Physical Design PA2 report

Data structures

1. B*tree
B*tree is a binary tree, consists by a ***Node data structure***.

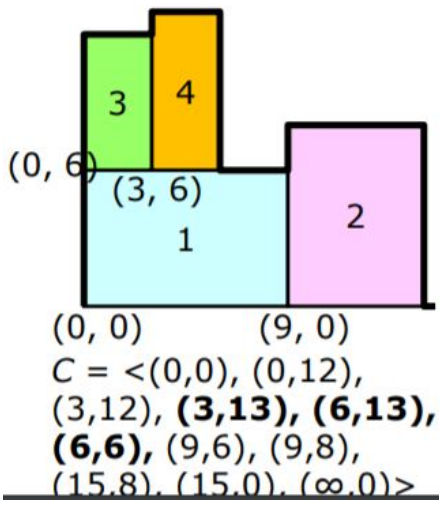
```
Each node data structure consist of:
(1) parent pointer
(2) left child pointer
(3) right child pointer
(4) rotation: define if the block is rotate
(5) id: the index of the ***Block data structure*** in the Block vector
```

From the binary tree representation, we can packed it into a compact floorplan structure with a simple depth first search (DFS) and insertions in a ***Contour data structure***, the detail packing algorithm is shown in the algorithms part.

2. Contour data structure
A Contour datastructure is a simple ***single linked list*** (though the original paper utilize a double linked list, a single linked list is enough).And the data stored in the list is a ***Contour node data structure***.

```
A contour node data structure is consists of:
(1) next: the pointer point to the next Countour node
(2) center: a (x,y) conordinate point
```

Each contour data structure represents a contour of our current packing (see the figure below). Everytime when we packed a new block, a ***insert block operation*** is called, the detail of this operation is shown in the algorithm part.


3
4
(0, 6)
(3, 6)
1
2
(0, 0)
(9, 0)
C = <(0,0), (0,12), (3,12), **(3,13), (6,13), (6,6),** (9,6), (9,8), (15,8), (15,0), (∞,0)>

3. Block, Pin, net data structure
A pin data structure represents a pin terminal in the floorplan, consist of x-value, y-value and its name.
A block data structure is a derived class of the pin data structure, and it is consists of (x,y) values of its left-bottom corner, height and width of the block, and a Node data structure.
A net data structure is a vector of pointers pointing to the Pin data structure (pointing to a Pin or a Block).

Algorithms

1. Fast Simulated Annealing (Fast-SA)
Compared to a Classical SA, which reduces temperature by a constant ratio. Fast-SA has three stages of teh annealing process: (1) High-temperature random search stage, (2) Pseudo-greedy local search stage and (3) uphill climbing search stage. The temperature setting is set

as follows.

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

The first stage runs only 1 iteration with a very high temperature. The delta average is the average uphill cost of this problem, which is calculated by a random SA scheme. With the very hill temperature, uphill moves are mostly taken; therefore, random searching on the solution is performed in this stage.

The second stage runs k itereations with a extremly low temperature based on the average cost difference. The parameter is a user define parameter, we set it 7 in our problem. Also, the parameter c is set high to let the temperature low, we set it 100 in our problem. Notice that the current temperature averge cost difference is considered in the termperature formula. This is because, if the averge difference is small, that means the neighboerhood structure changes a little, so we reduce temperature to more to reduce number of iterations.

The third stage is similar to the classical SA while introducing a average cost term same as stage two.

2. Cost function evaluation technique

The original cost function is:

$\alpha * Area + (1 - \alpha) * Wire$

But if we simply apply this cost function in our problem, the resulting floorplanning will not be in the outline with high probability.

Therfore, we records the feasiblity rate of the recent n floorplans, the n is set 100 in our problem.

Then, a cost_ratio parameter is calculated by:

$cost\_ratio = 1.5 - feasible\_rate$

This value is set not to exceed 0.9 in our design.

A overflow_area_ratio is set as:

$overflow\_area\_ratio = (max(x_{best}, x_{outline}) * max(y_{best}, y_{outline}))/(x_{outline} * y_{outline})$ if it is infeaisble, and it is set to 0.

Therefore, the final cost function we proposed is:

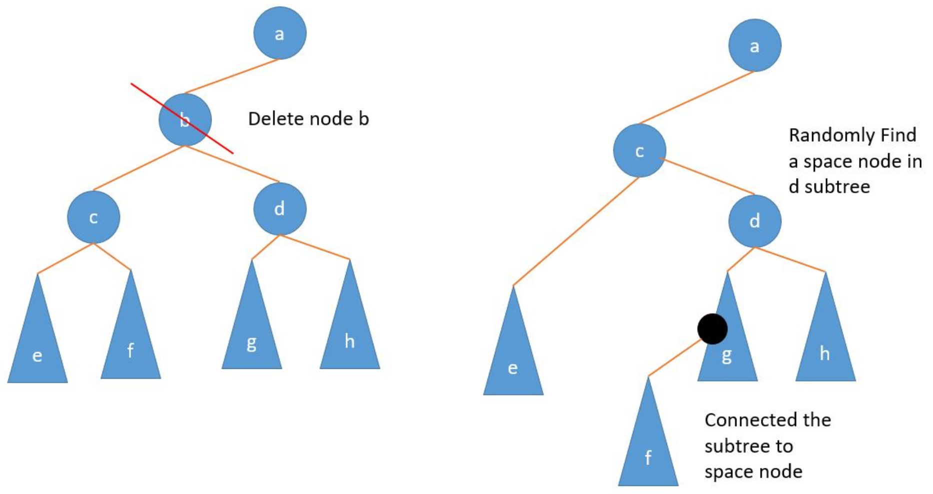
$(\alpha * Area + (1 - \alpha) * Wire) * (1 - cost\_ratio) + overflow\_area\_ratio * cost\_ratio$

We have tried lots of cost functions, and find this results fastest convergence.

3. B*tree opertions with artial Recovery

We define **three types of operation** in our problem:
(1) block rotation:
Operation on tree: Randomly choose a Node data structure rotate it.
Correponding reprsentation: Rotate a block
(2) swap nodes:
Operation on tree: random pick two different nodes and swap them.
Correponding reprsentation: pick two blocks and exchange them
(3) delete and insert nodes:
Operation on tree: random pick two different nodes, delete the first node and maintain the binary tree structure, and insert it as a child of the second node.
Correponding reprsentation: Move a block to other place

The details of **perturb operation**:
(1) block rotation: set the rotate element of the node inverse
(2) swap nodes: swap parent, left child, right child node of the two nodes. Notice that if two nodes originally have a parent-child relation, say Node A has right child Node B, then we can’t simply swap right child pointer of both of them, that will result in B's right child pointer pointing to itself. We need to set B's right child pointer to A.
(3) delete and insert nodes: in the delete operation, we can seperate it into three kinds.
a. the delete node has no child: simply delete it.
b. the delete node has one child: take its child to connected to its parent.
c. the delete node has two child: randomly take one of its child to replace its place, say we take the left child. Then, the left child’s right subtree is then removed and connected to a blank space in the delete nodes right subtree, see the figure below.

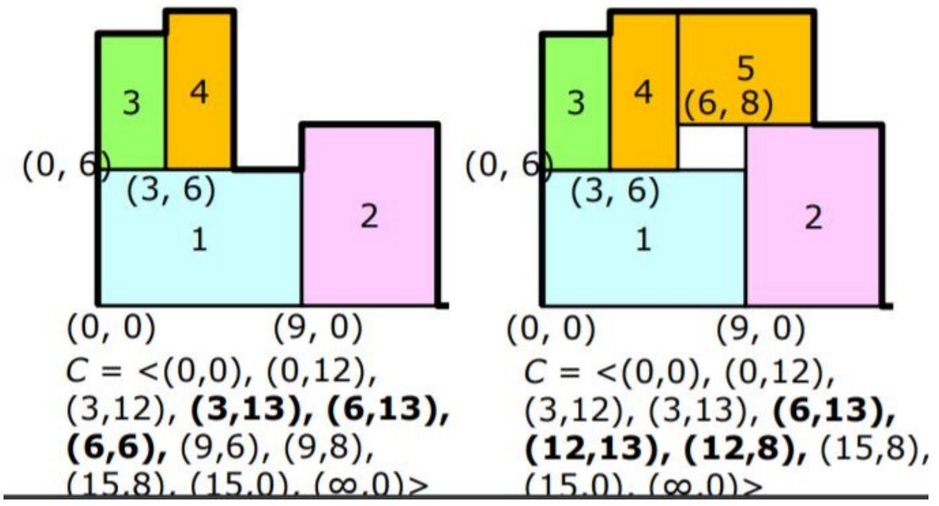

a
Delete node b
b
c
d
e
f
g
h
a
c
Randomly Find a space node in d subtree
d
e
g
h
f
Connected the subtree to space node

in the insert operation, we simply connected to the chosen node and sets its original child subtree to the insert nodes child subtree.

The details of **deperturb operation**, that is we need to recover to the solution when we have not done the perturb operation. Notice that instead of recording all information of the nodes, we apply a partial recover technique to allow only changing information of a few number of nodes.
(1) block rotation: same in perturb
(2) swap nodes: same in perturb
(3) delete and insert nodes: based on the recorded node and direction, delete the insert node and insert back to its place.

Notice that swap and block rotation is $O(1)$ complexity, while insert delete is worst time $O(n)$ and average time $O(lgn)$.

4. Packing algorithm of a B*tree
By a simple DFS we can get the x-value of each block. the x value of a right node is the same as its parent, while the left child has the x value of its parent plus its parents width (height if the parent block is rotate).
The simple DFS operation has complexity $O(n_{blocks})$.
For y value, for each new add block we calculate its projection interval on x based on the x value calculated by DFS. Then, we find the highest y value Contour node data strcuture that is in this projection interval. The new add block has it y value as this max y value. Also, update the contour data structure by removing contour nodes in the projection interval and add new contour nodes of this interval, see figure below for an exmaple.
The y value calculation has amortized complexity $O(n_{blocks})$.


3
4
(0, 6)
(3, 6)
1
2
(0, 0)
(9, 0)
C = <(0,0), (0,12), (3,12), (3,13), (6,13), (6,6), (9,6), (9,8), (15,8), (15,0), (∞,0)>
3
4
5
(6, 8)
(0, 6)
(3, 6)
1
2
(0, 0)
(9, 0)
C = <(0,0), (0,12), (3,12), (3,13), (6,13), (12,13), (12,8), (15,8), (15,0), (∞,0)>

Discussions

(1) Our cost functions is better than the aspect ratio based method:

Since aspect ratio and area has a trade-off relation, if we set the cost ratio of aspect ratio high, SA tends to fits the aspect but lack optimizing the area, resulting a infeasible solution with high probability. If the cost ratio area is too high, then the aspect might not be considered, also results in infeasible solution in high probability.

Though the infeasible rate recording can help to set this cost ratio, it might still not able to find a feasible solution. The initial cost ratio is still tuned by the user.

By experiment, if we penalzie by the overflow area ratio out of the outline, we can get better convergence rate, and can get into the outline of the 5 testcases easily.

(2) The overall complexity is $O(n_{steps} * n_{terminals})$.

$n_{terminals}$ is the number of pins and blocks in the net.

Since in each step, we need to evaluate the wire length cost, and the wire length cost requires seeing through all the terminals, it requires complexity $O(n_{steps} * n_{terminals})$.

Though B*tree operations are also done in each step, it has only $O(n_{blocks})$ in each step, n_{blocks} is always much smaller than $n_{terminals}$, therefore, complexity is still $O(n_{steps} * n_{terminals})$.

Results

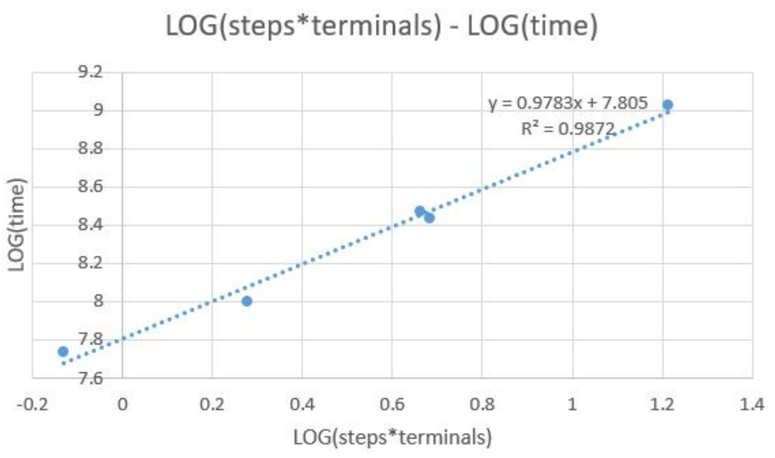
	hp	**xerox**	**apte**	**ami33**	**ami49**
Num. of terminals	226	459	278	425	922
Num. of blocks	11	10	9	33	49
Num. of pins	45	2	73	40	22
Num. of Nets	70	182	96	121	396
Total Area	8830584	19350296	46561628	1156449	35445424
Outline Area	20046048	37314123	75098716	1374905	40843128
Infeasible ratio	44.05%	51.86%	62.00%	84.11%	86.78%

$\alpha = 0.25$	**hp**	**xerox**	**apte**	**ami33**	**ami49**
Runtime	3.77	1.56	0.82	4.82	17.34
Num. steps	1070520	292800	213291	639540	1225970
Wire length	252350	493329	748244.0	85309.5	1331785.0
Area	9852528	21050792	47313280	1509543	40698420
Dead space	10.37%	8.07%	1.58%	23.40%	12.9%

$\alpha = 0.5$	**hp**	**xerox**	**apte**	**ami33**	**ami49**
Runtime	4.61	1.89	0.74	4.83	16.38
Num. steps	1312300	359200	196728	642180	1158355
Wire length	251846	698203.5	774456	89821.5	1282764
Area	9852528	34576752	47761324	1372686	40526136
Dead space	10.37%	44.04%	2.51%	15.75%	12.54%

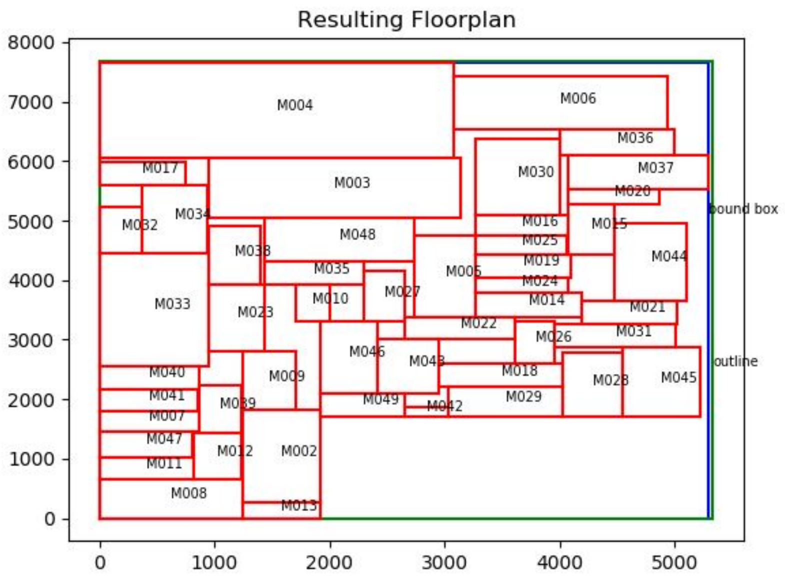
$\alpha = 0.75$	**hp**	**xerox**	**apte**	**ami33**	**ami49**
Runtime	5.06	1.82	1.05	5.37	17.42
Num. steps	1436600	343400	261899	712800	1227940
Wire length	265929	688704.5	787159.0	93809.5	1425683
Area	9440340	24638670	47503736	1328635	39444804
Dead space	6.45%	21.46%	1.98%	12.95%	10.14%

The following figure is the run time to $n_{terminals} \times n_{steps}$ when α is 0.5. It shows that run time and $n_{terminals} \times n_{steps}$ are strongly and linearly correlated. So the experiment is consistent with our complexity analysis.



Bonus

We write a simple python program to plot the blocks and outline of the floorplan. Notice that you should install the “matplotlib” package for running the code. The usage of the code is "python plot.py (http://plot.py) <output_file> <blocks>". For example, if you type “python ami49_out ami49.block”, the following figure is going to pop up!



Reference:

[1] Y. C. Yang, Y. W. Chang, G. M. Wu, and S. W. Wu, “B -trees: A new representation for nonslicing floorplans,” in Proc. Design Automation Conf., June 2000, pp. 458–463.
[2] T.-C. Chen and Y.-W. Chang, “Modern floorplanning based on B*-trees and fast simulated annealing,” IEEE Trans. Comput.-Aided Design Integrat. Circuits Syst., vol. 25, no. 4, pp. 637–650, Apr. 2006.