

III. Algorithm implementation – different views of C part in II

* + 1. Coloring bounding box view

In the macro view of our algorithm, we follow the traverse order of the problem. As the bellowing picture shows, we move our arrangement from left to right and from down to up. Therefore the time complexity can be shown as , where n is denoted as the number of the coloring density windows. Clearly we do this to minimize the pre-sorting pressure on memory usage and timing and to get stable running time that increase linearly in accordance with the quantity of the input coloring window, but at the cost of low precision of the afterward greedy algorithm that can powerfully minimize the color difference of single color density window but not promise the neighboring.

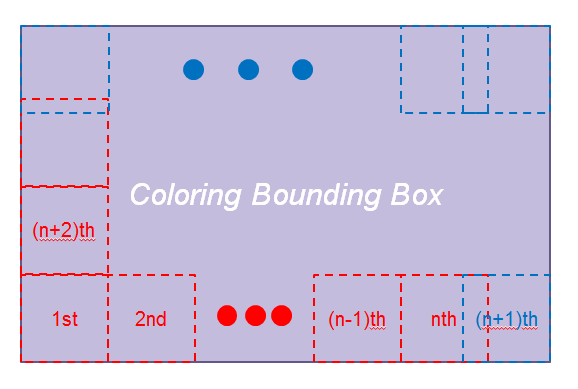


Figure 1. Traverse order of our heuristic

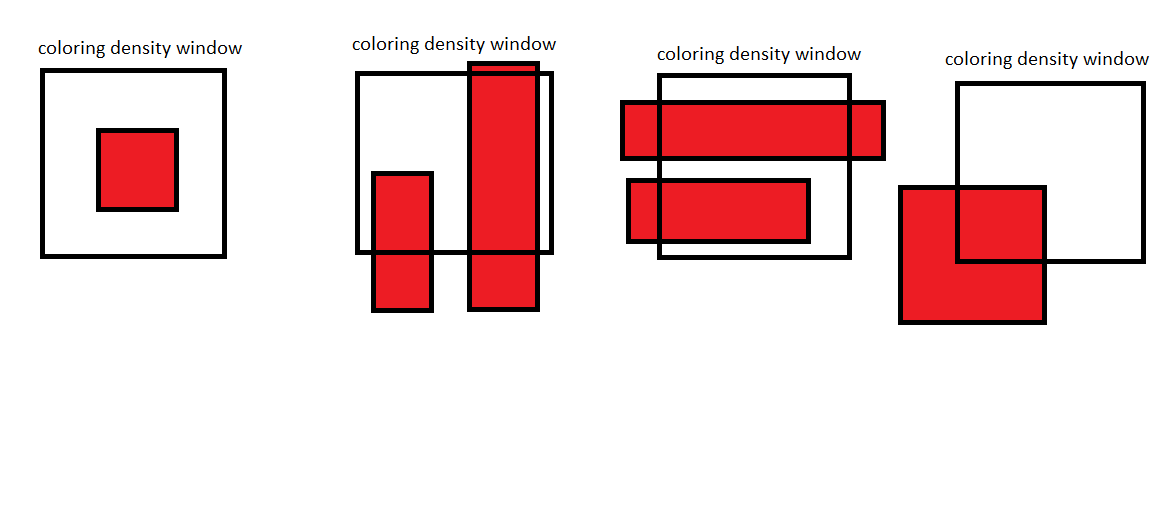
Note that the order of chen’s algorithm (we provided in the later section) does the traverse in a similar way. It applies pre-sorting on each coloring density window on the basis of the number of cross-window groups. In this way, it first travels all the legal vertices and adds cross-window group count while the shape (node) shares some of its area on different window. After collecting all the cross-window components (actually record them as count instead of has a pointer on the group of the node), sort the window by the cross-window count into a list. Afterwards start the iteration based on the list. In such way, most of the additional cost lies on the sorting procedure, which can be optimized when using the sorter provided by the C++ library, that is, the time complexity of the sorting procedure is optimized to , i.e. in mathematic form ,where n means the input size, in this case the input size is equal to the number of the coloring density window.

* + 1. Coloring density window view (the below is basically explained previously, you may omit this part)

After determine the traverse order of the whole coloring bounding box, we start the iteration. We do some jobs for minimizing the color difference of the color density window which is critical for gaining good grades on the contest in each iteration. At the beginning of the iteration, we traverse the whole coloring bounding box to find which group is in or cut in the coloring density window and which is not. Then collect the former type of groups.

Before explaining the complete algorithm for this part, we classify the 2 type of the groups that have contribution on the color difference of the coloring density window as follows, inside-window type groups and cross-window type groups. The former type of groups are defined that its vertices (shapes) are totally covered in the coloring density window and it can be easily calculated the contribution of the color difference which is either a positive or negative number in so far as the two-coloring style may be. The latter ones are identified that it has only some part of its shapes across the boundary of the coloring density window, which contributes some portion of the color difference to the coloring density window. It can be shown that the latter ones have contribution of color difference over many coloring density windows. In an extreme example, the connection (metal) of the VDD or the ground located in the different standard cells can stretch over a large number of standard cells to ensure the small voltage difference between each VDD or ground and stable voltage reference point. In our algorithm design, we determine the latter ones with the first encountered coloring density window and lock this allocation until finish the iteration. The lock message (signal) will be explain soon.

The above two types of groups are required to be treated in different way while finally get the color difference contribution to the chosen coloring density window. We have explained how to get the information of color difference contribution from the former type. Next, to determine how to get the similar information from the latter ones need taking traverse over the total nodes of the group and calculating the color difference contribution after specifying the following 4 types of shapes. The key element to the identification is to treat the boundary of both shapes and the coloring density window carefully. This work takes little effort on comparison. Until now, it is enough to decide the time complexity. Because we lock the variation of the color allocation after first visit, it is easy to see that each color difference information is calculated once and is used once. Therefore we can determine the time complexity ( ), that is, , where n is called the total count of the legal shapes. Despite the fact that the cross-window groups will be traveled more than once, the lock message block the request of changing the color allocation but allow having a copy of color difference. Obviously the above mathematic expression is approximation because it assumes little crossing-window groups, but it is accurate enough for the official test cases.



(d) Two-dimensional violation group

(c) X-direction violation group

(b)Y-direction violation group

(a) Inside-window group

Figure2. Color contribution of different kinds of crossing-window shapes

With the tool of calculating color difference in a coloring density window, we then sort them into a list from high difference to low difference. Before preceding the following discussion, let’s clarify our procedure of the heuristic. At the stage of calculation, we first find how many groups can contribute the color difference of the coloring density window. Next, random assign the color allocation obeying the two-coloring rule. In fact, we just implement it as first coloring type is found in the breath first search without lock signal. With the color style determined, the next work is to sort the color difference area (taken as the form of absolute value) which runs time into a list and record the current color difference area. Still, we will not change the color difference that is lock and only provide the color allocation optimized when first explored.

Finally, we do color assignment according to the list. Along with the current area difference[[1]](#footnote-1) and the area difference list, we flip the coloring type as long as the flipping result will reduce the area difference. The time complexity would be , where n is denoted as the total number of the group defined above in this coloring density window, if the area difference percentage has dropped below a given error threshold. Otherwise, the iteration will run through the whole groups on the area difference list.

We visualize this algorithm in coloring density window view as follows.

Step1. Before iteration I

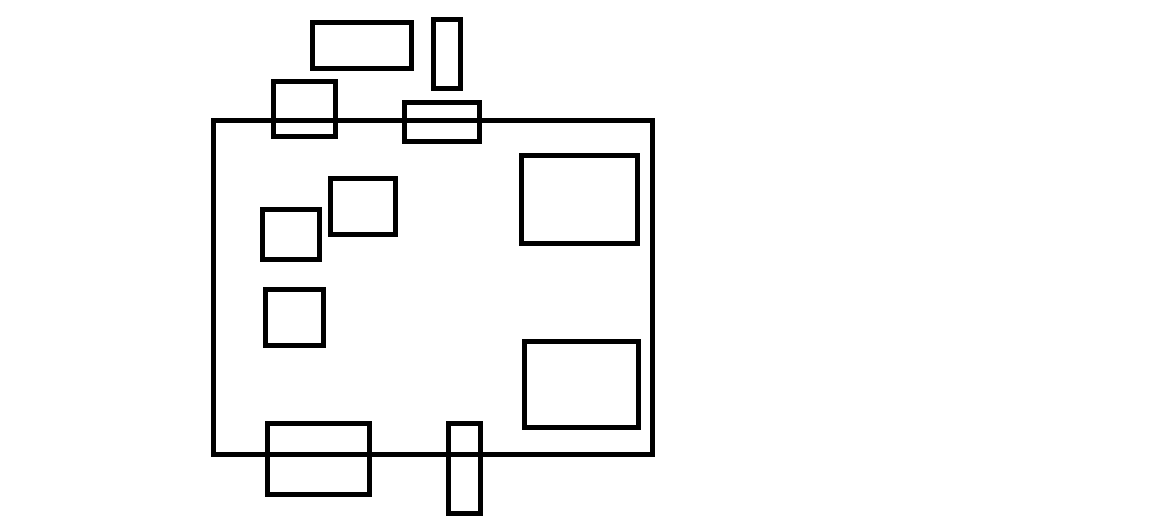


Figure3. Row graph

Step2. Before iteration II

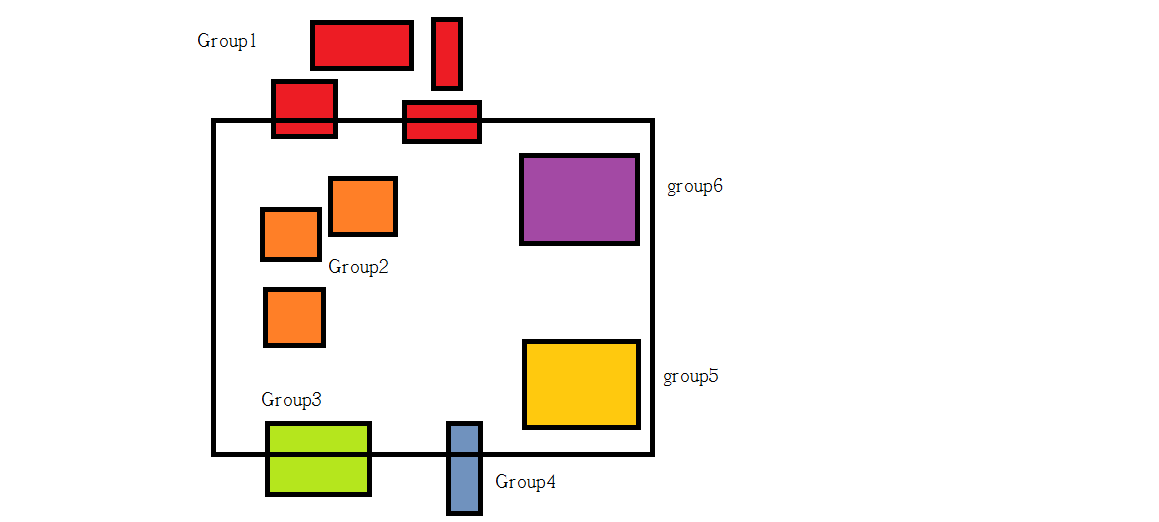


Figure4. Identify different groups and mark

Step3. At the beginning of the iteration

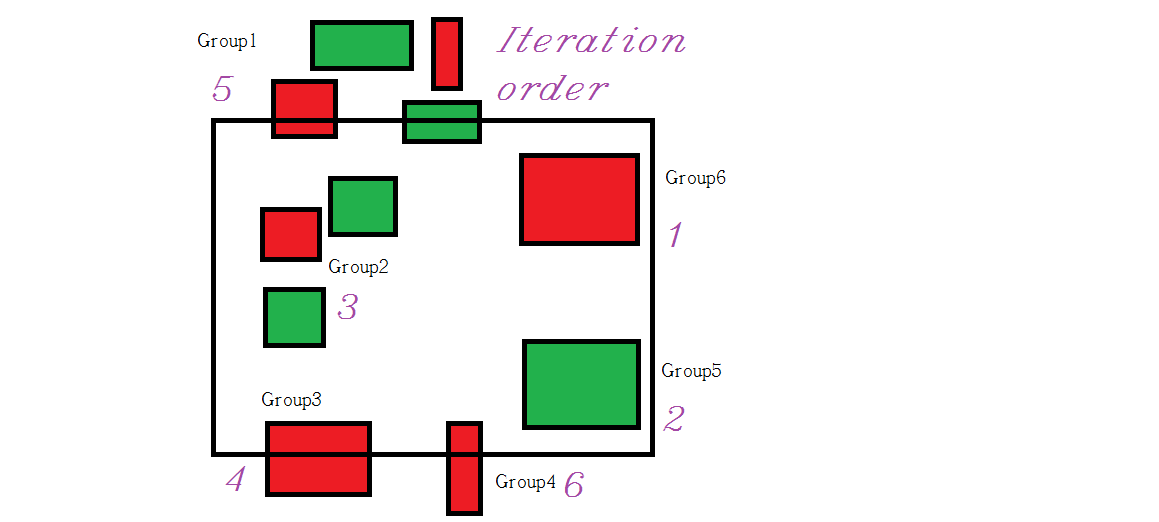


Figure5. Determine the iteration order according to the color difference area contribution list

Step4. Execution of iteration: order 1

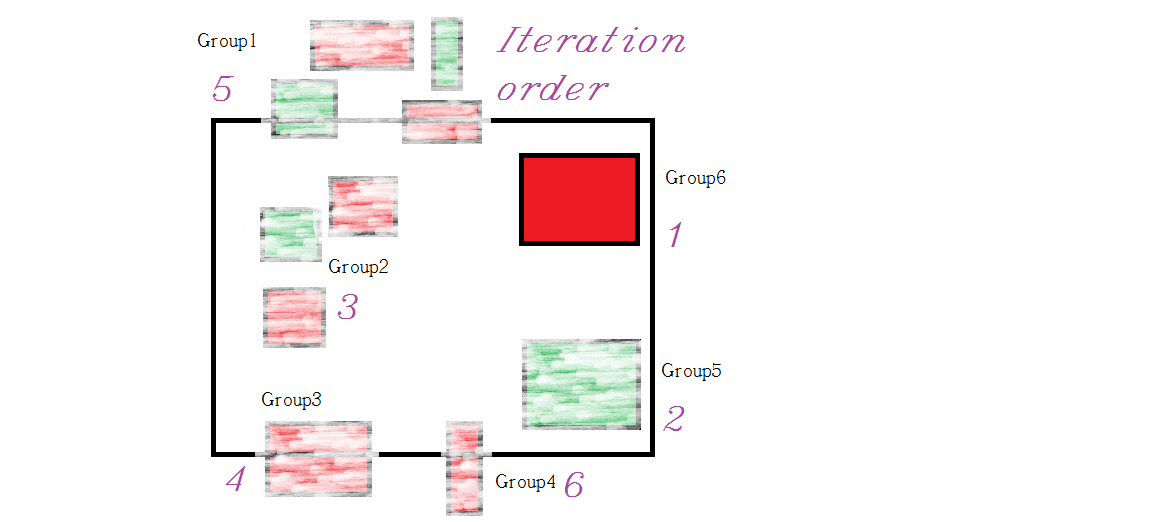


Figure6.

Step5. Execution of iteration: order 2

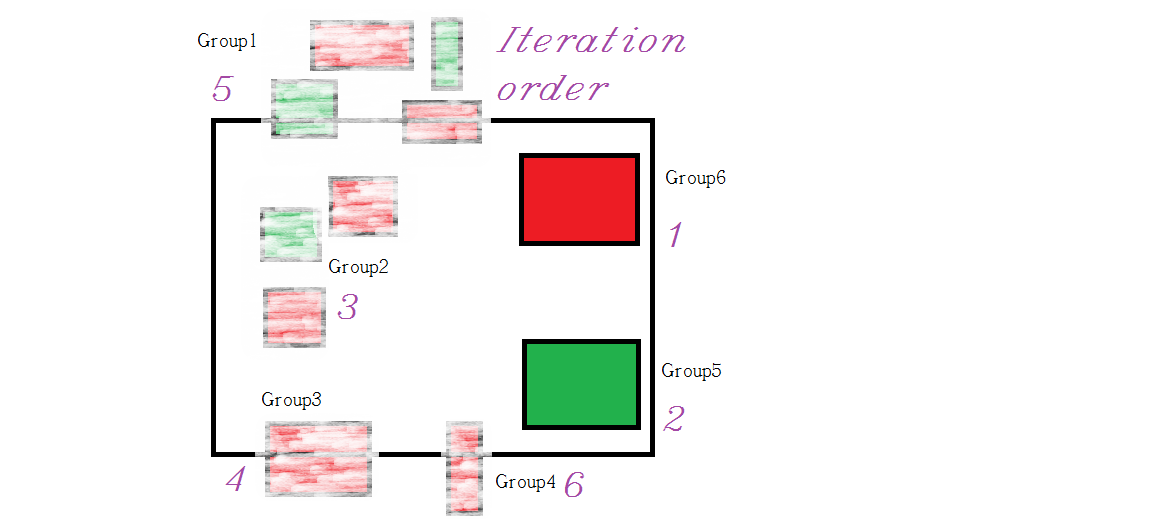


Figure7.

Step6. Execution of iteration: order 3

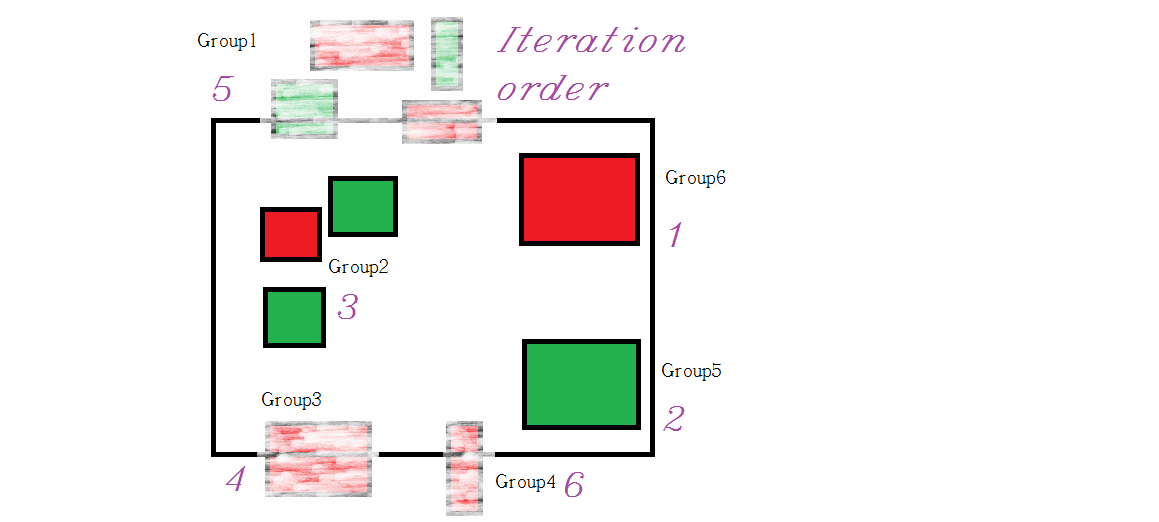


Figure8.

Step7. Execution of iteration: order 4

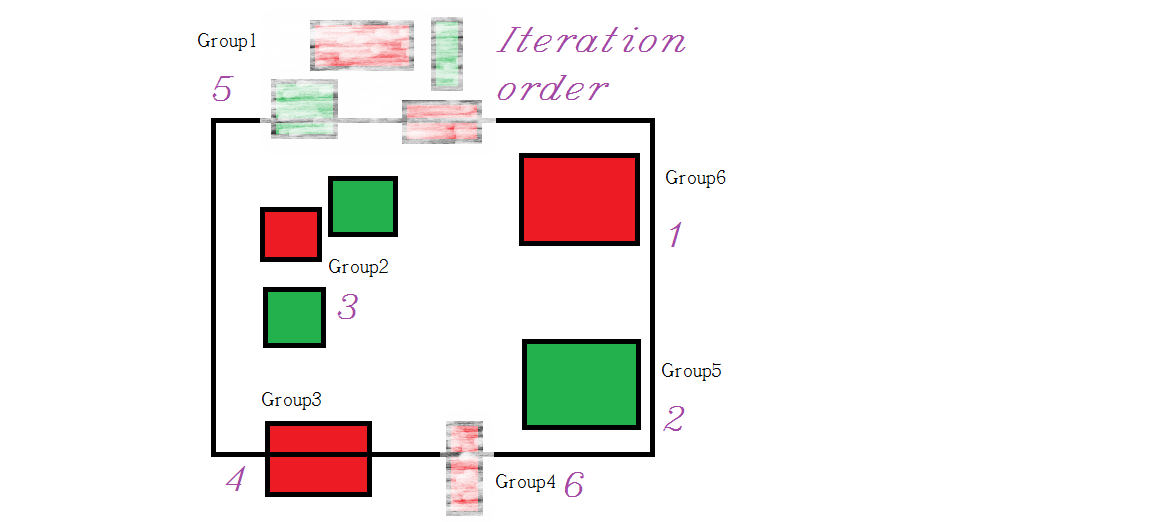


Figure9.

Step8. Execution of iteration: order 5

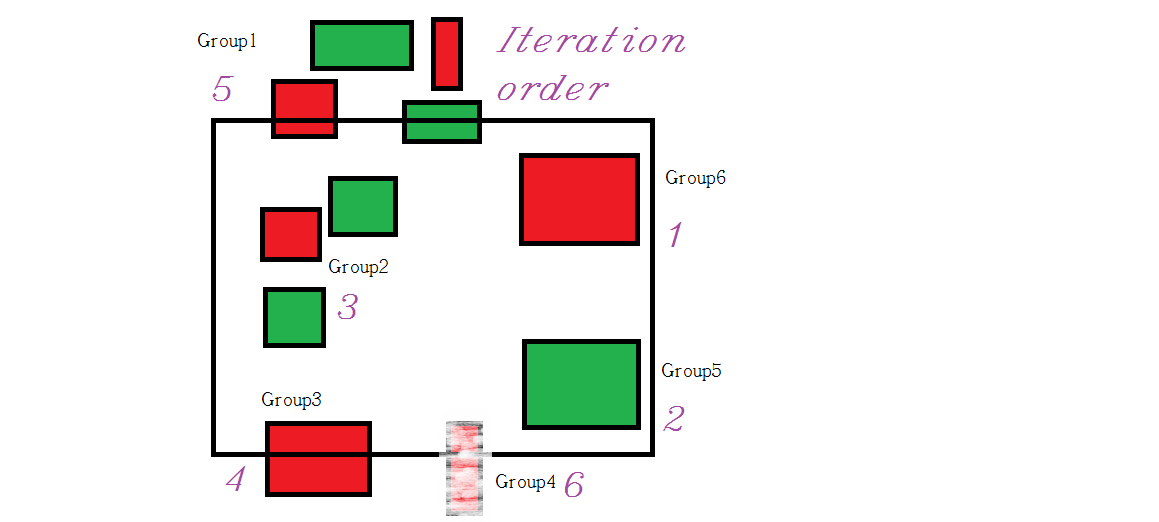


Figure10.

Step9. Execution of iteration: order 6

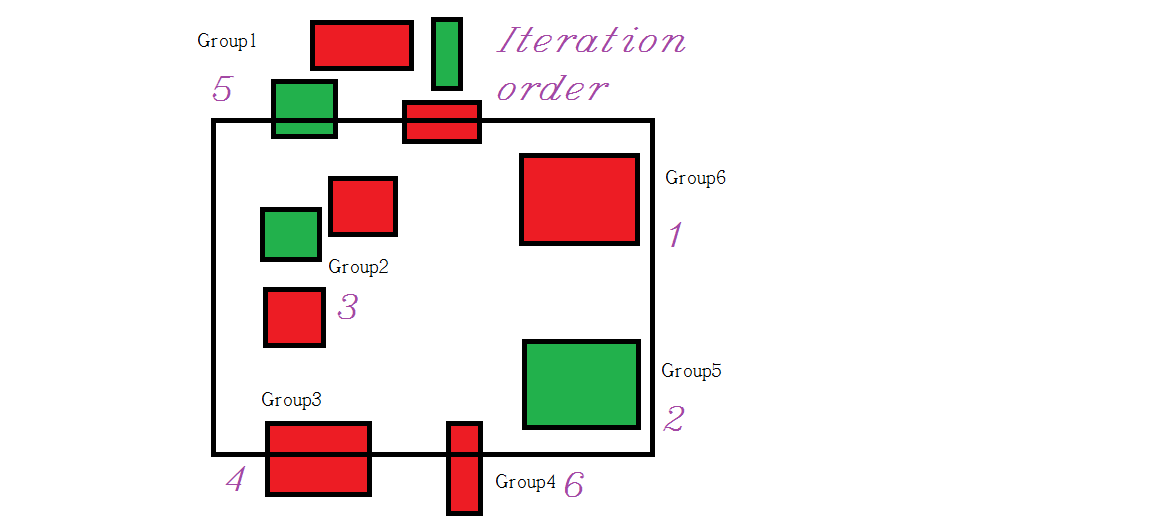


Figure11.

1. Experimental result
2. Other no adopted algorithm

(1)Determine which window to color first:

Because of the properties of this problem as a way to minimize the different color area in each window, we can assume a heuristic that the order of coloring to each window may affect the final result.

So, how should we decide the order of coloring? First of all, group that crosses over multiple window is an important issue, which drastically makes problem complex and hard to resolve. We therefore try to sort the order of coloring according to how many window-crossing group is in each window.

Example: a window with two window-crossing group

(2) Coloring each window

After deciding the priority to color each window, the problem is suddenly simplified. Now all we have to do is trying to make a window balanced with two different color.

And for the problem here, we only discuss about 2-colorable group. Assuming a window have total N groups, then it can therefore be 2^N comparison needed for brute-force method, not time efficient at all. As a result, we choose to utilize the property that if 2-colorable group swap color, the contribution to this window is actually -1 times the former one. Thus we can list our step for this coloring meth as below (assuming the two different colors is color A and color B, and two kinds of ways to color groups is called color-set 1 and color-set 2):

(a) Random assign color set for groups

In this step, the assigned color isn’t so important, only to get some initialization on this problem.

Therefore we choose color-set 1 for every cases.

(b) For each group, record it's (area of color A - area of color B), and name it as color difference of group

Sum of color difference of each group is exactly what we trying to minimize in this problem.

(c) Sort color difference of each group, plus get a sum of total color difference

(d) Try to minimize the sum

To achieve this, we look for color difference in decreasing manner, if

( sum- current difference \*2 ) is closer to zero, we change the color-set of this group (set 1 to set 2), then replace sum with ( sum- current difference \*2 ).

Otherwise, loop until the function end. As a result, what we get should be a pretty good solution to this window.

(e) Repeat the whole process on other window, till all groups are colored

Note: If later window see a group color set decided by the former one, then the color set should be kept intact.

1. Future work

(1) Improvement of heuristic

Maybe based on new idea or trial and error method we can come up with better heuristics to replace the old one. But considering the limit time of this project, the heuristic we found above is most suitable for our current condition, plus having a not bad performance.

(2) Add algorithm that "improves" the result such as Simulated Annealing

We didn’t adapt Simulated Annealing for the first place because the result may be really random and takes a really long time.

But maybe we use the value we get from previous introduced algorithm, and let the result improves over time. This may come up with a better solution in an acceptable time.

**Experimental result**

In the followings, the first (per-line) < … > contains the initial assignment of the group-color per legal group sequentially of the given problem. (e.g. for the legal group, the representing legal group of Pic\_00 is group2, 3, 4, 5 respectively) And, the (still, per-line) second < …. > represents the group-color sequence (from lowest legal group to the highest legal group) modified by this algorithm.

Thus, we call the < …. > as group-color sequence.

"iccad2015\_case1"

<1 1 -1 1 1 -1 -1 -1 -1 -1> ==> <1 1 1 -1 1 1 -1 1 -1 -1>

total score = 83.978790 (before modified by this algorithm: 88.953370)

runtime = 0.000261 sec

memory = 12.768000 MB

"iccad2015\_case2"

The group-color sequence is omitted (since 45 legal groups – too many to list here)

total score = 88.260880 (before modified by this algorithm: 85.582540)

runtime = 0.001956 sec

memory = 12.768000 MB

"iccad2015\_case3"

The group-color sequence is omitted. (too long, 4 hundred more legal groups)

total score = 104.552420 (before modified by this algorithm: 88.183000) (note5)

runtime = 0.007512 sec

memory = 13.032000 MB

"iccad2015\_case4"

The group-color sequence is omitted (since 45 legal groups – too many to list here)

total score = 95.581190 (before modified by this algorithm: 95.123160)

runtime = 0.001016 sec

memory = 12.768000 MB

"iccad2015\_case5"

The group-color sequence is omitted. (too long, 5 hundred more legal groups)

total score = 86.449894 (before modified by this algorithm: 91.037568)

runtime = 0.037797 sec

memory = 13.036000 MB

**Future work**

1. Improve/invent more heuristics

2. We're going to generate more input data ourselves. The samples released by ICCAD 2015 Contest in the problem “color balancing for double patterning”, to us, was not too meaningful for debugging and choosing the best parameters to improve our algorithm's performance and robustness. (actually, at least we can observe that the shapes are too few compared with the window size, which leads to negligible amount of |color\_A\_density.d – color\_B\_density.d|/5 for each window)

source code:

<https://www.space.ntu.edu.tw/navigate/s/A6B83516AEAA4ADFB5E2635E5CDF469DQQY>

(note5): this is due to the formula for calculating score uses “| |”in adding difference, if we switch to no “| |” then: the value modified is more reasonable.

1. Follow the equation: ,where pos[i] is denoted as the ith valid group with the positive area difference in the coloring density window and winpos is denoted as the sum of the positive area difference in the coloring density window. [↑](#footnote-ref-1)