PDA Programming Assignment 2

Chip Floorplanning – 2023 spring

Outline

- Problem Description
- Algorithm with Simulate Annealing
  - Slicing tree Floorplan
  - B*-Tree Floorplan

Outline

- Problem Description
- Algorithm with Simulate Annealing
  - Slicing tree Floorplan
  - B*-Tree Floorplan

Problem Description

$B = \{b_1, b_2, b_3, \ldots, b_k\}$ be a set of k rectangular hard IP blocks.

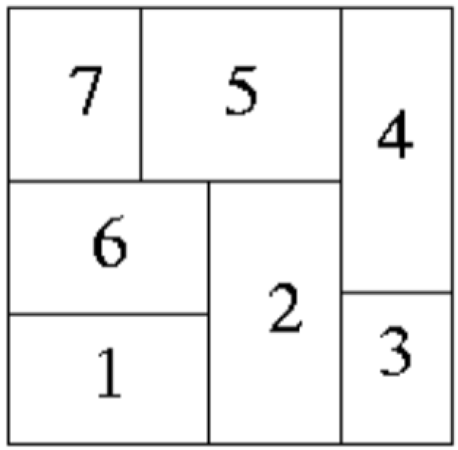
Target: Placing all blocks within a rectangular chip without any overlaps such that the area of chip bounding box, $\boldsymbol{A}$.

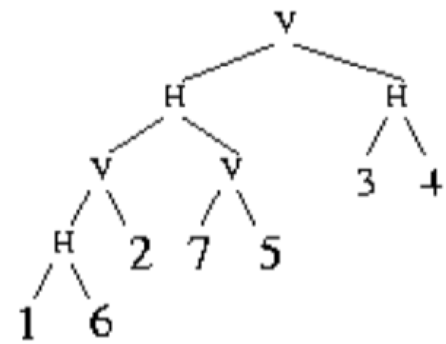
Input Format	Sample Input
$< R_{lowerbound} >< R_{upperbound} >$ $< Block\ name >< Block\ width >< Block\ height >$ ...	0.5 2.0 $b1$ 40 50 $b2$ 60 50 $b3$ 60 50 $b4$ 40 50

Outline

- Problem Description
- Algorithm with Simulate Annealing
  - Slicing tree Floorplan
  - B*-Tree Floorplan

Slicing Tree





$E = 16H2V75VH34HV$

$E = 16{+}2{*}75{*}{+}34{+}{*}$

Postorder traversal of a tree!

However, different slicing tree may map to same slicing floorplan...

Slicing Tree(Cont.)

We can solve above problem by applying

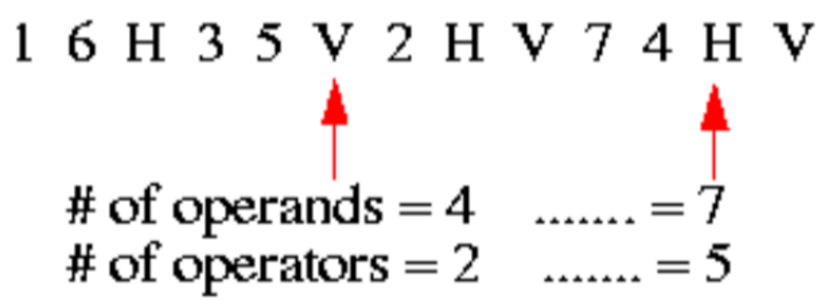
- **Balloting property**
- **Skewed property**

Polished Expression

An expression $\boldsymbol{E} = e_1 e_2 e_3 \ldots e_{2n-1}$ is called ***Polished expression*** iff

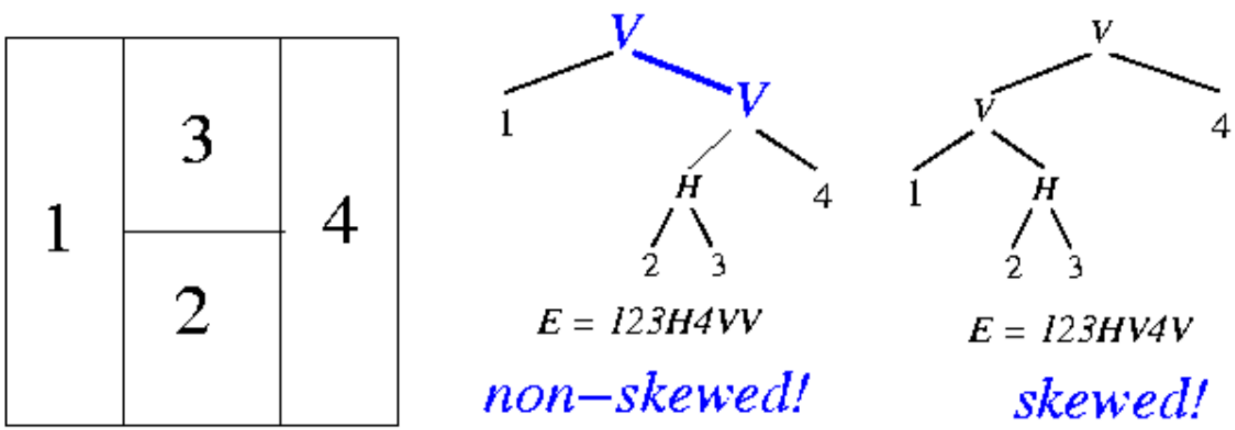
- Every operand appears exactly once in $\boldsymbol{E}$
- (***Balloting property***) For every subsequence of $\boldsymbol{E}$,

$$\boldsymbol{\#\ of\ operands\ >\ \#\ of\ operators}$$



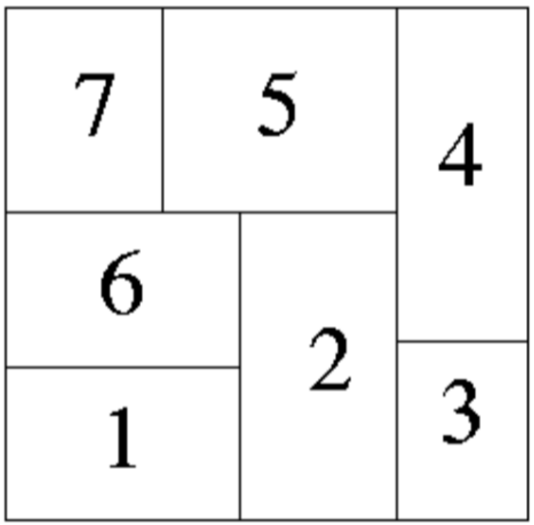
Normalized Polished Expression

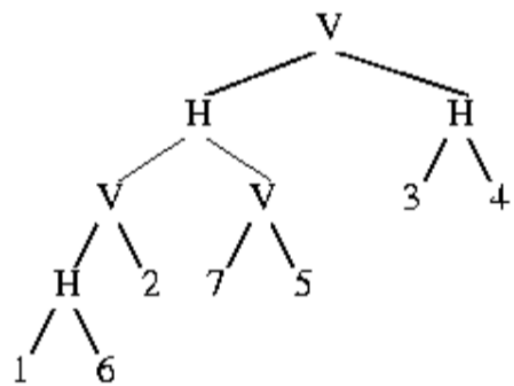
A polish expression ***E*** is called ***Normalized*** iff ***E*** has no consecutive operators of the same type.



Normalized Polished Expression(Cont.)

Once we have an ***Normalized Polished Expression***, we can construct an ***unique*** rectangular slicing floorplan.





$E = 16H2V75VH34HV$

A normalized Polish expression

Data Structure

```
class Block{
public:
    Block(){};
    Block(string a):type(a) {
        x = 0;
        y = 0;
        w = 0;
        h = 0;
        leftChild = nullptr;
        rightChild = nullptr;
    };
    Block(string a, list<pair<int,int>> b):type(a),whs(b) {
        x = 0;
        y = 0;
        w = 0;
        h = 0;
        leftChild = nullptr;
        rightChild = nullptr;
    };

    string type;
    int x, y;
    int w, h;
    list<pair<int,int>> whs;
// all possible width and height
    Block* parent;
    Block *leftChild, *rightChild;
};
```

Algorithm - Slicing Tree

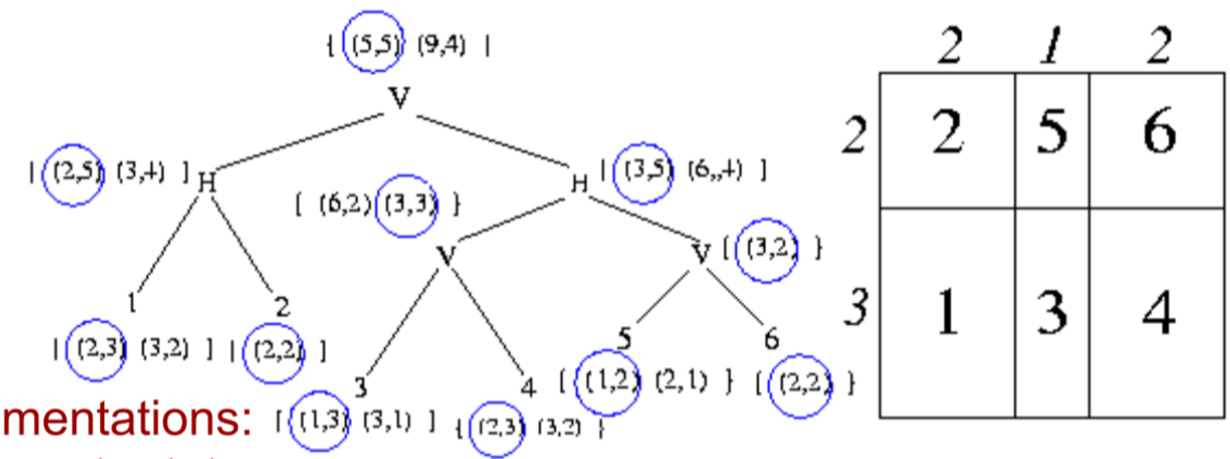
Suppose we have an NPE, we reconstruct it into a slicing floorplan by applying ***post-order traversal*** to the slicing tree (NPE).

```
stack<Block*> bs;
int len = npe.size();
for(int i = 0; i < len; i++){
    if(npe[i]->type != "V" && npe[i]->type != "H"){
        bs.push(npe[i]);
    }
    else{
        Block* operand2 = bs.top();
        bs.pop();
        Block* operand1 = bs.top();
        bs.pop();
        stackBlock(operand1,operand2,npe[i]);
        bs.push(npe[i]);
    }
}
```

Algorithm - Slicing Tree(Cont.)

During the traversal, we also need to record all possible combinations of the floorplan and finally select the best(smallest) one.

- Allow rotation: each block has up to two implementations.



2 implementations: rotate or not rotate

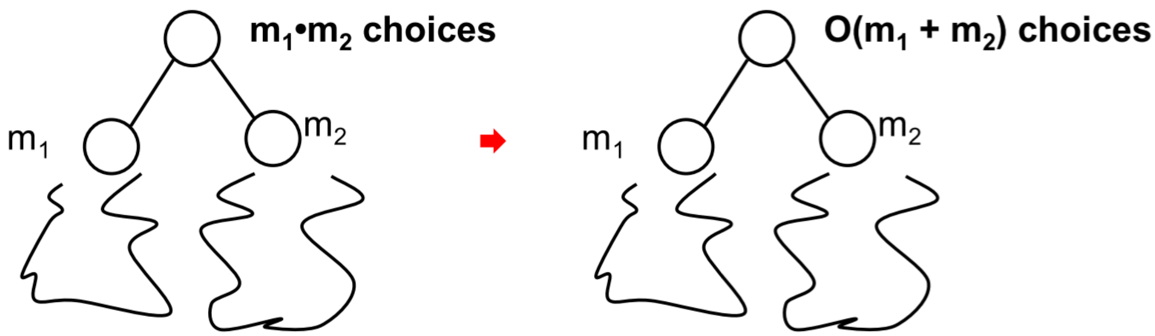
Algorithm - Slicing Tree(Cont.)

However, try all possible combinations results in $O(nm)$ complexity for each level of the tree, where n and m is the number of possible combinations of left child and right child, respectively.

How to improve it?

Algorithm - Slicing Tree(Cont.)

By applying Stockmeyer Algorithm, we can get tremendous improvement on time complexity from $O(mn)$ to $O(m + n)$.



Algorithm - Stockmeyer

If current cut is vertical:

- Sort ***whs*** of left child and right child in ***increasing width, decreasing height.***

If current cut is horizontal:

- Sort ***whs*** of left child and right child in ***decreasing width, increasing height.***

Algorithm – Stockmeyer(Cont.)

visit node a: Since the cut orientation is vertical;

$$L = \{(2,3),(3,2)\}$$
$$R = \{(2,4),(4,2)\}$$

i join $l_1 = (2,3)$ and $r_1 = (2,4)$: we get $(2+2, \max\{3,4\}) = (4,4)$. Since the maximum is from R, we join l_1 and r_2 next.

ii join $l_1 = (2,3)$ and $r_2 = (4,2)$: we get $(2+4, \max\{3,2\}) = (6,3)$. Since the maximum is from L, we join l_2 and r_2 next.

iii join $l_2 = (3,2)$ and $r_2 = (4,2)$: we get $(3+4, \max\{2,2\}) = (7,2)$.

Thus, the resulting dimensions are $\{(4,4),(6,3),(7,2)\}$.

Algorithm – Stockmeyer(Cont.)

However, sorting takes time complexity $O(nlgn)$, we can avoid these sorts by ***placing data in required sequence at the beginning***. Therefore, we can guarantee that the sequence of ***whs*** after applying Stockmeyer algorithm will be ***increasing width, decreasing height***.

```
list<pair<int,int>> ST::findWH(pair<int,int> wh){
    list<pair<int,int>> width_height;
    // rotate the hard block and store its possible width-height pairs
    // store width_height pair in "increasing width, decreasing height"
    if(wh.first == wh.second) width_height.push_back(wh);
    else {
        if(wh.first > wh.second) swap(wh.first,wh.second);
        width_height.push_back(wh);
        width_height.push_back({wh.second,wh.first});
    }

    return width_height;
}
```

Algorithm – Stockmeyer(Cont.)

Once we need applying Stockmeyer on horizontal cut, just iterate ***whs*** from its end, then we get ***decreasing width, increasing height***.

```
if(op->type == "V"){
    auto it1 = a->whs.begin();
    auto it2 = b->whs.begin();
    while(it1 != a->whs.end() && it2 != b->whs.end()){
        op->whs.push_back(verticalMerge((*it1),(*it2)));
        int h_1 = (*it1).second;
        int h_2 = (*it2).second;
        if(h_1 > h_2) it1++;
        else if(h_2 > h_1) it2++;
        else { // h_1 == h_2
            it1++;
            it2++;
        }
    }
}
```

```
else{ // op->type == "H"
    auto it1 = a->whs.rbegin();
    auto it2 = b->whs.rbegin();
    while(it1 != a->whs.rend() && it2 != b->whs.rend()){
        op->whs.push_front(horizontalMerge((*it1),(*it2)));
        int w_1 = (*it1).first;
        int w_2 = (*it2).first;
        if(w_1 > w_2) it1++;
        else if(w_2 > w_1) it2++;
        else { // w_1 == w_2
            it1++;
            it2++;
        }
    }
}
```

Algorithm - Slicing Tree(Cont.)

By applying above steps, we can construct a slicing floorplan from an NPE.

Next, we apply Simulate Annealing (SA) to get different NPE and compare their cost, where the cost function is:

$$curCost = \alpha * \frac{area}{totalArea} + (1 - \alpha) * penalty$$

```
double penalty = (R >= R_lowerBound && R <= R_upperBound)? 0.0:1.0;
```

Algorithm – Simulate Annealing

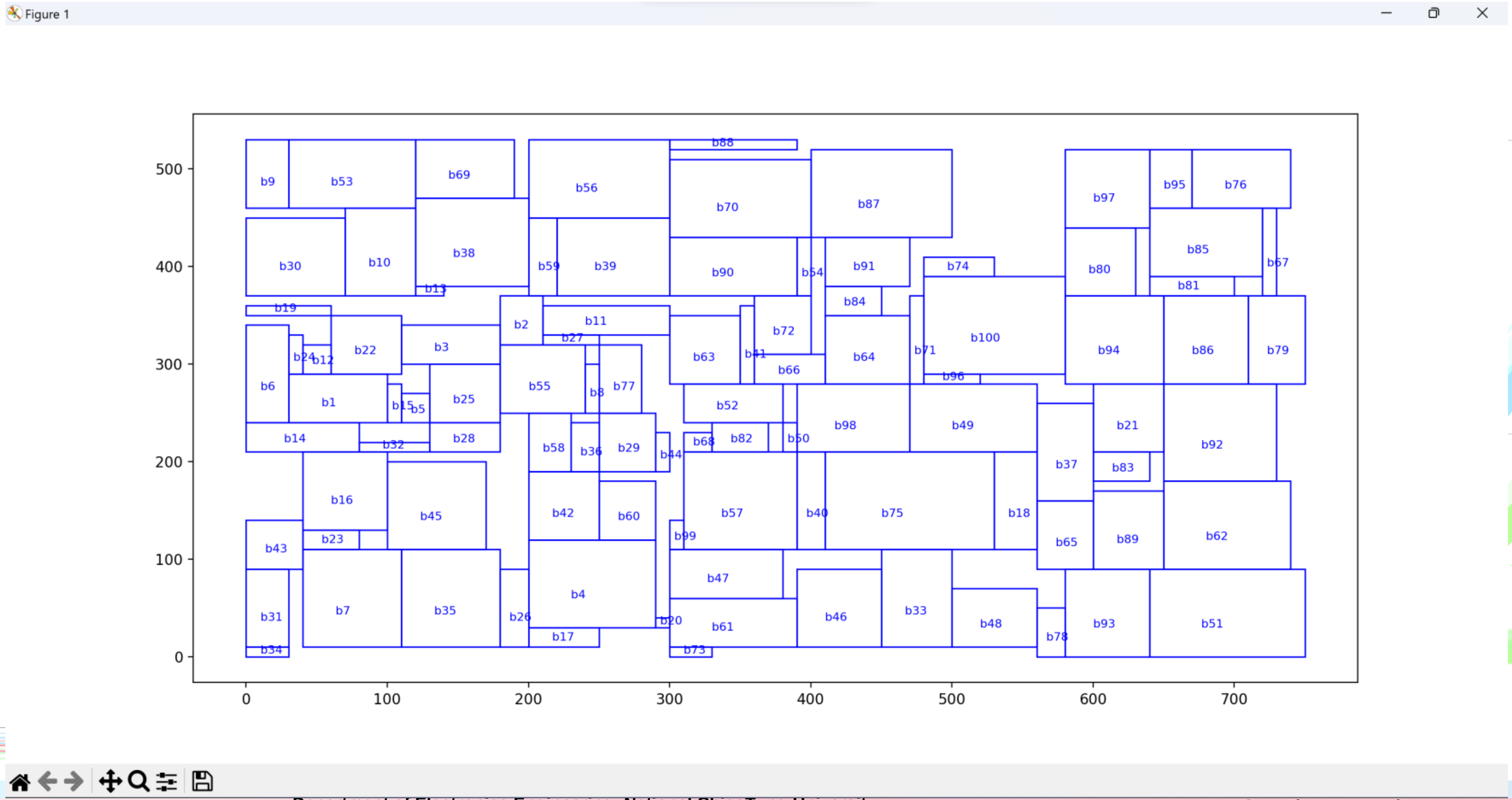
3 types of moves:

- *M1* (**Operand Swap**): Swap two adjacent operands.
- *M2* (**Chain Invert**): Complement some chain ($V = H$, $H = V$).
- *M3* (**Operator/Operand Swap**): Swap two adjacent operand and operator.

Only M3 may violate the balloting or skewed property!

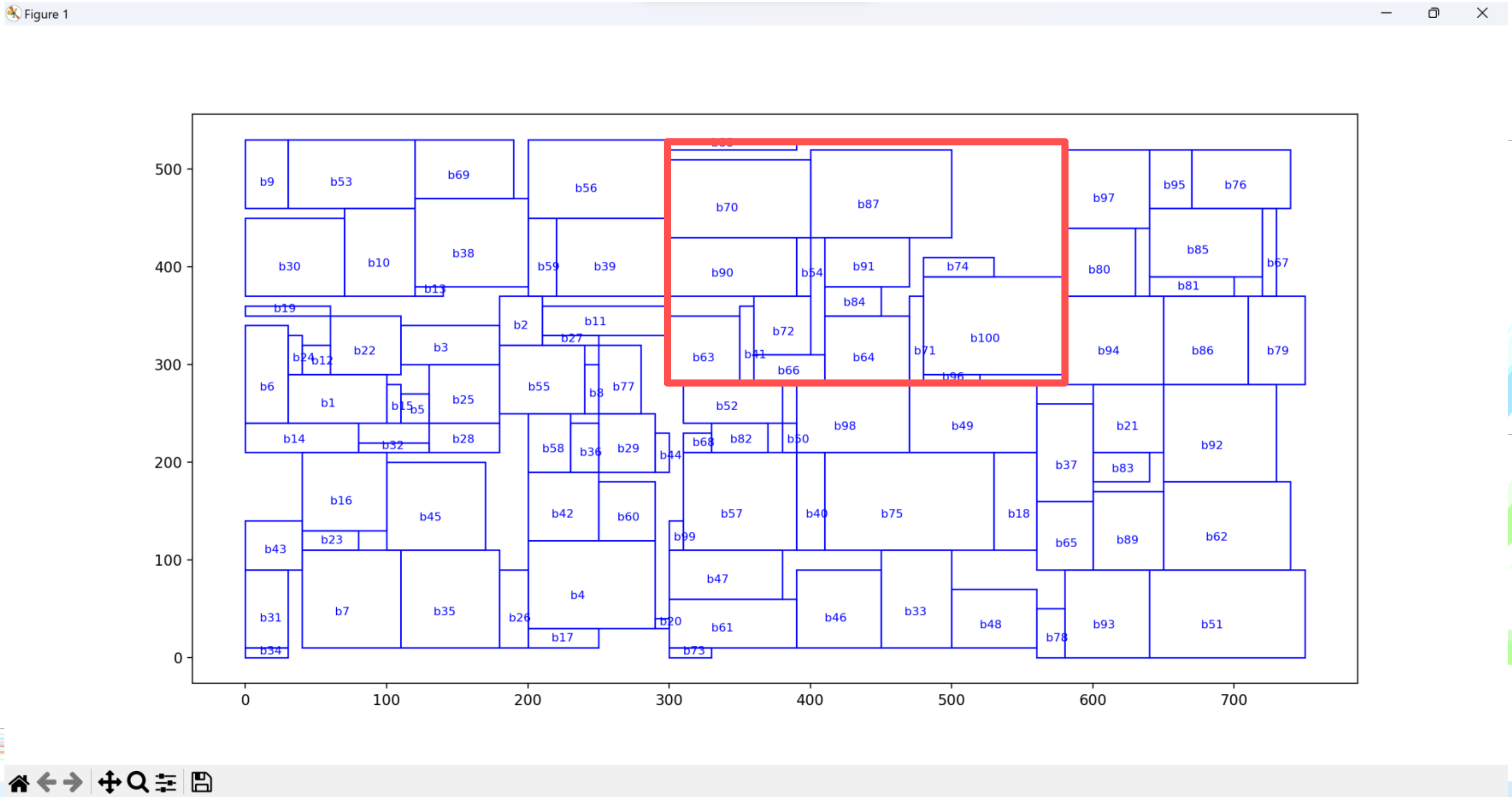
Final Floorplan(100 macros)

Takes time 4.002 sec, total area = 430700



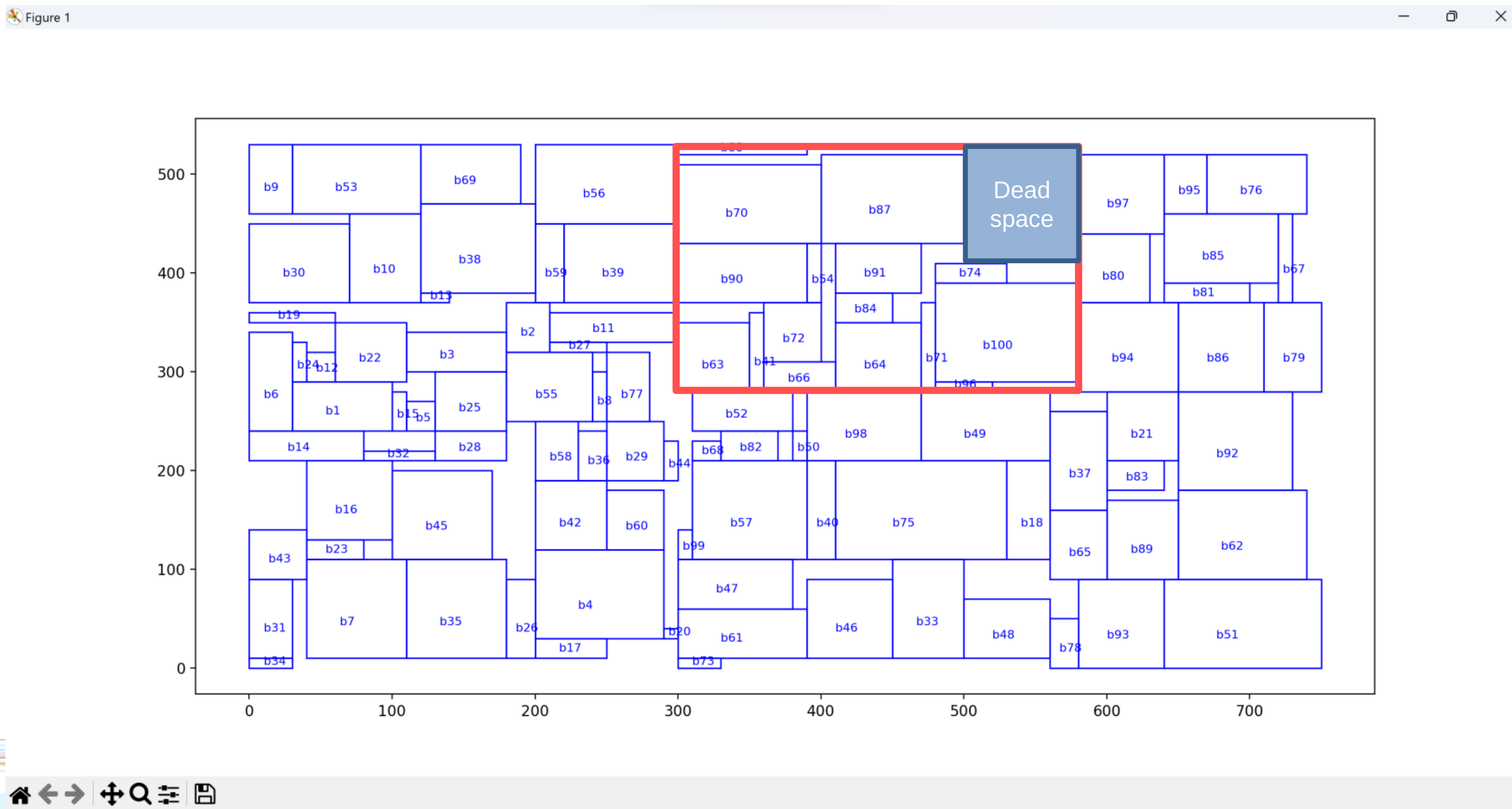
Final Floorplan(100 macros)

However, the property of "slicing" limits the solution space…



Final Floorplan(100 macros)

However, the property of "slicing" limits the solution space…



Outline

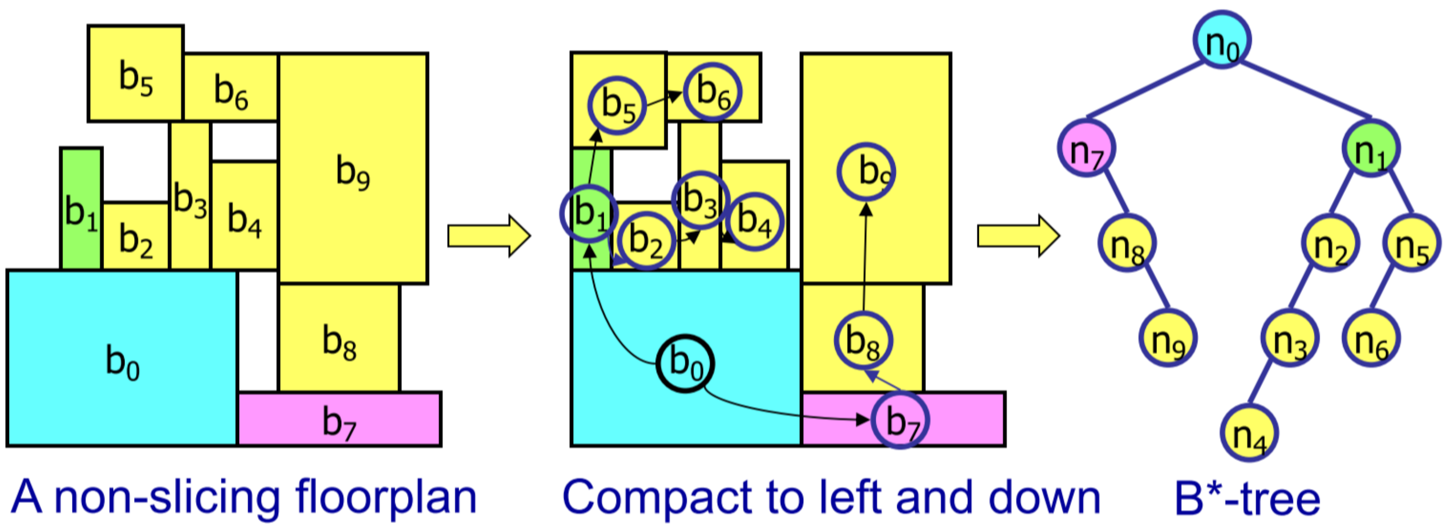
- Problem Description
- Algorithm with Simulate Annealing
  - Slicing tree Floorplan
  - B*-Tree Floorplan

Algorithm - B* Tree

- Compact modules to ***bottom-left***
- Construct an ***Ordered Binary Tree*** (B*-tree)

  Left Child: the lowest, adjacent block on the right

  Right Child: the first block above the current block



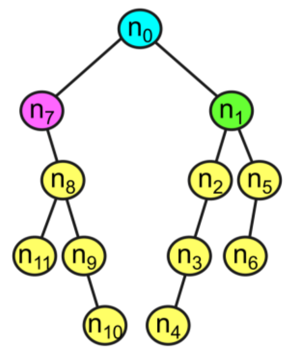
Algorithm – B*-Tree(Cont.)

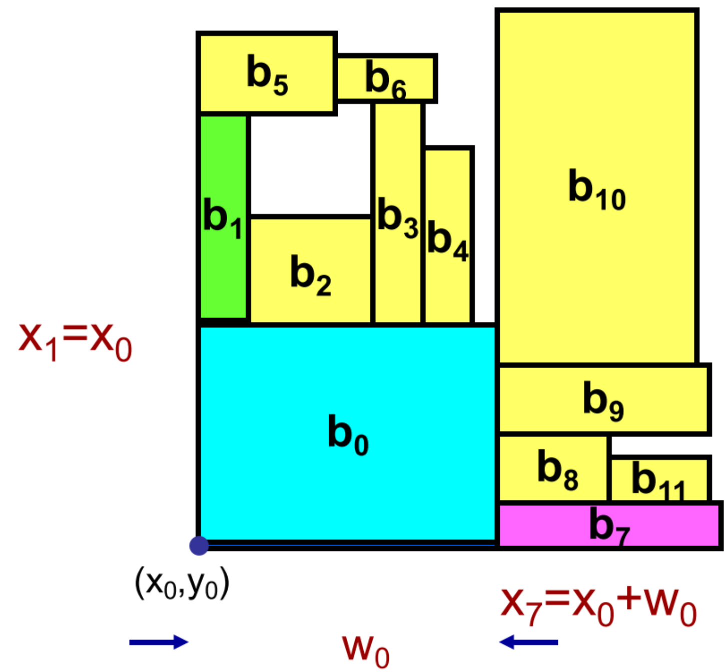
Traversing the tree by using ***preorder-traversal***, we can calculate x coordinates efficiently.

$\rightarrow x_{leftChild} = x_{curBlock} + w_{curBlock}$

$\rightarrow x_{rightChild} = x_{curBlock}$

How about y-coordinates?





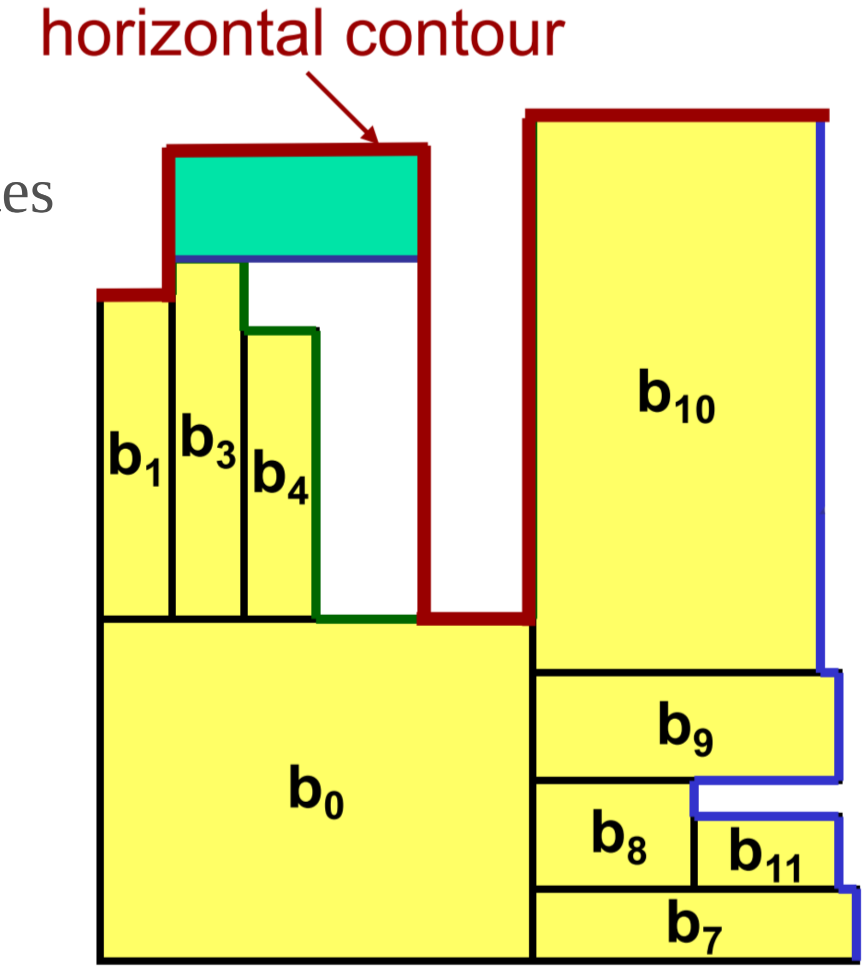
Horizontal Contour

Using ***Horizontal Contour***

→construct a vector, whose index indicates the x-coordinate, and its value is correspond to top y-value.

Ex:

For arr[50] = 3, it indicates that the top boundary of $x = 50$ is $y = 3$.



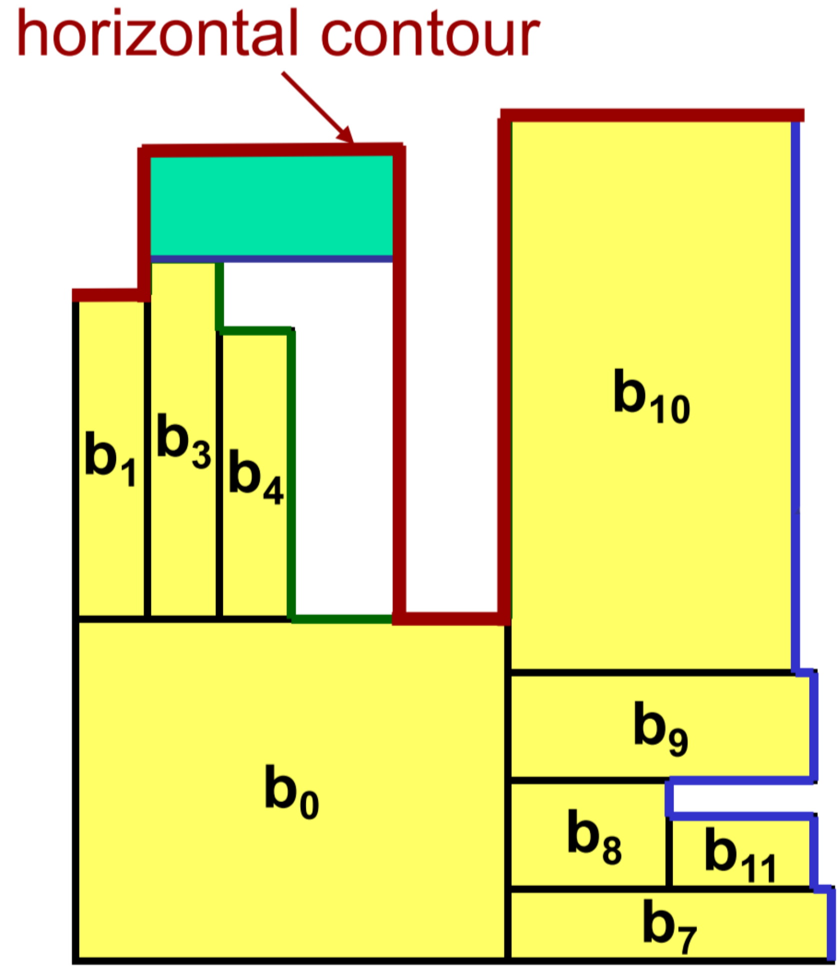
Horizontal Contour

By using such contour, we can decide what height should next block be placed into our floorplan.

Ex:

To place a block start from $x = 5$, and its width is 3, we can decide its y-coordinate by following equation:

$$y = \max(x[5], x[6], x[7])$$



Horizontal Contour (Cont.)

However, when the width of block is quite large, using vector to be the data structure of horizontal contour is time-consuming:

- Placing a block → $O(w_{block})$ on traversing vector
- Its space complexity is also quite large→ $O(w_{total})$.

Using ***LinkedList*** to solve above problem!

Horizontal Contour (Cont.)

Data Structure→std::list<ContourElement>

```
class ContourElement{
public:
    ContourElement(): x_start(0),
                      x_end(0),
                      y(0) {};
    ContourElement(int start, int end, int new_y): x_start(start),
                                                   x_end(end),
                                                   y(new_y) {};

    ~ContourElement(){};
    int x_start;
    int x_end;
    int y;
};
```

Ex: If a block is start from $x = 0$ to $x = 10$,

its $x\_start = 0$, $x\_end = 9$

Horizontal Contour (Cont.)

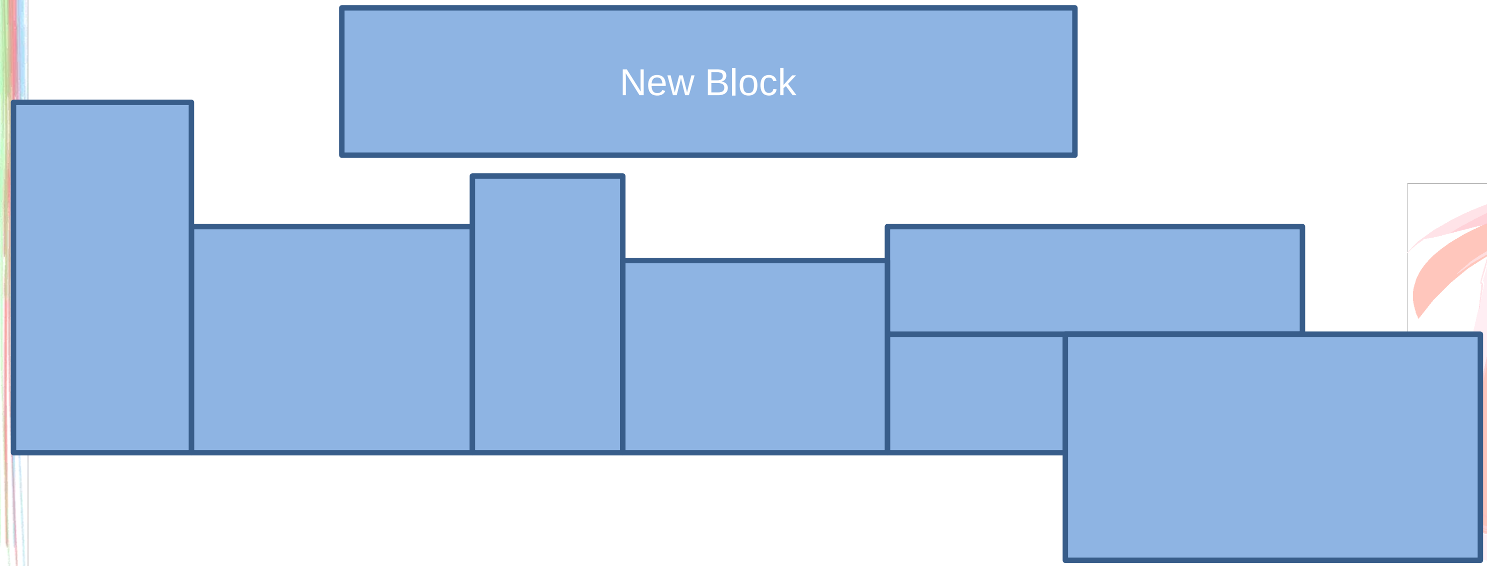
Roughly divide into 2 scenarios:

1. ***x_start*** and ***x_end*** cover several ***ContourElement***s
2. ***x_start*** and ***x_end*** are both inside a ***ContourElement***

Horizontal Contour (Cont.)

Roughly divide into 2 scenarios:

1. ***x_start*** and ***x_end*** cover several ***ContourElement***s
2. ***x_start*** and ***x_end*** are both inside a ***ContourElement***



Horizontal Contour (Cont.)

x_start and *x_end* cover several *ContourElement*s

- Replace the leftmost completely covered ***ContourElement*** into ***ContourElement*** with new ***x_start*** and ***x_end***.
- Remove all others completely covered ***ContourElement***s
- if $prev(\boldsymbol{ContourElement}_{new}).x\_end \geq \boldsymbol{ContourElement}_{new}.x\_start$

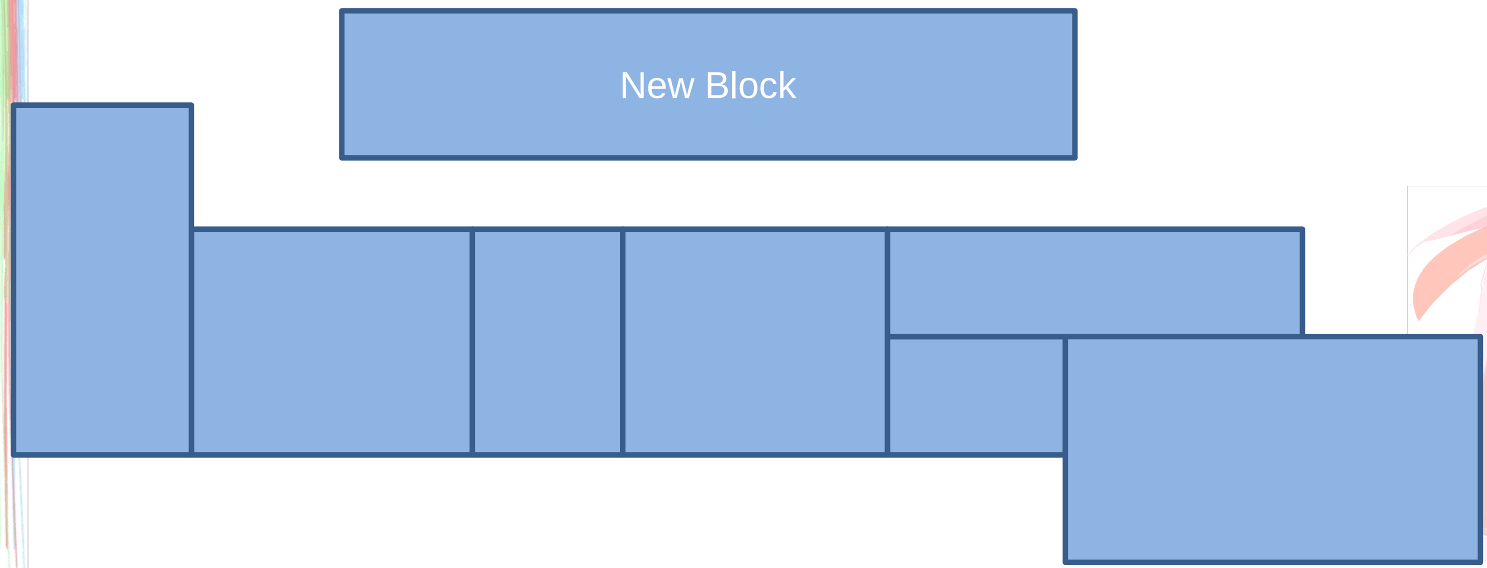
  $prev(\boldsymbol{ContourElement}_{new}).x\_end = \boldsymbol{ContourElement}_{new}.x\_start - 1$
- if $next(\boldsymbol{ContourElement}_{new}).x\_start \leq \boldsymbol{ContourElement}_{new}.x\_end$

  $next(\boldsymbol{ContourElement}_{new}).\ x\_start = \boldsymbol{ContourElement}_{new}.x\_end + 1$

Horizontal Contour (Cont.)

Roughly divide into 2 scenarios:

1. ***x_start*** and ***x_end*** cover several ***ContourElements***
2. ***x_start*** and ***x_end*** are both inside a ***ContourElement***



Horizontal Contour (Cont.)

x_start **and** ***x_end*** **are both inside a** ***ContourElement***

- $x\_start_{new} = x\_start_{old}$ && $x\_end_{new} = x\_end_{old}$
  →directly update *ContourElement.y*
- Only $x\_start_{new} = x\_start_{old}$
- Only $x\_end_{new} = x\_end_{old}$
- $x\_start_{new} > x\_start_{old}$ && $x\_end_{new} < x\_end_{old}$

Horizontal Contour (Cont.)

x_start and ***x_end*** are both inside a ***ContourElement***

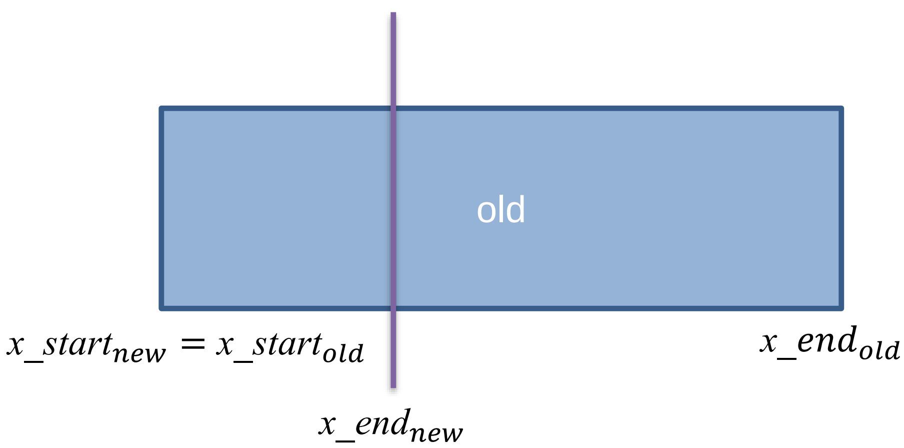
- Only $x\_start_{new} = x\_start_{old}$

```
else if(start == (*iter1).x_start){
            ContourElement ce(start,end-1,new_y);
            contour.insert(iter1,ce);
            (*iter1).x_start = end;
        }
```

Horizontal Contour (Cont.)

x_start **and** ***x_end*** **are both inside a** ***ContourElement***

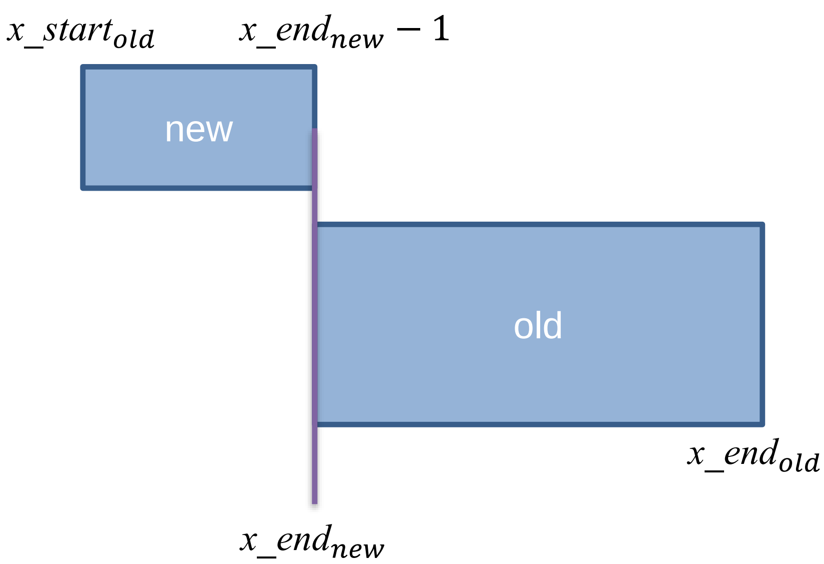
- Only $x\_start_{new} = x\_start_{old}$



Horizontal Contour (Cont.)

x_start **and** ***x_end*** **are both inside a** ***ContourElement***

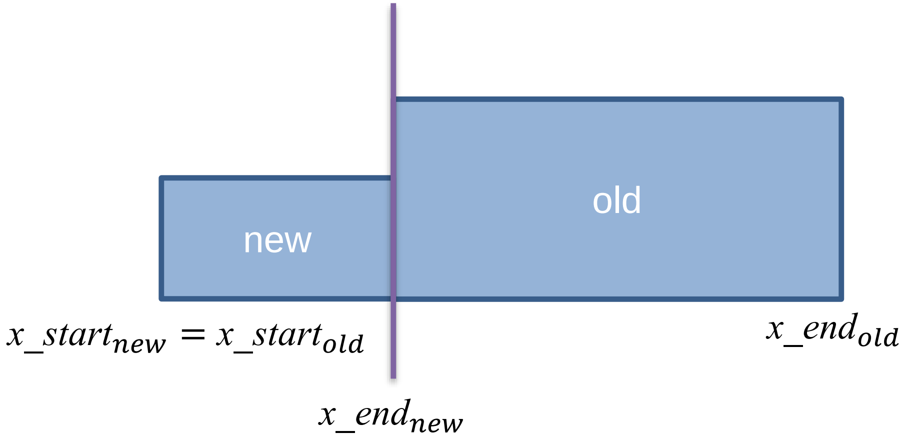
- Only $x\_start_{new} = x\_start_{old}$



Horizontal Contour (Cont.)

x_start and x_end are both inside a ***ContourElement***

- Only $x\_start_{new} = x\_start_{old}$



Horizontal Contour (Cont.)

x_start and *x_end* are both inside a ***ContourElement***

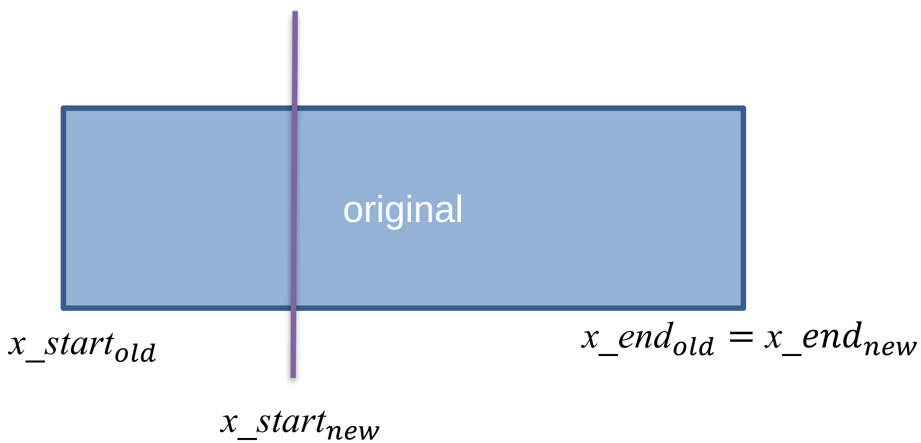
- Only $x\_end_{new} = x\_end_{old}$

```
else if(end-1 == (*iter1).x_end){
            ContourElement ce((*iter1).x_start,start-1,(*iter1).y);
            contour.insert(iter1,ce);
            (*iter1).x_start = start;
        }
```

Horizontal Contour (Cont.)

x_start and ***x_end*** are both inside a ***ContourElement***

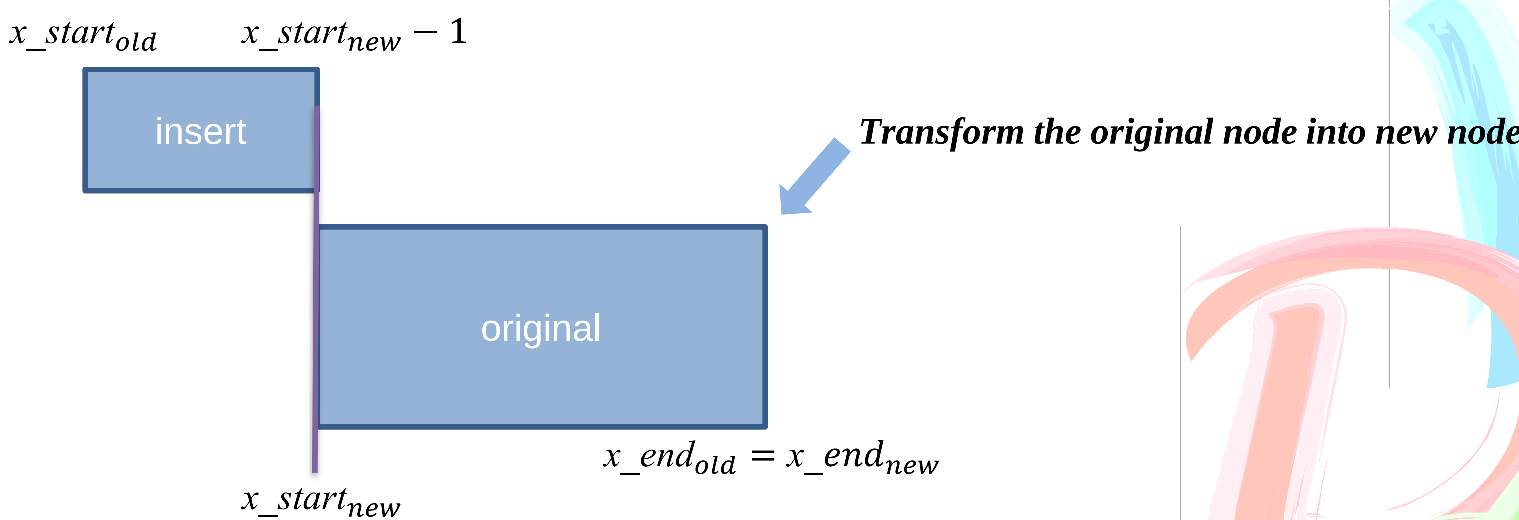
- Only $x\_end_{new} = x\_end_{old}$



Horizontal Contour (Cont.)

x_start and *x_end* are both inside a ***ContourElement***

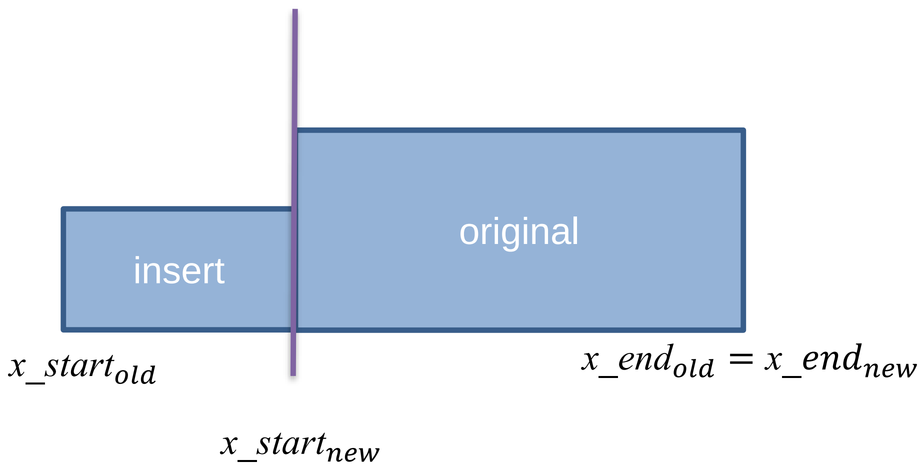
- Only $x\_end_{new} = x\_end_{old}$



Horizontal Contour (Cont.)

x_start **and** ***x_end*** **are both inside a** ***ContourElement***

- Only $x\_end_{new} = x\_end_{old}$



Horizontal Contour (Cont.)

x_start and ***x_end*** are both inside a ***ContourElement***

- $x\_start_{new} > x\_start_{old}$ && $x\_end_{new} < x\_end_{old}$

```
else{
            ContourElement ce1((*iter1).x_start,start-1,(*iter1).y);
            contour.insert(iter1,ce1);
            ContourElement ce2(start,end-1,new_y);
            contour.insert(iter1,ce2);
            (*iter1).x_start = end;
        }
```

Horizontal Contour (Cont.)

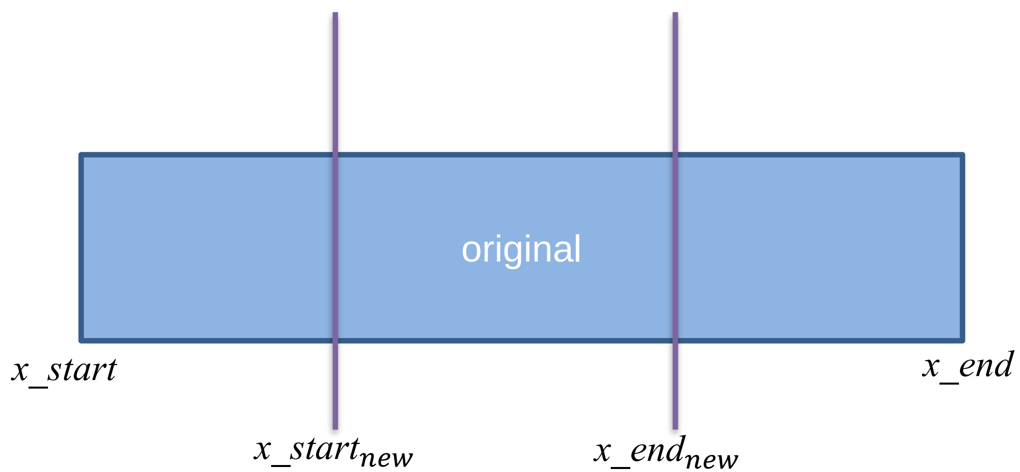
x_start and ***x_end*** are both inside a ***ContourElement***

- $x\_start_{new} > x\_start_{old}$ && $x\_end_{new} < x\_end_{old}$

Horizontal Contour (Cont.)

x_start and ***x_end*** are both inside a ***ContourElement***

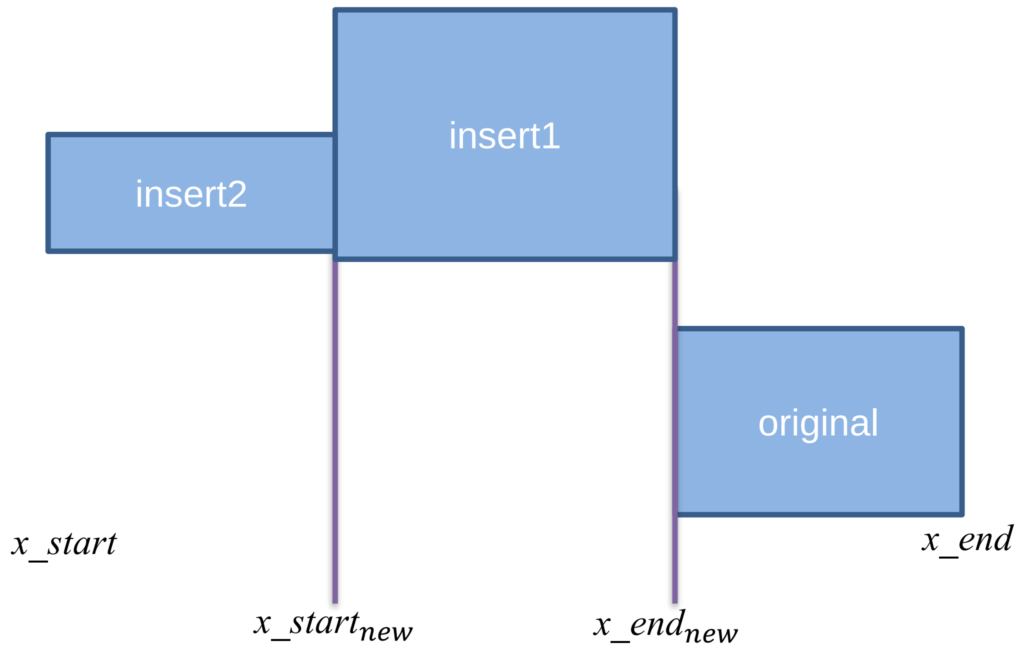
- $x\_start_{new} > x\_start_{old}$ && $x\_end_{new} < x\_end_{old}$



Horizontal Contour (Cont.)

x_start and ***x_end*** are both inside a ***ContourElement***

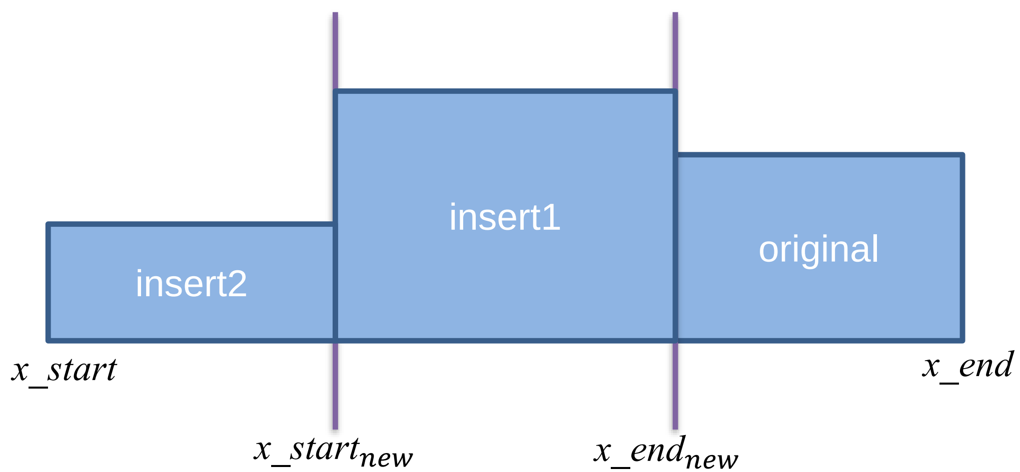
- $x\_start_{new} > x\_start_{old}$ && $x\_end_{new} < x\_end_{old}$



Horizontal Contour (Cont.)

x_start and ***x_end*** are both inside a ***ContourElement***

- $x\_start_{new} > x\_start_{old}$ && $x\_end_{new} < x\_end_{old}$



Horizontal Contour (Cont.)

By applying LinkedList Horizontal Contour, we only need to spend time on finding the target ***ContourElement*** (at most $O(n)$, where n is the length of LinkedList), and all remaining steps take $O(1)$.

It's a good way to improve efficiency when every w_{block} is large.

Algorithm – B*-Tree(Cont.)

By applying above steps, we can construct a floorplan from a B*-tree.

Next, we apply Simulate Annealing (SA) to get different B*-tree and compare their cost, where the cost function is:

$$curCost = area$$

```
for(Block const &block:blocks){
    width = max(width, block.x + block.w);
    height = max(height, block.y + block.h);
}

double area = width * height;
```

Algorithm – Simulate Annealing

3 type of moves:

- ***Rotate** the block*
- ***Swap** 2 node*
- ***Remove** and **insert** the node*

Final Floorplan(100 macros)

Takes time 0.483 s, total area = 359600