PD PA2 Report

R13943144 吳紹宇

- **Method Concept:**

I use a B*-tree representation combined with a two-stage simulated annealing approach to place the blocks within the fixed outline while optimizing the area and half-perimeter wirelength (HPWL).

The B*-tree supports three operations: swapping two nodes, rotating a block, and deleting and reinserting a node.

For simulated annealing, I apply different cost functions at different stages:

1. In the first stage, the cost function consists of three terms: area, HPWL, and exceed area (the area extending beyond the fixed outline). This stage primarily focuses on ensuring that all blocks are placed within the fixed outline.
2. In the second stage, the cost function considers only area and HPWL. In this phase, I slightly increase the temperature and adjust the optimization ratio according to user preferences, aiming to achieve a better trade-off between area and wirelength.

- **Data structure:**

There are five classes in my code: **Block, Net, SegNode, Contour**, and **Floorplanner**. The details of each class will be described below.

Block represents either a terminal or a hard macro. It stores both the current and best (x, y) coordinates of the block, along with three pointers to other blocks. Each Block is treated as a node in the B*-tree structure.

```cpp
class Block
    private:
        string          _name;      // module name
        size_t           _w;         // width of the block
        size_t           _h;         // height of the block
        size_t          _x;        // min x coordinate of the block
        size_t          _y;        // min y coordinate of the block
        bool          _isTerminal; // true if the block is a terminal

        Block * _parent; // parent block (block not terminal)
        Block * _top;
        Block * _right;

        size_t _bestCoordinate[4]; // best coordinate for SA,
                                   // 0: x0, 1: y0, 2: x1, 3: y1

        static size_t   _maxX;      // maximum x coordinate for all blocks
        static size_t   _maxY;      // maximum y coordinate for all blocks
};
```

Net stores a list of blocks and terminals that belong to the same net.

```cpp
class Net
{
public:
    // constructor and destructor
    Net()   { }
    ~Net()  { }

    // basic access methods
    const vector<Block*> getTermList()   { return _blockList; }
    const size_t getSize() { return _blockList.size(); }

    // modify methods
    void addTerm(Block* term) { _blockList.push_back(term); }

    // other member functions

    double calcHPWL() {···

private:
    vector<Block*>   _blockList;  // list of terminals the net is connected to
};
```

SegNode is a node in the linked list structure of the Contour. Each SegNode represents a horizontal segment, defined by its leftmost x-coordinate (x0), rightmost x-coordinate (x1), and a common y-coordinate (y).

```cpp
// segment
class segNode
{
    public:
        // constructor and destructor
        segNode(): _prev(nullptr), _next(nullptr), _x0(0), _x1(0), _y(0) { }
        segNode(size_t x0, size_t x1, size_t y): _prev(nullptr), _next(nullptr), _x0(x0), _x1(x1), _y(y) { }
        segNode(segNode* prev, segNode* next, size_t x0, size_t x1, size_t y): _prev(prev), _next(next), _x0(x0), _x1(x1), _y(y) { }
        void clear(){…
        ~segNode() { }
        friend class Contour;

    private:
        segNode* _prev;
        segNode* _next;
        size_t _x0;
        size_t _x1;
        size_t _y;
};
```

Contour is implemented as a doubly linked list of SegNode objects. During the B*-tree packing process, the getY member function is used to determine the appropriate y-coordinate where a block can be placed. After placing a block, the contour segments are updated accordingly to reflect the new top boundary.

```cpp
// doubly linked-list of segments
class Contour
{
    public:
        // constructor and destructor
        Contour(): _head(nullptr), _tail(nullptr), _last(nullptr), _trash(nullptr), _size(0) { };
        ~Contour();

        size_t getY(size_t x, size_t w, size_t h);
        void clear();
        void print();

    private:

        segNode* findSegNode(size_t x);
        void insert(size_t x0, size_t x1, size_t y, segNode* frontNode, segNode* backNode);
        void erase(segNode* node);
        void moveTrash(segNode* node);

        segNode* _head; // head of the linked list
        segNode* _tail; // tail of the linked list
        segNode* _last; // last visited segNode
        segNode* _trash;
        size_t _size; // size of the linked list
};
```

Floorplanner is the main floorplanning engine that performs layout optimization using a B*-tree representation and simulated annealing. It manages the parameters for simulated annealing, stores information about the blocks and nets, maintains the B*-tree data structure, and provides various functions for B*-tree operations, simulated annealing steps, and related tasks.

```cpp
class Floorplanner
    public:
        friend void printFloorplanResult(std::fstream& output, Floorplanner* fp, chrono::duration<double>& duration);
        Floorplanner(std::fstream& input_blk, std::fstream& input_net, double alpha);
        ~Floorplanner();
        void floorplan();

        void SA_basic();
        void putBlockInDamnBox();
        void finalTuning();
        void fastSA();

    private:
        // read file
        void readBlock(std::fstream& input_blk);
        void readNet(std::fstream& input_net);

        void initPlace();
        void initPlace_2();
        void initPerturbation();
        void calCost();
        void updateBestCost(); // update best code when normalized area and hpwl are changed
        void updatePrevCost(); // update prev code when normalized area and hpwl are changed
        void calFinalCost();
        double calAspectRatioTerm(size_t height, size_t width);
        size_t calExceedArea(size_t height, size_t width);
        void updateGamma(bool increase_anyway, double exponent, double increase_ratio, double &upper_bound, double &lower_bound);

        // B* tree
        void packing();
        void updateBestBSTree();
        void retrieveBestBSTree();
        // op1 random rotate
        // void randomRotate(Block *block);
        // op2 random delete and insert
        void deleteBlock(Block* block, vector<InsertInfo>& insertInfoVec, vector<Block*>& parentBlockVec);
        void insertBlock(Block* block, Block* hostBlock, int flag); // flag means the block is inserted to the right or top of hostBlock
        void reInsertBlock(Block* block, vector<InsertInfo>& insertInfoVec, vector<Block*>& parentBlockVec); // undo
        // op3 random swap
        void swapTwoBlocks(Block* block1, Block* block2);
```

```cpp
class Floorplanner
    private:
        // floorlan information
        size_t _numBlocks; // number of blocks
        size_t _numTerminals; // number of terminals
        size_t _numNets;    // number of nets
        size_t _outlineWidth; // outline width
        size_t _outlineHeight; // outline height
        double _givenAlpha; // user given alpha
        std::vector<Block*> _blockVec; // vector of blocks
        std::vector<Net*>    _netVec; // vector of nets
        unordered_map<string, int> _name2IntMap; // map of block name to block num in _blockVec

        // for B* tree
        Block* _bSTree; // b* tree root
        Contour _contour; // contour of the b* tree
        vector<vector<Block*>> _bestBSTreeTop; // best b* tree topology

        // for SA
        size_t _numIter; // number of iterations for SA
        size_t _numIterPerTemp; // number of iterations per temperature
        double _temperature; // temperature for SA
        double _coolTemp; // cooling temperature for SA
        double _coolRate; // cooling rate for SA
        double _alpha; // alpha for SA
        double _beta;
        double _gamma;

        size_t _normArea; // normalized area for SA
        double _normHPWL; // normalized HPWL for SA
        double _normAspectRatio; // normalized aspect ratio for SA (R* - R)^2
        size_t _normExceedArea; // normalized exceed area for SA

        Cost _prevCost; // previous cost for SA
        Cost _cost; // cost for SA
        Cost _bestCost; // best cost for SA
```

- **Implementation Detail:**

Floorplanning flow:

```
initPlace();
initPerturbation();
SA_basic();
retrieveBestBSTree();
while (!_bestCost.valid) {
    putBlockInDamnBox();
    retrieveBestBSTree();
}
finalTuning();
retrieveBestBSTree();
```

For the initial placement, I arrange the blocks starting from the bottom-left corner and move toward the right. If a block exceeds the outline width, I place the next block on a new row above. This process is repeated until all blocks are placed.

During the initial perturbation stage, I randomly apply the three B*-tree operations several times equal to the number of blocks. After perturbing the floorplan, I compute the normalized area, HPWL, and exceed area, which are then used to set up the cost function for simulated annealing.

After completing the initial placement, I perform the first stage of simulated annealing. The setup is as follows:

- ✓ Initial temperature: 500
- ✓ Cooling rate: 0.98
- ✓ Cooling temperature threshold: 0.001
- ✓ Iterations per temperature: 1000
- ✓ Cost function: $\alpha * \frac{area}{norm_{area}} + (1-\alpha) * \frac{HPWL}{norm_{HPWL}} + \gamma * \frac{exceedArea}{norm_{exceedArea}}$

The parameter γ is adaptively adjusted throughout the annealing process. Initially, γ is set to 0.1 and increases exponentially as the temperature decreases, reaching 1.0

when the cooling temperature threshold is met. The update formula for γ is:

$$\gamma(temperature) = \gamma(previous\ temperature) * e^{ratio}$$

In addition to this exponential growth, γ is dynamically fine-tuned based on the placement validity during annealing:

- ✓ If the temperature is close to the cooling threshold and some blocks still cannot fit within the fixed outline, γ is increased.
- ✓ Specifically, if a certain number of consecutive iterations result in invalid floorplannings, γ is increased by 0.1.
- ✓ Conversely, if a certain number of consecutive iterations produce only valid placements, γ is decreased by 0.1.

This adaptive adjustment of γ effectively ensures that all blocks are successfully placed within the fixed outline by the end of the simulated annealing process.

If the first stage fails to produce a valid floorplan, I initiate an additional simulated annealing process that focuses specifically on fitting all blocks within the fixed outline. I repeat this process until a valid floorplan is obtained. The setup for this additional annealing process is as follows:

- ✓ Initial temperature: 1.0
- ✓ Cooling rate: 0.95
- ✓ Cooling temperature threshold: 0.0001
- ✓ Iterations per temperature: 1000
- ✓ Cost function: $0.5 * \frac{area}{norm_{area}} + 0.5 * \frac{exceedArea}{norm_{exceedArea}}$

After completing the previous steps, I move on to the final stage: the second-stage simulated annealing. The goal of this stage is to optimize the floorplan based on the user-specified area-to-HPWL optimization ratio. The simulated annealing setup is as follows:

- ✓ Initial temperature: 1.0
- ✓ Cooling rate: 0.95

- ✓ Cooling temperature threshold: 0.001
- ✓ Iterations per temperature: 1000
- ✓ Cost function: $\alpha * \frac{area}{norm_{area}} + (1-\alpha) * \frac{HPWL}{norm_{HPWL}}$

B*-tree delete and insert operation:

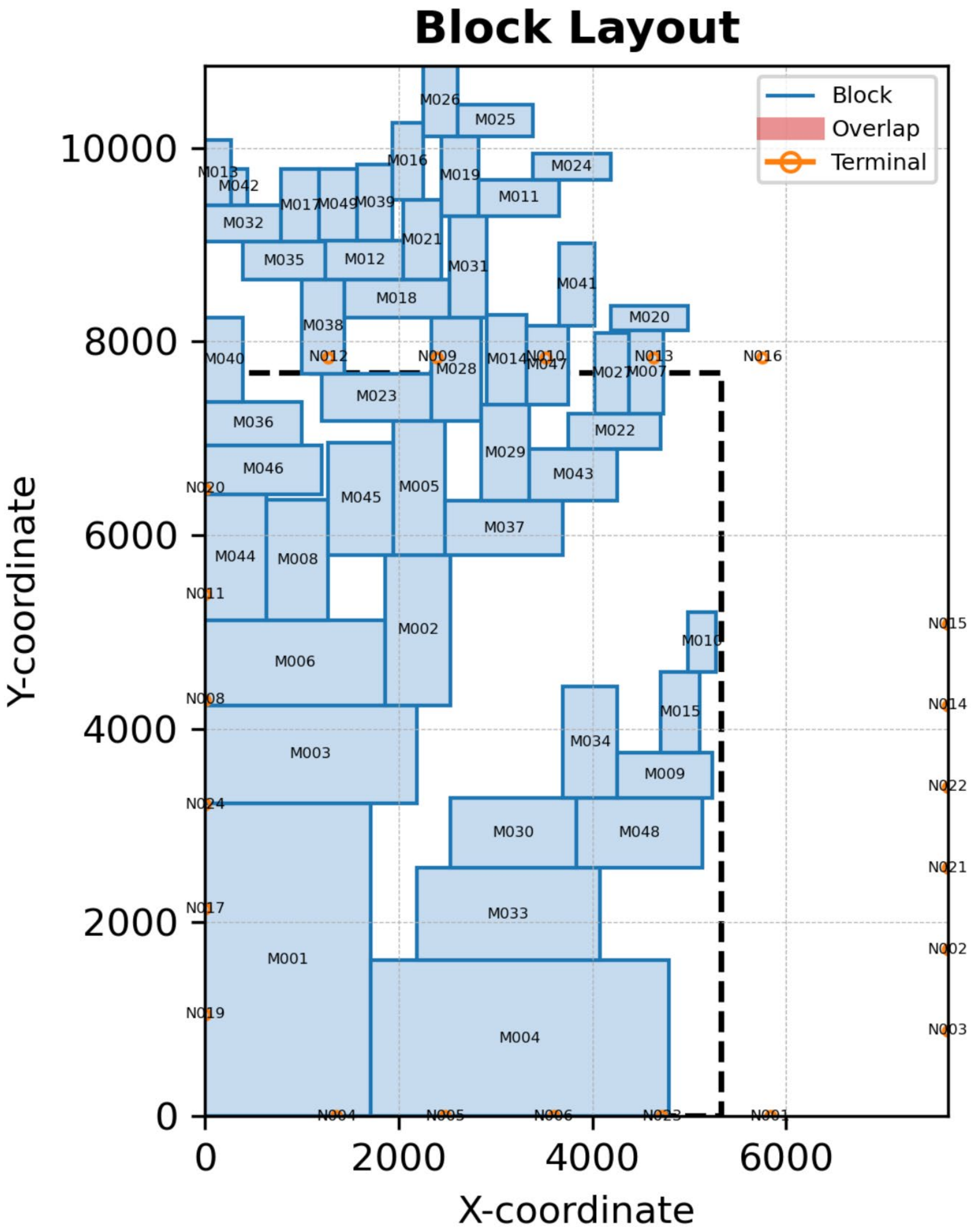
In node deletion, there are three possible cases:

1. The node is a leaf: In this case, the node can simply be deleted.
2. The node has one child: I delete the node and directly connect its child to the node's parent. Whether the child becomes the left or right child of the parent is determined randomly.
3. The node has two children: In this situation, I randomly select one of the two children and swap it with the node to be deleted. I continue this process recursively until the node becomes either a leaf or has only one child, at which point I can apply one of the simpler cases above. If undoing the deletion is required later, I record the sequence of swapped blocks so that I can reverse the swaps and restore the original B*-tree topology.

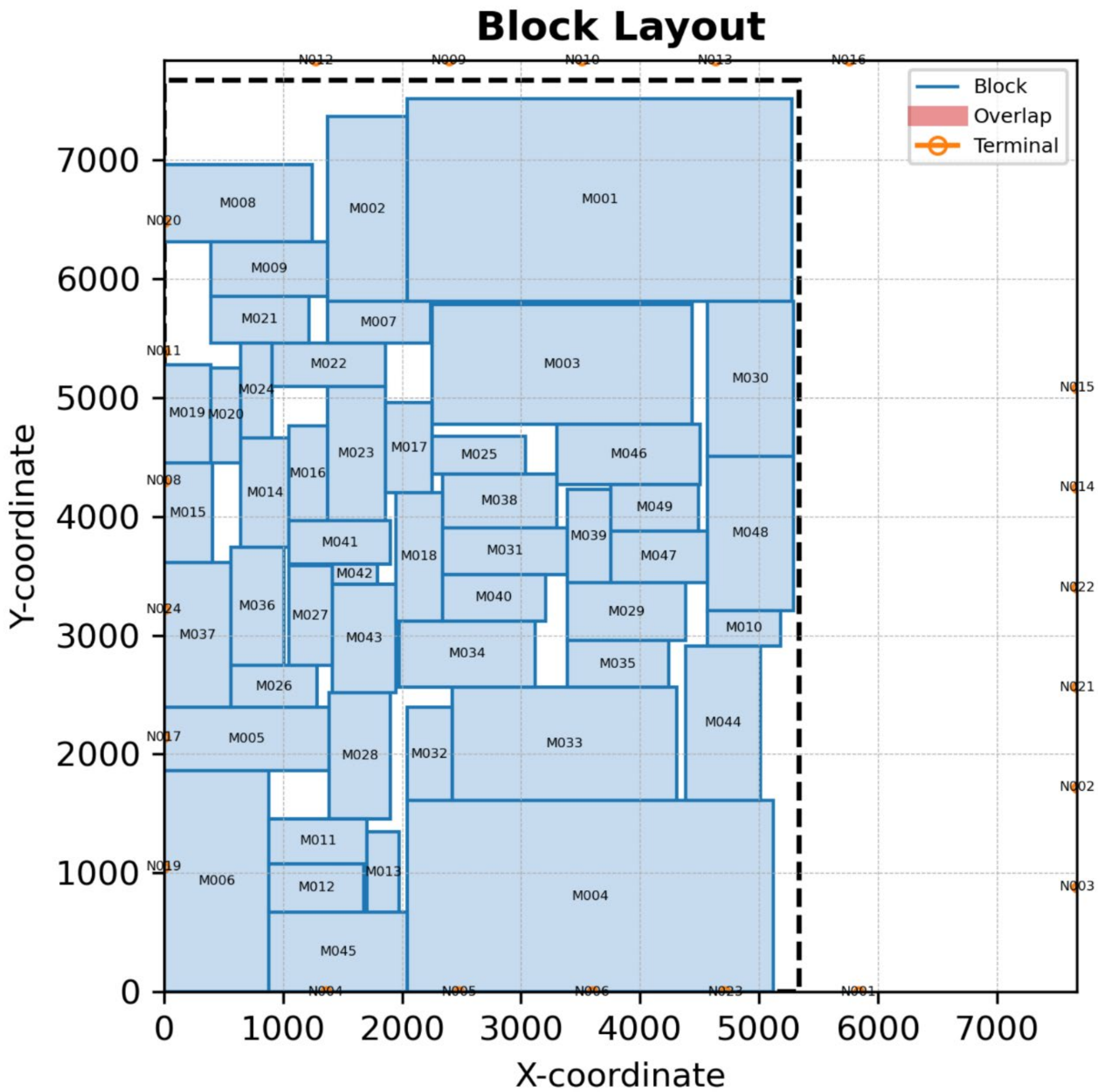
In node insertion, I randomly select a node and insert the previously deleted node as either its left or right child, chosen randomly. The original child on that side (if it exists) will then be randomly assigned as either the left or right child of the newly inserted node.

- **GUI Feature:**

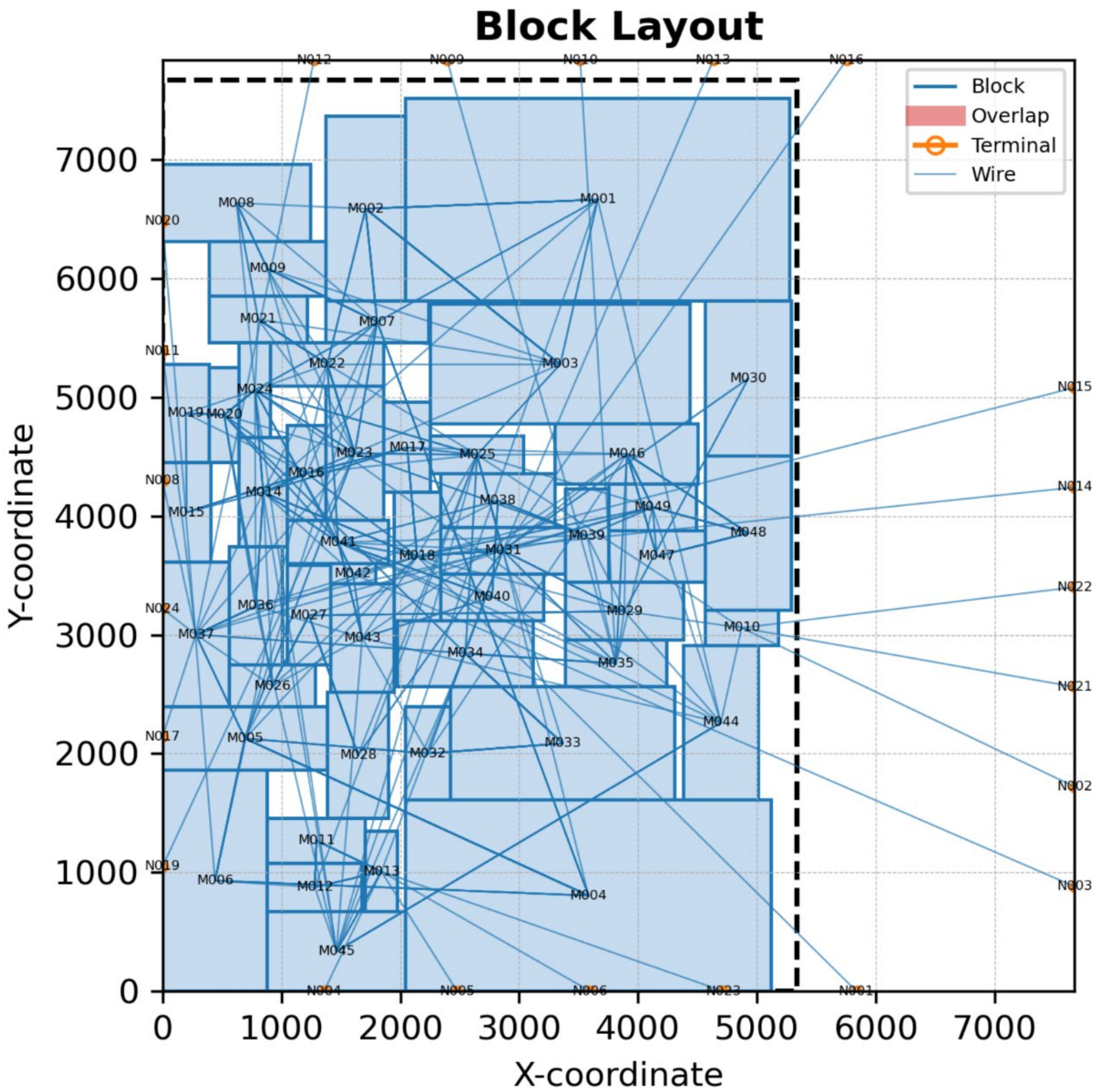
Initial placement example (ami49):



Result example (ami49):



Block Layout
Block
Overlap
Terminal
Y-coordinate
X-coordinate
M001
M002
M003
M004
M005
M006
M007
M008
M009
M010
M011
M012
M013
M014
M015
M016
M017
M018
M019
M020
M021
M022
M023
M024
M025
M026
M027
M028
M029
M030
M031
M032
M033
M034
M035
M036
M037
M038
M039
M040
M041
M042
M043
M044
M045
M046
M047
M048
M049

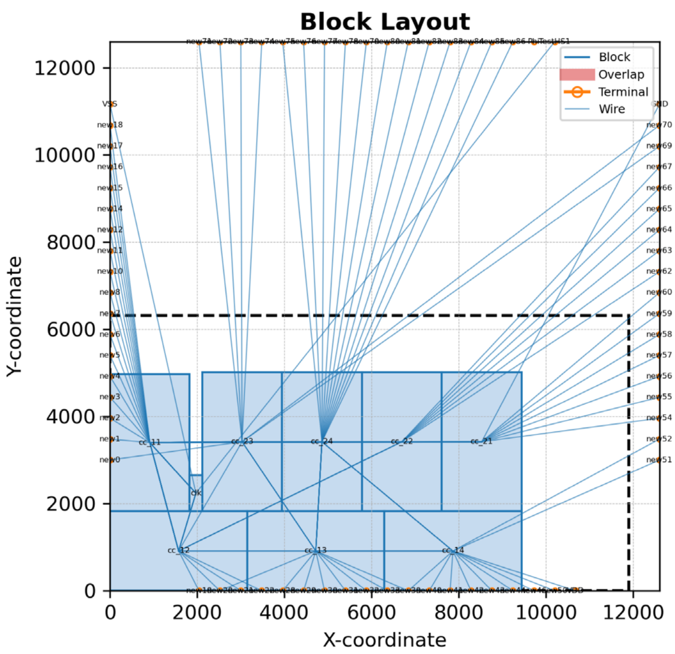


- **Observation:**

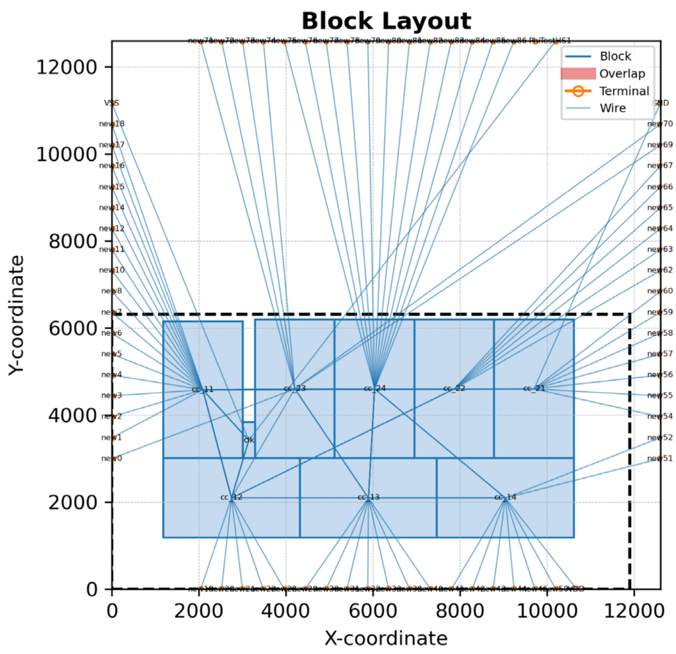
1. I experimented with two initial placement methods. The first method builds the initial B*-tree as a balanced binary tree. The second method places blocks from left to right until the outline width is exceeded, then moves to the next row above. Based on my experiments, the second method produced better results.

2. Using two-stage simulated annealing with different cost functions yields better results than using a single cost function. In my approach, the first stage focuses more on fitting all blocks into the fixed outline, with the exceed area term weighted can be as heavily as half of the total cost. In the second stage, the optimization emphasizes the user-specified ratio between area and HPWL.

3. From my observations, the terminals are distributed along the edges of a rectangle — meaning that if we sequentially connect all the terminals, they would outline a rectangular shape, usually larger than the fixed outline. However, because of the nature of B*-tree packing, the blocks are typically clustered toward the bottom-left corner, which is not ideal for wirelength optimization. To address this, I introduced a simple adjustment: after generating a floorplan, I shift all the blocks together within the fixed outline, within a controlled range. This helps align the floorplan better with the terminal distribution without changing the total area, and as a result, improves overall wirelength performance.



hp before shifting



hp after shifting