2-Way F-M Circuit Partitioning Report

姓名：李宇哲
學號：B10732040
校系：國立台灣科技大學 四資工四乙

一、 Pseudocode

```
procedure partition
begin
    Input File
    parseInput()
    initialPartition()
    iterNum ← 0
    do
        accGain ← 0
        maxAccGain ← 0
        bestMoveNum ← 0
        Clear moveStack

        for i = 1 to numOfCell do
            maxGainCell, maxGain ← get_maxGain_cell_and_value()

            Push maxGainCell into moveStack
            Lock maxGainCell
            Erase maxGainCell from the bucket list

            G_before ← Group before maxGainCell move
            G_after  ← Group after maxGainCell move
            Increase size of G_before
            Decrease size of G_after

            Update_Gain(maxGainCell)          // Refer to Lecture Note #3, page 14
            Update bucket list

            accGain += maxGain
            if accGain > maxAccGain then
                maxAccGain ← accGain
```

```
                bestMoveNum ← i
            endif
        endfor

        increase iterNum

        moveCell()
        Unlock all cell

    while maxAccGain > 0
```

※函式介紹:

1. get_maxGain_cell_and_value():

   從 Bucket List 中找出 gain 值最大，且換到另一個 group 仍然平衡的 cell，找到後回傳該 cell 及其 gain 值。

2. moveCell():

   C = { moveStack[0], moveStack[1], ・・・, moveStack[bestMoveNum] } 是當前 iteration 要移動的 cell 集合，移動 cell 這個動作會需要更新以下資訊：

   (1) 將每一條 net 於兩個 partition 中的 cell 數量初始化為 0
   (2) 將每一個 cell 的 gain 值初始化為 0
   (3) 將 C 集合內的 cell 移到另一個 partition
   (4) 更新兩個 partition 的 cell 數量
   (5) 更新每一條 net 於兩個 partition 中的 cell 數量
   (6) 更新 Gain 值及 Bucket List

二、 資料結構

我的程式是修改課堂提供的 sample code，所以這邊我只介紹自己**更動過或是新增的資料結構**：

1. map<int, Node*> _bList 改成 vector<Node*>

   原因：

   std::map 底層是用 red-black tree 實作的，因此每一次 access bucket list（搜尋 key）都需要花 O(logN)的時間複雜度（N 為 bucket list 的大小 = 2 * maxPinSize + 1），有一點費時。

   我們已知 bucket list 最大容量為 N，且 std::vector 支援 random access，因此換成用 vector，可以把每一次 access bucket list 的時間複雜度降到 O(1)。

2. bucket list 裡的 list 改成 Doubly Circular Linked List

原因:

我在更新 Bucket List 及從中挑出最大 Gain 值得 node 時，可以分成兩種模式，分別為 Last In First Out(LIFO)和 First In First Out(FIFO)。原本 Bucket List 是用 Double Linked List，它可以用常數時間完成 LIFO，卻得用線性時間完成 FIFO，因為如果要將 node 插入到最後一個位子的話，需要遍歷前面所有的 node，非常沒有效率。
若改成 Doubly Circular Linked List 的話，LIFO 和 FIFO 都可以用常數時間來完成。

三、 我的發現

1. 把程式碼長度短且頻繁被呼叫的函數變成內嵌函數（在函數前加上 inline），可讓程式的 runtime 變小。
2. 承二 2.，我分別實驗了用 LIFO 和 FIFO 來更新及挑選 Gain 值最大的 node，發現在大部分的測資中，用 LIFO 都能獲的更好的 cut size，以下是比較表：

Test Case	Cut Size with LIFO	Cut Size with FIFO
Input_0.dat	7481	15076
Input_1.dat	1253	1233
Input_2.dat	2199	2226
Input_3.dat	27627	28081
Input_4.dat	44613	45673
Input_5.dat	143498	144087

因此，我的程式最終採用 LIFO 模式。

四、 最終結果

Device:

Type: HPE ProLiant DL360 Gen10
CPU: Intel Xeon
CPU Clock: 2.4 GHz * 40
Memory: 64 G
OS: Ubuntu 20.04

Test Case	Cut Size	Run Time(s)
Input_0.dat	7481	5.711
Input_1.dat	1253	0.036
Input_2.dat	2199	0.070
Input_3.dat	27627	6.329
Input_4.dat	44613	10.762
Input_5.dat	143498	34.694