VLSI Physical Design Automation HW2
王領崧 107062107

P.S. 執行和測試數據都是在 ic51 上面

2) How to compile and execute your program

– Compile:

”./src”資料夾輸入 make，即可 compile，生成執行檔 ”hw2” 在 ”/bin” 資料夾裡面。

e.g.

$ make

g++ ./main.cpp LCN.cpp Partition.cpp –std=c++11 –O3 ––optimize –o ../bin/hw2

– Execute:

在 ”./bin” 資料夾中，用 ./<exe> <net file> <cell file> <output file> 格式輸入，即可執行。

e.g.

$./hw2 ../testcases/p2–1.nets ..testcases/p2–1.cells ../output/p2–1.out

3) The final cut size and the runtime of each testcase

	p2–1	p2–2	p2–3	p2–4	p2–5
Cutsize	5	123	636	45746	124773
Runtime	0.1001	0.16	3.73	17.74	6.79
Seed	670768047	–380638439	–848872715	–749993789	–29006286

4) Analyze your runtime

	p2–1	p2–2	p2–3	p2–4	p2–5
I/O	0.001	0.01	0.19	0.36	0.93
Computation	0.1	0.15	3.54	17.38	5.86
Runtime	0.1001	0.16	3.73	17.74	6.79
I/O Ratio	1%	6.25%	5%	2%	13%

由上面的圖表可以看得出來，不論是何種大小的 testcase，I/O 的時間都佔不到整體的 10% (除了第五個 testcase)，整體的 bottleneck 是在 Computation time，因此此次的優化都在 computation 上面。另外 p2–4, p2–5 雖然 net, cell 的數量一樣，但是由於 totalCellSize & pin 的數量都不同，在 computation 和 outsize 上都有著極大的差異。

與其他同學討論後，也發現 implementation 的不同，會導致 p2-4, p2-5 執行時間快慢的差異，像我自己是 2-5 < 2-4，但有的同學是 2-4 < 2-5，我認為這點很符合 FM 的本質是 SA 的特性，不同的執行過程和方式都會使得最後出來的結果和時間有差異性。

5)

– Difference between your algorithm and FM Algorithm

我的演算法基本上跟講義的 FM 是一致的。主要分為兩個 part: Initialize, Loop。Initialize 為進行初始設定，包括將 net, cell 從文件裡面 load 進來 -> random partition cells 為兩堆 -> 計算 net distribution -> 計算 cells 的初始 gain -> 計算 bucketList Pmax -> 根據 cell 的 group 和 gain 來建造兩個 bucketList。

```
// Initialize part
void Init();
void RandomPartition();
void CalculateInitNetAB();
void CalculateInitGain();
void CalculatePmax();
void BuildInitBucketList();
```

```
// Loop part
int Loop(bool firstTime);
void CalculateNetAB();
void CalculateGain();
void BuildBucketList();
void FreeCellLock();
int UpdateGain(Cell *baseCell);
Cell* GetBaseCell(int cannotMove);
Cell* GetBaseCellFromGroup(int group);
bool CheckIfBalanced(Cell *baseCell);
void ForwardToMaxPartialSum(int step);
void UpdateGroupSize(int from, int to, Cell *cell);
```

Loop 相當於每個 path，1) 每次都要先更新 cell gain、net distribution -> build bucketList -> free cells lock。2) 接著開始找這一輪的 baseCell，與講義上的做法相同，比較兩個 group 中最大 gain 的 list 的 cell，並且不會違反 balance 的條件，拿取較大 gain 的 cell 作為 baseCell。3) 然後針對將這個 baseCell 移動到另外一個 group、status 設為 lock，並且 update 有關聯 cell 的 gain (此部分的實作與講義的 pseudo code 相同)。4) 當所有的 cell 都被移動過後，我們會找出 maxPartialSum step，並做一個 recover 的動作，到此即為一個 path 的結束。

– bucket list data structure

我的 bucketList 是由 unordered_map <int, Node *> 所構成，int 對應那一層的 gain 值，而 Node * 對應相同 gain 的 cell 所組成的 doubly linked list 的每個 node。為了簡化 insert / delete node，我在每個 doubly linked list 都加一個 dummy node 作為輔助，因此 bucketList[gain]->next 才會是相應 gain 的第一個 cell。

– find the maximum partial sum and restore the result

Find the maximum partial sum: 會有一個變數紀錄目前最大的 partial sum，有一個變數紀錄最好的 step，有一個 stack 紀錄 baseCell 的順序。每次 update_gain

後，都會回傳當前 baseCell 的 gain，經過累加比較後，就能知道新的結果是否會優於目前最佳 partial sum，如果有的話就更新上述變數，反之就維持。
Restore the result: 因為使用 stack 來紀錄 baseCell 的順序，以及此次 path 最好的 partial sum 的步數，因此 restore result 的部分只需要依序 pop stack 中的 baseCell 並且做復原，直到最好的步數。

– speed up
將 map 改為 unordered_map
在尋找 baseCell 時，講義上的做法是將 gain 由大到小的 cell 做比較 & verify 是否 balance，但在經過觀察實驗數據後發現，當這個 gain 的 cell 不符合 balance 時，有很大的機會相同 gain 的其他 cell 也會不符合 balance，因此為了加速篩選 baseCell 的速度，當遇到不 balance 的情況時，就會直接去 gain–1 的 list 裡面做 search，經由實驗後發現得到的 minCutSize 不會因此變差，而且執行速度快上許多。

– Parallelization: 無

6) compare your results with the top 5 students

	Cut size					Runtime (s)				
Ranks	p2-1	p2-2	p2-3	p2-4	p2-5	p2-1	p2-2	p2-3	p2-4	p2-5
1	6	191	4441	**43326**	**122101**	0.01	0.07	3.05	5.01	42.06
2	6	**161**	1065	43748	125059	0.01	0.1	3.11	9.84	18.77
3	6	358	2186	45430	122873	0.04	0.78	21.21	115.38	59.78
4	**5**	302	1988	46064	124862	0.03	0.17	7.04	6.93	8.22
5	6	411	**779**	46356	125151	0.01	0.16	5.49	12.31	13.57

	p2–1	p2–2	p2–3	p2–4	p2–5
Cutsize	5	123	636	45746	124773
Runtime	0.1001	0.16	3.73	17.74	6.79
Seed	670768047	–380638439	–848872715	–749993789	–29006286

前三個測資我的 minCutSize 都比第一名還好，而且 Runtime 是差不多的，然而比較大筆的 testcase (p2–4 p2–5) 表現卻比第二、三名還差，雖然執行時間稍好一點。因此綜觀 runtime 和 cutsize，我認為自己比第三名還好，因為他的 runtime 普遍高太多，然後跟第二名各有優缺點，因為在執行時間差不多的情況下，他大測資的表現比較好，而我是在小測資上面。

7) What have you learned from this homework? What problem(s) have you encountered in this homework?

1. STL 的使用

由於這次作業需要 maintain 很多 data structure，像是 cell 連接的所有 nets、net 連接的所有 cells、以及每個 group 各自的 bucketList，因此選對 data structure 可以省下大量的時間。一開始在 maintain bucketList，我就是使用 map 來運作，後來發現這樣效率很差，因為每次 update cell gain -> update bucketList，map 就會因此要再重新排序一次，非常耗費時間。但我其實不需要讓它做排序，因為每次的 access 都是根據 gain 的大小去看，所以使用 unorder_map 就可以了。

2. BucketList 取跟拿的方式影響

我 bucketList 的實作是以 unorder_map<int gain, Node *> 實作，每個 gain 會先 insert 一個 dummy node 作為 doubly linked list 的 head，方便之後 insert/delete node。

一開始實作的時候，我是從最大 gain 的 list head 去取得 baseCell，然後從 list tail 去插入更新 gain 的 cell，但後來發現再跑 p2-1 testcase 時，雖然 minCutSize 能到 5, 6，但機率很低，反之很常出現一些蠻差的 cutSize，然而我如果是同一邊取 / 插入 cell，不論是 head or tail，效果就會變得很好。我覺得會有這種原因是因為 SA algorithm 本身就是存在機率變因，因此不同的調整都有可能導致最後趨近不同的極值。那同邊取/拿會比較好是因為新 insert 的 cell 與前一個 baseCell 有共同的 net 連接，如果能在下一輪能馬上對這種 cell 進行 update，這條 net 就會有很大的機會不會被 cut 到，反之如果先處理不相關的 cell，很可能在 update gain 時，讓那些 cell 的 gain 降低，進而降低 priority，那原本那條 net 很可能就會繼續維持在 cut state 的狀態，最後導致 minCutSize 變差。