CS6135

VLSI Physical Design Automation

Programming Assignment Report

Student ID:1234567

Student Name:陳聿廣

一、 Abstract

本次作業主要是要求實做出一個繞線器(router)。我們將實做重點放在 Global routing，並以 grid base 為基礎，題目給定每條 net 擁有的 pins 的位置(以 grid[x, y]位置表示)，grid-grid 之間可容納的繞線數目(density)，以及 grid 是否為 hotspot 的資訊。題目要求以 C 或 C++實做出一個 router，首要目標是能正確達成 routing 的動作，其次是降低 overflow，再來希望繞線能儘量不要穿越是 hotspot 的 grid，最後希望總繞線長度越短越好。

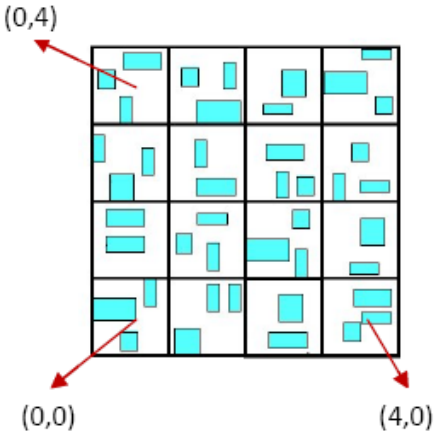
這次的期末作業，我完成了三種不同的 router，第一種 router 是陽春型版本(以下稱為 stupid-route)，採用 L-route 方式，不考慮 density 與 hotspots，單純的就任兩點間，以 L 型方式繞線。第二種 router 採用 Lee’ s algorithm(以下稱為 maze-route)，並透過 trace back 時 always 尋找 density 較大或非 hotspot 的路徑走，以達到降低 overflow 與穿越 hotspot 的 net 數。第三種 router 我以 NTHU-route 為基礎，修改部分參數與 I/O 介面後，得到用 NTHU-route 繞線的結果。

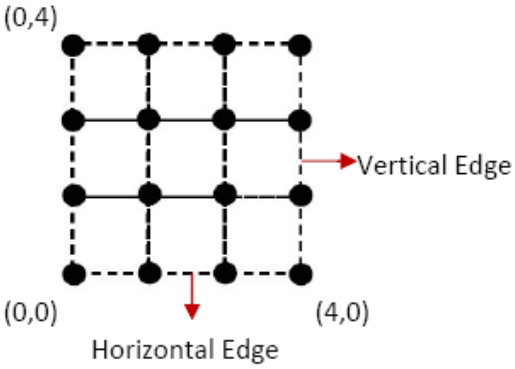
在結果評比方面，整體而言，stupid-route 在各項的表現都是最差，而 NTHU-route 最好。依據這次的評分標準，若比較 overflow 數量， NTHU-route 大勝，幾乎把 overflow 都壓到 0；若比較穿越 hotspot 的 net 數量，NTHU-route 雖然以些微差距得到較好的結果，但差距就不若 overflow 那麼大；而比較 wire length 的結果，仍然是以 NTHU-route 險勝。

二、 Introduction

A. Problem Description

本期末作業主要要求實做出一個繞線器(router)軟體。在 IC 設計中，繞線是一個複雜且重要的步驟，可大略分為 Global routing 與 Detail routing 兩種，本作業主要實做 Global routing 部分。在 placement 步驟後，我們已知所有 circuit 的擺放位置，如何將 circuit 間應相連的 pin 腳們連起來，讓整個 chip 能順利運作，是 Global routing 的主要問題。在本作業中，我們以 grid 方式表示 circuit 在 chip 上的位置，以方便軟體實作。下圖描述了 grid base 在本作業中的定義。





Global routing 需面對的考量點有很多，首先，針對每個 grid 之間，routing 的資源是有限的(routing channel is limited)，即最多只能有有限個 net 從此區穿過。如果 routing 出來的結果，穿越此區的 net 數大於此區所可以容納的 net 數，稱為 overflow。本作業的第一個目標即在如何降低 overflow。第二，grid 中有些位置比較不適合繞線(可能是放在該 grid 中的 circuit 的特性導致)，這種 grid 稱為 hotspot，本作業的第二個目標，在於使繞線的結果通過 hotspot 的線段越少越好。第三，因為繞線長度越長，IC 製造時的成本就越高，同時也越容易出錯。因此，本作業希望將繞線的長度降到最低，也就是使用最少的線條達到繞線效果。

本次作業給定 input file，內容包含 grid 個數，grid 與 grid 間的 capacity，net 的總數，每條 net 的詳細資訊(對一條 net 而言，有哪些 pin，分別落在哪裡)等，要求 output 繞線結果，以每一條 net 的繞線結果呈現。其中，繞線結果輸出只能走垂直或水平兩個方向。評分的依據則依序由 overflow 多寡，穿越 hotspot 的線段個數，總繞線長度決定。

B. Contributions and result

本次的作業我採用三種不同的 routing algorithm，實做出三種不同的 router，分別有不同的結果，詳述如下。

第一種 router(stupid-router)採用 L-route algorithm 實做。L-route algorithm 是一種直覺式的演算法，藉由任兩點間皆可以 L 型繞線，我們可以輕易的完成繞線器，在不考慮 overflow 與 hotspot 的狀況下。此種繞線器複雜度低且能達成繞線的最基本要求，實做這種繞線器的目的，主要是用來與其他演算法做比較。Stupid-router 執行的速度快，但出來的結果很差。第二種 router(maze-router)我採用 Lee's algorithm，也就是 maze route algorithm 實作，透過填表的方式找到 route 的最短線段，

同時在回溯時考量 overflow 與 hotspot 的問題，得到一條僅可能滿足三個條件的繞線路徑。Maze-router 執行速度普通，結果與 L-route 相比，有大幅度的進步。第三種 router 我採用 NTHU-route 的現成程式碼，改良部分原始碼、參數設定與輸入、輸出之 I/O 格式之後得到繞線的結果。NTHU-route 由多位學長開發，且經歷過更新與再版，對於 overflow 的繞線效果自然比自行開發的 router 好很多，不過因為沒考量 hotspots 的問題，在 hotspot 項目上，結果的差距不若 overflow 來的大。

借由這個作業，讓我學習如何自行開發繞線器，能實做出不同版本的繞線器，並比較與分析結果。

三、 Algorithms

A. L-route

L-route 是最簡單的 routing algorithm，對於相異的任兩點，在限制止轉一次彎的前提下，我們可用一條直線或是 L 形狀的連線連接此兩點，達到 routing 的最低要求－把應該接在一起的 pins 都接在一起。若兩點的 x 座標或 y 座標相同，則只須以直線連接兩點即可；若 x y 座標皆不相同，則有兩種 L 形狀可供選擇。若選擇所經過的 grid 的 density 加總較大的 L 形狀，則會有較大的機會避免 overflow 的上升。而選擇所經過的 grid 的 hotspots 加總較小的 L 形狀，則可以減少穿越的 hotspot 個數。又選擇在此 net routing 中已經有人走過部分 grids 的 L 形狀，則可以減少整體 wire length 的長度，因為可共用重疊的部分。

在時間複雜度上，我們只須確定在同一條 net 上的所有 pin 都能找到一個其他點與自己相連，且不會形成迴圈，就算完成 routing。所以對於每個 pin 而言，都必須當成出發點做一次計算，找到一個其他點且不會形成迴圈，再二擇一 L 的繞法，如果 x y 座標皆不同。唯有最後一個點除外。假設一條 net 上有 m 個點，則對一個點找到其他點的時間是 O(m-1)，共須做 m-1 次，費時 O(m^2)。若有 n 條 net，則需費時 O(nm^2)(這只是粗略的分析，因為每條 net 的 pin 數不同，故 m 值也不同，若取最大 m 值可能是非常悲觀的估算)。這樣的時間複雜度並不讓人滿意，一個改良方法是固定一點為核心，所有人都只需考量與核心的兩種或一種繞法，時間複雜度便可降至 O(nm)，不過相對的，solution space 會比較小。

L-route 因演算法簡單，實作迅速且執行速度較快，雖然 L-route 針對本作業的三個目標皆有最佳化的方法，不過因此演算法限制只轉一次

彎，相對的 solution space 被大幅限制，因而大多會得到較差的結果。

B. Lee's algorithm (maze route)

Maze route 中的 Lee's algorithm 是在課堂上講述過的演算法，實作方式為先就眾多 pin 中任選一個起點，由起點開始，針對其上下左右可走的格子填寫數字，之後把此數字加一，再由剛剛的上下左右格子為起點，繼續往該格的上下左右填寫數字，如此不斷重複，直到某個格子為另一個 pin 為止，此格子即為終點。以上的步驟稱為正向尋找。而在找到一終點後，必須反向推回繞線的路徑，反向推回的方法為每次都尋找一個比目前數字小的格子反向推回，直到回到起點為止。反向推回所經過的路徑即為繞線的路徑。之後將目前所繞出的路徑中的每個格子都設為終點，再以其他 pin 出發，直到找到任何一個終點為止。(另一說法是將目前所繞出的路徑中的每個格子都設為起點，依照一樣的方式填表，直到找尋到下一個終點)。這種繞線方式可以保證繞線的正確性，同時針對每次的起點與終點，得到最短距離的繞線結果。

針對這次的作業，首要目標是 minimize overflow，一個可行的做法是在反向推回的時候，若有多於一個格子可供選擇，總是選取 density 較大的格子，如此可以避免大量的 overflow。而對於第二個目標，通過的 hotspot 數量要最少，可行的做法是當通過 minimize overflow 之後，仍然有多於一個格子可選擇時，總是選擇非 hotspot 的格子走，來減少穿越 hotspot 的線段數。而對於第三個目標，則是同 stupid-route 的做法，儘量選擇重疊路徑以節省線段長。

Maze-route 的時間複雜度取決於 grid 的大小，若 grid 的大小為 m*n，則每次的填表最差需要花上 O(mn)時間來做填表，O(mn)的時間尋找回頭路徑，對於單一 net 而言，時間複雜度是 O(mn)。若假設有 k 條 net，則最高須花上 O(kmn)時間。

Maze-route 的實作著實比 stupid-route 複雜許多，但其實上述的最佳化方法，是限制於符合 maze-route 的基本精神－尋找最小數字的路徑反向推回，所以 solution space 仍有限制，結果也就不那麼完備。

C. NTHU-route

NTHU-route 是學長開發的成果，routing algorithm 極其複雜，雖在上課時曾經講授，但我仍不是百分之百的了解。由於各參數之間互相有關連性，又有 iteration 不確定性的問題，時間複雜度並不好分析。不過

此 router 因考量縝密，繞線後的結果是三種中最好的！

四、 Implementation

A. L-route

i. Data Structure

在我所實作的 stupid-route 中，為降低複雜度，我採用「先尋找核心點，之後考慮所有人與核心點的兩種 L 繞法擇一採用(即上述 O(mn)的演算法)」實作。在實做中，我以亂數方式找尋中心點，再以亂數方式決定其他點與中心點的 L 連線方式(1/2 機會)，如果其他點與核心點的連線不是直線。

在本演算法的實作中，首先必須要由 input file 讀入線段資訊並記錄，之後再將繞線的結果保存下來，並 dump 到 output file 中。使用到的資料結構如下：

+用來存取線段資訊的資料結構

```
struct net{
    char net_name[100];
    int net_id;
    int net_pins_number;
    PIN_LOCATION *net_pins; //用來記錄這條 net 有哪些 pin
    int route_number; //紀錄共有幾組繞線結果
    ROUTE *routePtr;//用來記錄繞線結果
};
```

+用來存取 net 上有哪些 pin 的資料結構

```
struct pin_location{
    int     pin_x;
    int     pin_y;
};
```

由於可由 input 檔案中線段資訊知道有幾組 pins，故我在實做上採用動態 array 方式處理，不用 link-list 實作，以節省搜尋時間

+用來存放繞線結果的資料結構

```
struct route{
    int start_x;
    int start_y;
```

int end_x;

int end_y;

*ROUTE *nextPtr;*

};

因繞線結果無法事前知道有幾組，故以 link-list 方式實作

ii. Flow

程式執行流程如下：

+int main(int argc,char *argv[])

+void parser(char *input_file_name)//讀入 inputfile

+NET *create_net() //建立 Net 資料結構

+void insert_net_value(int count,char *net_name,int net_id,int net_pin_number)//填入 Net 資料結構

+void insert_pin_location(int counter,int big_array_counter,int temp_loc_x,int temp_loc_y)//填入 pin 資料結構

+void stupid_route(NET *net_arrayPtr)//實行繞線

+int select_center(NET *net_arrayPtr)//選出中心

+void point_to_point_route(NET *net_arrayPtr,int center_index)//各點對中心繞線

+void dump_output_file(char *output_file_name)//處理輸出檔

B. Maze route

i. Data Structure

在 Maze-route 的實作中，我以符合 maze-route 的基本精神－尋找最小數字的路徑反向推回方式實作，在實做完單純的 maze route 後，先跑了一組實驗數據，之後再加上針對本作業規格的最佳化－找 density 高的路徑走與找非 hotspot 的路徑走。經由最佳化後所得到的結果會比無最佳化者好一些。

在資料結構的實作上，除了前述 stupid-route 所用的資料結構仍會用到外，另須新建資料結構存取 density 與 hotspot 相關資訊，亦須再建構 maze-route 填表與 trace back 所需要的資料結構。相關資料結構詳列如下：

+用來存取 density 與 hotspot 相關資訊之資料結構

struct grid{

int loc_x;//這格的 x 位置

int loc_y;這格的 y 位置

```
    int cap_H;//這格右邊的 capacity(即 density)
    int cap_V;//這格下面的 capacity(即 density)
    bool is_hotspot;
  };
```

在我的實作中，capacity 的記錄，cap_H 是記錄該 grid 的右方，cap_V 是記錄該 grid 的下方。之所以只記錄下方與右方的原因是，上方與左方自然會有上面與左邊的格子記錄，以用來區分四個不同的方向。

+用來實作 maze route 的資料結構

```
  struct grid_route{
    int loc_x;
    int loc_y;
    bool is_terminal; //紀錄已經繞完的
    bool is_route;   //記錄這個點是否繞過
    int route_number; //填表用
    int cap_H;
    int cap_V;
    bool is_hotspot;
  };
```

+用來記錄 maze route 進度的資料結構

```
  struct maze_queue{
    int loc_x;
    int loc_y;
    int counter;//紀錄以這個格子長出去的人,要填什麼數字
    MAZE_QUEUE *nextPtr;
  };
```

這其實是一個 Queue，一但一個格子填了上下左右的數值，會同時把上下左右加到 Queue 中，以便再繼續往下填。

ii. Flow

程式執行流程如下

```
+int main(int argc,char *argv[]) //主程式
    +void parser(char *input_file_name)//讀入 inputfile
        +NET *create_net() //建立 Net 資料結構
        +void insert_net_value(int count,char *net_name,int
          net_id,int net_pin_number)//填入 Net 資料結構
```

+void insert_pin_location(int counter,int big_array_counter,int temp_loc_x,int temp_loc_y)//填入 pin 資料結構

+void maze_route(NET *net_arrayPtr)//針對多點下去 route

+GRID_ROUTE **create_grid_route_struction()//創造記錄資料結構

+void do_maze(int loc_x,int loc_y,GRID_ROUTE **grid_routePtr, NET *net_arrayPtr)//一點一點下去做

+MAZE_QUEUE *file_table(int loc_x, int loc_y, bool is_beginning,int number ,GRID_ROUTE **grid_routePtr,MAZE_QUEUE *maze_queue_tailPtr,NET *net_arrayPtr)//填上下左右的位置

+void dump_result(int loc_x, int loc_y, NET *net_arrayPtr,GRID_ROUTE **grid_routePtr)//trace back recursively

C. NTHU-route

i. What did I do

針對 NTHU-route，因原始碼繁多，在此我只敘述我的貢獻。

首先，我更改了 ISPD2008-NTHU-R-CodeRelease-Updated/src/router/route.cpp 檔案，更改的部分如下：

```
if(ap.caseType() == 0){
        //IBM Cases
        Layer_assignment(ap.output());
        clock_t t4 = clock();
        printf("time: %.2f %.2f\n",(double)(t4-t3)/CLOCKS_PER_SEC,(double)(t4-t0)/CLOCKS_PER_SEC);
……
```

第二，我更改了 Makefile，更改部分如下：

```
DebugOptions = -O3 -DNDEBUG -static -funroll-loops -march=nocona -finline-fun ctions -DIBM_CASE
```

第三，我重寫了 output file 轉換器，讓 output file 能夠與驗證程式相容，轉換器程式為 output_parser.c

五、 Experimental Result

A. Platform and Programming Language

本次的作業要求以 C 或 C++實作，針對 stupid-route 與 maze-route，我都已完全的 C 語言實作。而 NTHU-route 為學長的作品，採用 C++語言實作，而我針對 I/O 部分做格式處理部分，用 C 語言實作。

所有的程式開發、編譯、執行與實驗數據都是在 nthucad 工作站群中的 ic21 完成，ic21 的資訊如下：

CentOS release 4.4 (Final)

Kernel 2.6.9-42.ELsmp on an x86_64

Platform = amd64

Platform = LINUX

B. Output Result

在 Output Result 的分析中，我以四種結果加以比較分析：

1. stupid-route
2. maze-route-without-opt (即不考慮 overflow 與 hotspot)
3. maze-route-with-opt (即考慮 overflow 與 hotspot)
4. NTHU-route

以下呈現四種不同方法，針對五個 case 的數據：

[[[[[stupid-route]]]]]

stupid_route_result			
case	overflow	# nets pass hotspots	wirelength
ibm01	51388	4555	134703
ibm02	168948	13797	411401
ibm03	119997	11359	328685
ibm04	96750	12858	316902
ibm05	397444	36715	1134746

[[[[[maze-route-without-opt]]]]]

maze_route_result_no_opt			
case	overflow	# nets pass hotspots	wirelength
ibm01	5173	3689	68697
ibm02	11504	10118	186395
ibm03	7379	8413	161934
ibm04	10481	10292	179578
ibm05	2312	23572	454727

[[[[[maze-route-with-opt]]]]]

maze_route_result_with_opt			
case	overflow	# nets pass hotspots	wirelength
ibm01	4148	3626	68998
ibm02	10084	10121	186812
ibm03	5632	8459	162233
ibm04	7996	10146	180147
ibm05	1872	23631	455114

[[[[[NTHU-route]]]]]

nthu_route_result_with_opt			
case	overflow	# nets pass hotspots	wirelength
ibm01	0	3510	63089
ibm02	0	9529	171126
ibm03	0	7849	146812
ibm04	2	9555	168199
ibm05	0	22001	410033

由上數據可以明顯看出 NTHU-route 對於 overflow 處理得非常好，幾乎都壓到 0。

C. CPU Time

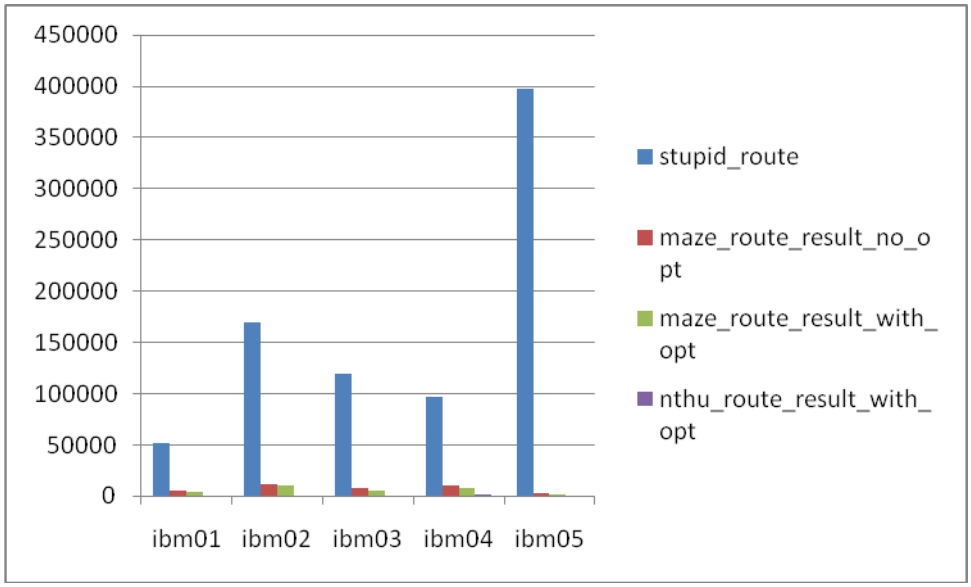
針對不同的 Algorithm，CPU 執行的時間有差。Stupid-router 的執

行時間<1sec，maze-router 約 6~10sec，NTHU-route 則需要約 30 秒的時間(韓 output 檔案處理)。大致上簡單的演算法跑起來較快，複雜的演算法相對要較久的時間。

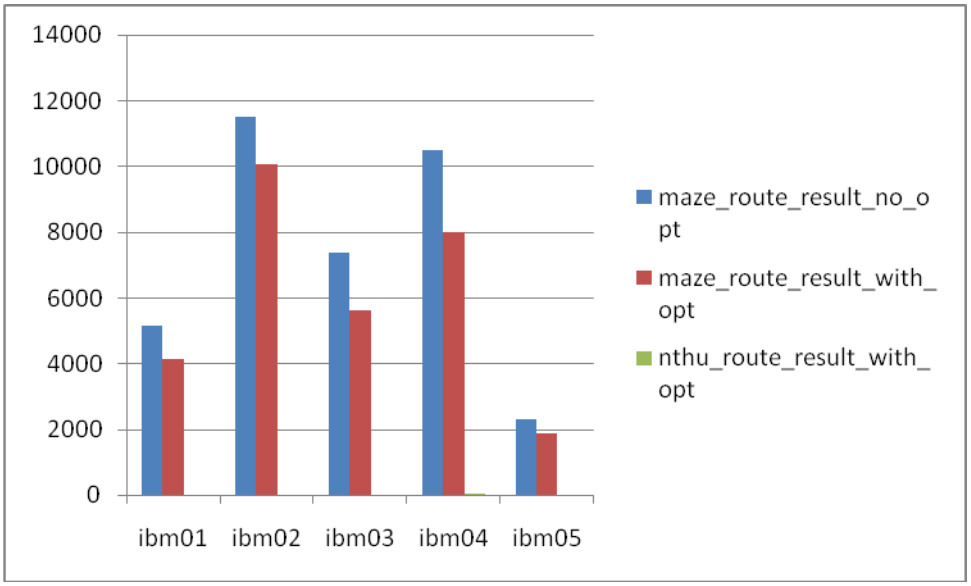
D. Data Analysis

以下將針對上面的數據作圖分析，在不同的實作方法下，針對不同需要最佳化的條件逐一分析之。

首先探討本作業最重要的最佳化目標：overflow，下圖是針對不同的 case 與不同的實作方法下，overflow 之情形

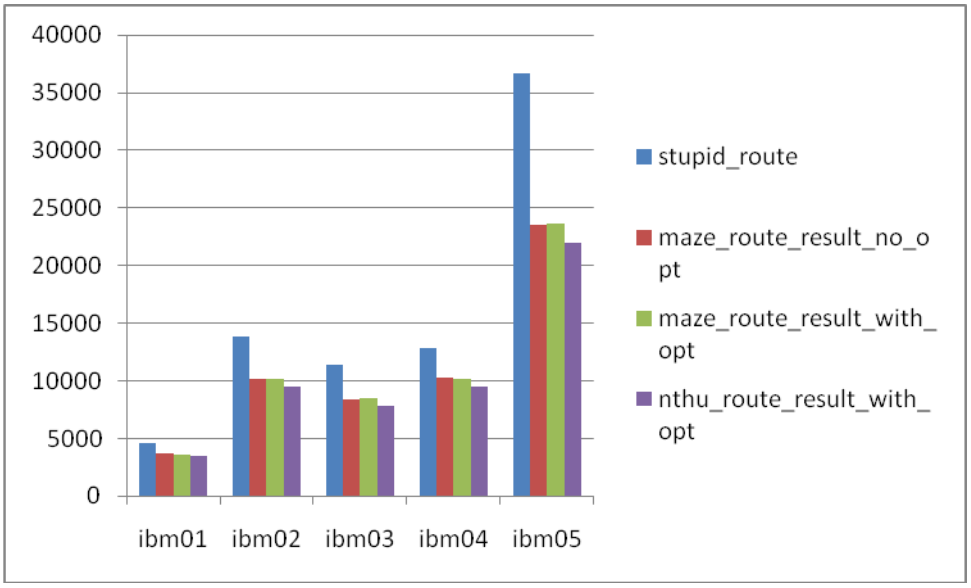


由上圖，我們明顯的看出 stupid-route 的 overflow 遠大於其他 router，最大甚至高達 400000 倍，原因是因為 stupid-route 並沒有做 overflow 的最佳化。由此可以看出，一但 overflow 最佳化條件被考慮，加上放寬 solution space，就有很大的進步。我們就其他 router 的結果再作圖來看



由上圖，我們發現對 maze_route 做考慮 overflow 的最佳化之後的結果，都比之前要來的好，而且有一定程度的進步。但與 nthu-route 比起來，仍然遜色很多，最壞情況大約是 10000 倍。

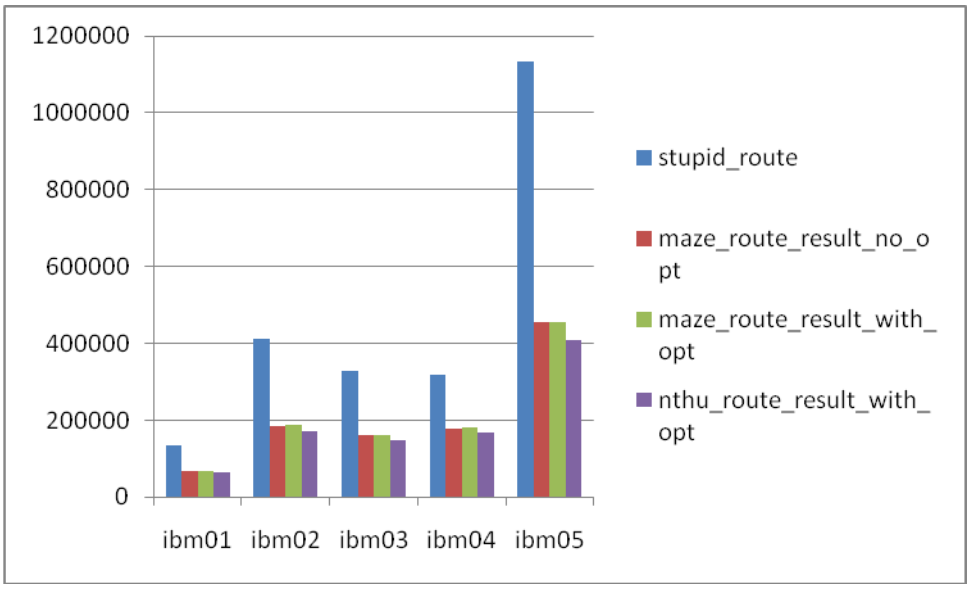
其次，我們所關心的是穿越 hotspot 的 net 數量。下圖是針對不同的 case 與不同的實作方法下，穿越 hotspot 的 net 數量之情形



由上圖很明顯的看出，沒做最佳化的 stupid-route 擁有最差的結果。一但採用 maze-route，就使結果進步約 30%。但就其他三種 router 而言，maze-router 做最佳化與否的差距似乎就沒有 overflow 那麼明顯，可能原因是 hotspot 的最佳化是在 overflow 之後才做的，很有可能為了得到較好的 overflow 值而犧牲 hotspot 最佳化。另一個值得注意的現象是，

NTHU-route 的數值與 maze-route 的數值差距不大，可能的原因一是兩者對 hotspot 處理方式雷同，二是為了得到較好的 overflow 值而犧牲 hotspot 最佳化。

最後，再就 wire length 探討。下圖是針對不同的 case 與不同的實作方法下，total wire length 之比較



由上圖，沒做最佳化的 stupid-route 之結果約是其他 router 的 2 倍~4 倍。Wire length 的結果與 hotspot 的結果類似。

由上面一系列的分析我們可以發現，NTHU-route 是盡力把 overflow 壓到最小，但其他的要求則都有被犧牲或沒做的情形。

E. Conclusion

藉由以上的數據與圖表分析，我們發現 L-route 與 maze-rout 的結果有一定的差距存在，而有考慮最佳化與否也會有所差距。若比較 overflow 數量， NTHU-route 大勝，幾乎把 overflow 都壓到 0；若比較穿越 hotspot 的 net 數量，NTHU-route 雖然以些微差距得到較好的結果，但差距就不若 overflow 那麼大；而比較 wire length 的結果，仍然是以 NTHU-route 險勝。

六、 Reference

1. CS6135 VLSI Physical Design Automation 課程講義與網頁
2. C ,How To Program (Textbook for programming design)

七、 Appendix (How to execute my codes)

A. Stupid-route

%gcc ukroute.h ukroute.c –lm –o stupid_route_random.exe

%./ stupid_route_random.exe input.in output.out

B. Maze-route

% gcc ukroute.h ukroute.c –lm –o maze_route_v2.exe

%./ maze_route_v2.exe input.in output.out

C. NTHU-route

%cd ISPD2008-NTHU-R-CodeRelease-Updated/src

%make clean

%make

%cd ../ bin/runibm0X X=要執行的 case

註 0：Output file 為 ibm0X_real_result.out

註 1：runibm01 代表第一個 case runibm02 代表第二個 case 以此類推

註 2:若要執行隱藏 case,請先將該 case 的.txt.gz 以及.txt 檔複製到此目錄下,並請開啟 runibm00,將下列紅字處換為要跑的檔案,再執行 runibm00 即可

./route --input=i1 --output=tmp.out --p2-max-iteration=5

--p3-max-iteration=5

--overflow-threshold=0

./output_change.exe ibm01.modified.txt tmp.out ibm01_real_result.out

rm tmp.out