

ALGORITHM FINAL PROJECT
COLOR BALANCING FOR DOUBLE PATTERNING

湯智帆 b00404017

徐吉吉 t03902115

鍾瑞輝 b02202008

1 Introduction

The traditional lithography cannot print patterns at advanced nodes. Double patterning technology (DPT) is a method of breaking up a layout so that sub-resolution patterns are separated onto two distinct masks. Typically, the problem of separating layout patterns onto two masks is solved by transforming it into a 2-coloring problem. A balanced coloring would allow more space for scattering that leads to better patterning quality. Thus, balanced and uniform color density is preferred during layout decomposition. The objective is to minimize the difference between color-A density and color-B density as much as possible for all color density windows.

2 Build Coloring Graph

The input file is read in through *iostream* I/O function *getline*. We can get *iostream* to achieve much greater speed with a call to *std::ios::sync_with_stdio(false)*, since we do not need to synchronize with the C I/O functions.

We construct a class *Node* in the adjacency-list representation to store the information contained in retangle shapes (or vertices), such as their *x* and *y* coordinates of the bottom-left and top-right corner, the container used is *vector<Node*>*. Then we sort them with the value of *x* coordinate of the bottom-left corner in the ascending manner in order to later construct edges systematically.

Then there comes the most expensive operation of all three steps, generating an edge connecting any two vertices corresponding to two shapes satisfying *x*-spacing $< \alpha$ or *y*-spacing $< \beta$. We pass in *V* (sorted *vector<Node*>*) below to our implementation CONSTRUCT-EDGE. Class *Edge* is constructed to store edges in *E*

```
CONSTRUCT-EDGE (V,  $\alpha$ ,  $\beta$ )
1   for  $i = 1$  to V.size-1
2       for  $j = i + 1$  to V.size
3           if V[i] and V[j] is x-direction overlaped and  $x\text{-spacing} < \alpha$ 
4               construct edge (V[i], V[j]) to vector<Edge*>
5           if V[i] and V[j] is y-direction overlaped and  $y\text{-spacing} < \beta$ 
6               construct edge (V[i], V[j]) to vector<Edge*>
```

Noted that both *vector<Node*>* and *vector<Edge*>* are further stored in class *Graph* with functions to manipulate other object class and data members.

Complexity Analysis

How fast is Build-Coloring-Graph algorithm? Reading in file and iterating through all the vertices (shapes) have time complexity $O(V)$. Sorting takes $O(V \lg V)$ and CONSTRUCT-EDGE

takes $O(V^2)$. The total runtime is therefore $O(V^2)$.

3 Coloring

Apply a 2-coloring scheme to coloring graphs. Any two adjacent vertices must have different colors. If one has color-A, the other must have color-B. The problem can be solved in polynomial time with the breadth-first-search procedure *BFS*. In the meantime, we construct class *connected component (CC)* to categorize a minimum numbers of subgraph such that in this subgraph every pair of vertices u and v , we have $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is, vertices u and v are reachable from each other. Every vertices (or Nodes as we construct) in graph are attached several additional attributes. We store the color of each vertex $u \in V$ in the attribute $u.color$ (white means undiscovered, black discovered), the distance from the source s to vertex u computed by the algorithm in the attribute $u.d$, and the boolean attribute $u.e$ to determine whether the vertex is explored by BFS before.

COLORING (G)

```

1  for each vertex  $u \in G.V$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.e = \text{FALSE}$ 
5  for each vertex  $u \in G.V$ 
6      if  $u.e == \text{FALSE}$ 
7          BFS-Coloring (G, u)

```

BFS-COLORING (G, s)

```

1   $s.color = \text{BLACK}$ 
2   $s.d = 0$ 
3   $s.e = \text{TRUE}$ 
4   $Q = \emptyset$            //queue
5   $T = \emptyset$          //vector for temporary purpose
6   $C = \emptyset$          //class for connected component
7   $C.push\_back(s)$ 
8   $C.colorable == \text{TRUE}$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{Dequeue}(Q)$ 
12     for each  $v \in G.adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color == \text{BLACK}$ 
15              $v.d = u.d + 1$ 
16              $v.e == \text{TRUE}$ 
17              $T.push\_back(v)$ 
18              $C.push\_back(v)$ 
19             ENQUEUE( $Q, v$ )
20 for  $i = 1$  to  $T.size - 1$ 

```

```

21      for  $j = i + 1$  to  $T.size$ 
22          if edge  $(V[i], V[j]) \neq \infty$ 
23               $C.colorable == FALSE$ 

```

Complexity Analysis

How fast is Coloring algorithm? It appears that line 5 iterates through all the vertices, each calling BFS. But since the bookkeeping attribute $u.e$ nullify most of the iterations, line 5-7 is therefore equivalent to BFS the whole vertices. The total complexity is $O(V + E)$.

4 Calculating Color Density

We propose a $O(N \log N)$ greedy algorithm and a $O(N^4 \log N)$ optimization solution, as compare to the exponential time naive means, to minimize the differnec of colors in the density box. Here N denotes the number of connected components defined in the last section, and it should be of some factor smaller than the number of vertices.

Suppose we have m color density windows in the coloring bounding box, and n sets of connected components formed in the first step of building the coloring graph. We consider them from a vectorial standpoint to form n numbers of m dimensional vectors. Denote the i^{th} component of each vector its value of off-color, say the number of a color-A vertices substract the ones being color-B, in the i^{th} windows. To minimize the the difference between color-A density and color-B density as much as possible for all color density windows is equivalent to minimize the modulus of total sum of n vectors. In other words, to minimize the sum of *absolute* value of the difference between colors, we minimize the sum of *square* difference between colors.

Without loss of generality, consider a color boundary box; within it are 4 color density windows and 4 connected components. Denote the 4 sets of 4 dimensional vectors (a_1, a_2, a_3, a_4) , (b_1, b_2, b_3, b_4) , (c_1, c_2, c_3, c_4) and (d_1, d_2, d_3, d_4) . The component a_1 denotes the difference of colors of a connected component in the first color density window. Now the only *operation* we can make is mutipltng a vector a scalar value of -1 , which means to reverse the colors of vertices withn the particular connected components. We will choose to make or not make a operation when adding two vector by comparing their *inner product*, i.e., two vectors seperated by a bigger degree of angle will have smaller value of inner product, and the additive modulus will tend to cancel out.

We can perform numerical calculation to give a optimized solutions. Given 4 variables $\alpha, \beta, \gamma, \sigma$ subject to constraints

$$\alpha^2 = 1, \beta^2 = 1, \gamma^2 = 1, \sigma^2 = 1$$

which is a means of quantifying the operation, and a function

$$f(\alpha, \beta, \gamma, \sigma) = (\alpha \mathbf{a} + \beta \mathbf{b} + \gamma \mathbf{c} + \sigma \mathbf{d})^2$$

we can minimize the function through Lagrange multiplier

$$\delta f(\alpha, \beta, \gamma, \sigma) - \delta \sum_{\alpha, \beta, \gamma, \sigma} \lambda_k (k^2 - 1) = 0$$

The partial differential equation leads to a dense nonlinear equation of which the null space is to be solved.

In our implementation we use the *greedy algorithm* instead so as to give a reasonable efficiency. The greedy choice is the largest modulus of all vectors. The only expensive operation is to firstly *sort* the connected components in a modulus-descending manner, and then loop over in linear time. In each iteration we compute the inner product of two vector, and reverse the sign to check whether it gives the larger degree of angle.

Complexity Analysis

How fast is the method of Lagrange multiplier? Typical linear equation solver, such as *xGESV* of *LAPACK*, has complexity $O(N^3)$. The *Newton-Raphson* method is used to tackle the nonlinearity and has complexity of typical $O(\log N)$. With extra $O(N)$ of inner products (of supposedly sparse vector) to be performed, the total complexity to solve the above equation is $O(N^4 \log N)$.

How fast is the vectorial greedy algorithm? First we compute the modulus of every l dimensional vectors. We have N vectors so the operation takes $O(lN)$. l can also be viewed as the number of color density windows. The time to sort the vectors according to their modulus is $O(l \lg N)$. The time to iterate through the vector and compare the larger degree of angle of any two vector is $O(lN)$. The total complexity is therefore $O(N \lg N + lN)$. In the case that the number of color density windows is much smaller than N , the time complexity will have $O(N \lg N)$.

Noted that N , the number of connected components, is not equal to V , the number of vertices (shape). We should have $N < V$ in all cases.

5 Experiment Result

We have tested 5 cases provided by the official. We retrieving system time with *gettimeofday()* which has a resolution of microseconds.

```
System: Intel(R) Xeon(R) CPU 2.53GHz
OS: Linux Debian
Compiler: gcc version 4.9.2 (Debian 4.9.2-16)
```

```
case 1, 17 shapes, total time: 786 us
case 2, 56 shapes, total time: 2329 us
case 3, 424 shapes, total time: 48648 us
case 4, 30 shapes, total time: 903 us
case 5, 559 shapes, total time: 80992 us
```

6 Conclusion

The efficiency of the whole algorithm is obviously bounded by the time complexity of building the coloring graph, which is $O(V^2)$ in general. To give a optimization solution to the balancing problem imposed a greater challenge also. We have come up with the one utilizing the method of Lagrange multiplier, complexity being $O(N^4 \log N)$ which is still not resonable for the real implementation. Nevertheless, the greedy algorithm if sufficient to give a nice

optimization.

7 Work Distribution

The total process can be divided into three category : BUILDING COLORING GRAPH (section 2); COLORING (section 3); CALCULATING COLOR DENSITY (section 4).

湯智帆	algorithm and programming BUILDING COLORING GRAPH
徐吉吉	algorithm and programming COLORING, programming CALCULATING COLOR DENSITY
鍾瑞輝	algorithm CALCULATING COLOR DENSITY and the report

8 Reference

http://cad-contest.el.cycu.edu.tw/problem_E/default.htm

台大電機演算法第一組
2015/6/30 補充
有關於計分以及檢察著色正確性
B00404017 湯智帆

```
color A: -5  
color B: -5  
  
color A: 1  
color B: 0  
  
color A: 1  
color B: 0  
  
color A: 1  
color B: 0  
  
CORRECT COLORING!!!
```

(ps: color 為-5 表示無法著色的 edge!)

我們在每一筆資料都有進行 coloring check 就是檢查相鄰 Edge 兩端的 vertex 應該要有不同色，而 iccad 提供的測資五筆經過我們的著色演算法結果都是 Correct coloring。因此根據官方網頁提供的公式

$$f(x) = \begin{cases} 20, & \text{if coloring graphs for test case } x \text{ are built correctly} \\ 0, & \text{others} \end{cases}$$

$$g(x) = \begin{cases} 10, & \text{if color density windows for test case } x \text{ are shown correctly} \\ 0, & \text{others} \end{cases}$$

$$h(x) = \begin{cases} \sum_{d=1}^{k(x)} \left| \frac{70}{k(x)} - \frac{|color_A_density_d - color_B_density_d|}{5} \right|, & \text{if coloring for test case } x \text{ is correct} \\ 0, & \text{others} \end{cases}$$

可以開始著手計算得分

第一筆測資得分：

```
the score of test casecase1 is:  
95.1354
```

第二筆測資得分：

```
the score of test casecase2 is:  
90.5821
```

第三筆測資得分：

```
the score of test casecase3 is:
-45.9988
```

第四筆測資得分：

```
the score of test casecase4 is:
89.4279
```

第五筆測資得分：

```
the score of test casecase5 is:
87.8728
```

而根據上述計算平均公式：

$$final_score = \frac{1}{n} \sum_{x=1}^n score(x)$$

可以進行平均分數計算

$$\frac{95.1354 + 90.5821 + (-45.9988) + 89.4279 + 87.8728}{5} = 63.40$$

以上是我們針對結果的分數計算部分。

很明顯，我們可以發現有一個 outlier，就是 case3，因為 case3 都是單獨一個一個 vertex 的，因此我們的著色演算法沒有考慮到這點，因此造成分數較低，這是我們之後需要思考與改進演算法的部分。

✧ 補充：有關於計分的澄清，因為我們之後才發現其實他有做一個修正，把原本的有外面絕對值得改成沒有絕對值，因此才會有負數的情形

$$h(x) = \begin{cases} \sum_{i=1}^{n(x)} \left(\frac{70}{n(x)} - \frac{|densityAi - densityBi|}{5} \right), & \text{if coloring is correct in case } x \text{ and its value } \geq 0 \\ 0, & \text{others} \end{cases}$$

分工情形

湯智帆

- a. main.cpp 檔案讀取部分以及 main.cpp 後面 color checking 部分
- b. getpart.cpp getpart.h
- c. graph.cpp graph.h(主要由助教作業提供，有些部分如 connected component(cc)由湯智帆修改)
- d. Progress presentation
- e. 後續計分程式，如把 density 從 output 取出的 parse.cpp 以及最終計算成績的 score.cpp

徐喆

- a. main.cpp 著色實作，包含 BFS 確定是否 colorable 以及向量著色演算法的實作以及 coloring density 計算
- b. graph.cpp 內的 sortCCByVecLen、CCCompByVecLen 函數
- c. 後續計分結果討論

鍾瑞輝

- a. 提出向量著色演算法以及 coloring density 計算演算法
- b. 期末報告撰寫
- c. 複雜度分析
- d. 軟體 run time 測試

結語：

因為我們這組只有三個人(有一位中途停修)，因此人力分配也較為吃緊，雖然結果差強人意，不過我們仍在這次專題中學習到很多，也有了分工合作的經驗。