# Physical Design Report in PA2

R08921053 電機研一 梁峻瑋
r08921053@ntu.edu.tw

**※程式實驗結果**

| | 運算時間 (s) | 給出寬度/限制 | 給出高度/限制 |
|---|---|---|---|
| ami33 | 12.3552 | 1323/1326 | 1204/1205 |
| ami49 | 15.3033 | 5334/5336 | 7672/7673 |
| apte | 13.6207 | 11822/11894 | 6316/6314 |
| hp | 10.8857 | 5390/5412 | 3724/3704 |
| xerox | 20.6668 | 6937/6937 | 5390/5379 |

**※設計的資料結構與演算法**

首先，我把整個程式作業的工作拆成了下列的函數來分工運作.

```
1.  void parseInput_blk(fstream&);
2.  void parseInput_net(fstream&);
3.  void floorplan(double);
4.  double buildplan(double, double&, double&, double&);
5.  void packing(Macro);
6.  void coordinate(Macro, Level, bool);
7.  size_t nowX(Macro*, Level*, bool);
8.  size_t nowY(Macro*, Level*, bool, size_t);
9.  void   Range(size_t&, size_t&);
10. double Length();
11. void report(double, fstream&);
```

基本上, parseInput_blk 和 parseInput_net 是讀入的”Read-in”函數. 而 packing()則是把 block 裝入 B* tree 的函數.

Buildplan()則包含了計算座標-coordinate(), 尋找當下 Block 該放置的(x,y)座標-nowx(),nowy(), 以及計算電路長寬-Range(), 導線長-Length()等等的函數.

換言之, buildplan 就是從給定 B* tree 後, 到得出 cost 之間的所有工作總結. 最後, floorplan 則需要做 Simulated Annealing, 分別從 B* tree 來計算 S 和 S’的 cost, 再利用 rand()的給值來判斷是否移動.

**※問題 1: 如何實作"amotized O(1)-time 查找當前高度"這個工作**

事實上，這就是我的 nowy 函數所做的工作. 基本上在計算座標時，排入每一個 block 都需要先計算左下角安插的(x,y)座標. 很顯然地，參考當前 block 是 parent 的 left child 或 right child，就能在 O(1)時間計算出 x 座標，即 nowx 工作.

另一個前備工作，是在安插完每一個 block 後，要順便紀錄當前 k 個 block 整體的水平線高度為何，也就是課堂提到的"doubly linked list". 直觀來看，只要拿 parent-block 的安置位置，得到 liked list 其中的一個 node，就可從這個 node 往後推，在接近 O(1)時間內，掃過底下壓到的所有 block 的高度，得到 max(高度).

由於上述的方式，不會頻繁地回頭查找後方的 block 或水平線高度，所以整體來分析，確實是線性的時間.

**※問題 2: 所以簡單來說，要如何計算所有 block 的左下角座標?**

首先，我們之前先擺好了一棵 B* tree. 也就是 packing 階段的工作. 或者是 OP1, OP2, OP3 修改完後的樣貌. 實作上是 doubly linked list 的結構.

再來，對這棵 B* tree 使用 pre-order 來拜訪 (traversal). 每當拜訪了一個樹的 node，就要計算這個 block 的左下角放置的座標，並且利用 amotized O(1)-time 的動作來更新水平線高度，基本上就是在水平線高度這個 doubly linked lis 上再插入一個 node，順便移除掉老舊的 node，進行更新.

最後，當我們的 travsersal 結束的瞬間，就可以得到所有 block 應放置的位置.

**※效能取捨: 額外使用 map 結構來簡化實作複雜性.**

比較麻煩的一點是，我計算所得的座標都是儲存在 linked list 上，每次查找資料都需要使用 tree traversal 在 O(n)時間內找座標，相當的費力且麻煩. 於是，我決定犧牲一部分的時間，使用 map 資料結構，來對應 block 和他的座標. 仔細分析一下這個做為的代價:

| | 一次查找時間 | 全體查找時間 | 實作考量 |
|---|---|---|---|
| 用 linked list 查找 | O(n) | O(n) | 相當繁瑣 |
| 額外用 map 查找 | O(logn) | O(nlogn) | 相對簡易 |

**※待改進的問題**

值得一提的是，我在實作 simulated annealing，比較 S 和 S'時，實際上也使用了較簡潔的方法: 首先在 packing 完成 B* tree 後，把 S 的 wire-length, area, ratio 等等都算出，再移除所有的資料，挑選一種 Operation 來修改 B* tree，並且重新計算一次 wire-length, area, ratio. 最後，如果選擇 S'，就不須更動; 如果選擇 S，就再進行一次相同的 operation 把資料轉換回去.

在這個實作方法上，具備以下的缺點:

1. 因為很難把所有使用的資料刪除乾淨，因此可能會有 dead data，造成記憶體的浪費，在 Simulated annealing 進行最後的迴圈運算時，造成 memory 不足，甚至是 killed 的問題.

2. 每一個 Simulated annealing 的回合，都需要分別重頭計算 S 和 S'的 cost，相當的不具備效率. 理論上，是可能把選擇的 S, S'結果，儲存到下個回合，節省下一半的運算及資源的.

一個可以改進的點，可能是把選擇的 S 或 S'結果，保留到下個回合使用，進而加速 50%的運算效率(理論層面). 另外一個可以改進的點，則是採用另一個新開的空間來計算 S'，不需要每次計算完後都把資料刪除光，才能進行選擇.