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

1. 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](https://cool.ntu.edu.tw/courses/12758/files/1763723?wrap=1), 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

1. 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](https://www.geeksforgeeks.org/doubly-circular-linked-list-set-1-introduction-and-insertion/)

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。

1. 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。

1. 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 |