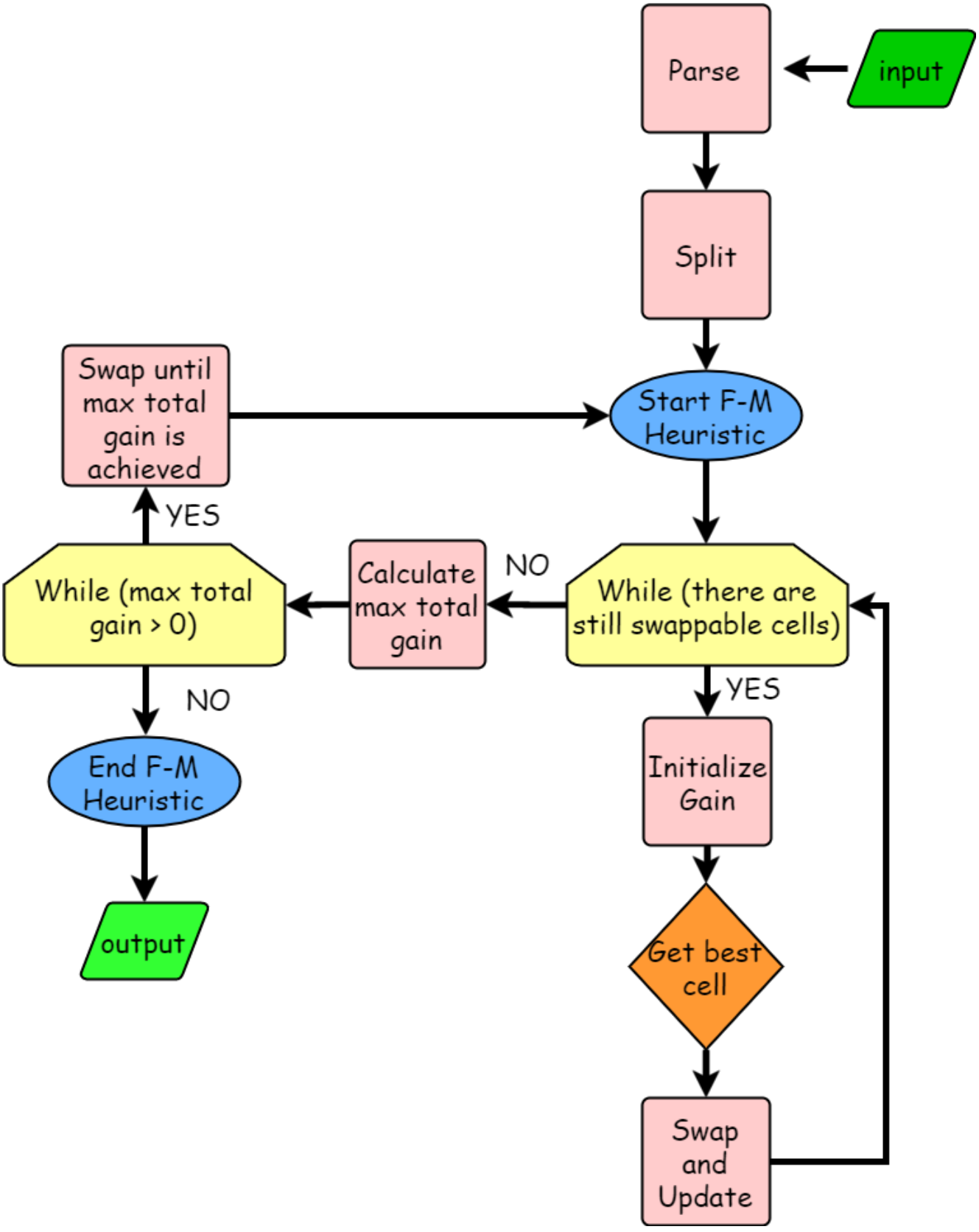
Physical Design for Nanometers ICs, PA 1

B03901056, 電機三, 孫凡耕

(1) Algorithm flow:



(2) Pseudo code:

Input : a netlist file with unit weight cells.

Output: the best found 2-way split that minimized the cutsize.

Begin:

```
Parse
Initial Split
Do:
    Gains ← ∅
    Cells ←∅
    While (there are still swappable cells):
        Initialize gain
        Max_Gain ←value of max gain pointer
        Gains.push_back( Max_Gain )
        C ← the cell with the maximum gain
        Cells.push_back( C )
        Swap C
        Update gain according to C
    Max_Total_Gain ← Find_Max_Prefix_Sum( Gains )
    Total_Gain ← Sum( Gains ) //Total Gain should always be equal to zero.
    While (Total_Gain < Max_Total_Gain):
        Total_Gain += Gains.back()
        Gains.pop_back()
        Restore_Cell( Cell.back()] )
        Cells.pop_back()
While (Max_Total_Gain > 0)
```

End

P.S.: For programs that run through multiple iterations, go through the above process several times and output the best result.

(3) Detailed description:

Parsing →

Since the output file doesn't need to remember the name of the nets, my program replace all nets' name with an integer id counting from 0. My program assumes that all descriptions of cell are in the format of "'c'+integer_id". After reading the input file the first time, my program will record the largest cell id and open a vector. Then I will read the input file once again and put all cell in the corresponding slot of the vector.

Also, I would remove duplicated edge when a net connects to the same cell twice or more.

Initial Split →

I wrote four kinds of initial splitting method:

i. According to the cell_id.

ii. Sort the order of the cell according to the number of the pins on it.

iii. Sort the order of the cell according to f(x), where f(x) = max(net_sizes) + alpha*average(net_sizes). The net_sizes refers to the array of the size of all the nets on this particular cell, where the size of a net is defined as the total number of connected cells. Alpha is a tunable hyper-parameter. For my multi_xxx_fm program, it will try alpha over from 0 to 8.

iv. Split randomly.

Gain Initialization →

The same as the original F-M Heuristic but just simply ignores the effect of a single-size net.

Cell Swapping and Gain Updating →

There is a max pointer that hold the maximum gain value. Whenever a cell's gain is updated, the program will check if the updated gain exceeds the value of the max pointer or not, and if yes, just replace the value.

Also, after swapping all the available cells and calculated the value of the maximum total gain, I will perform restore operations to change the state of the cells until the maximum total gain is reached backwardly. This may cause different behavior if this step is done forwardly.

The rest are the same as the original F-M Heuristic but just simply ignores the effect of a single-size net.

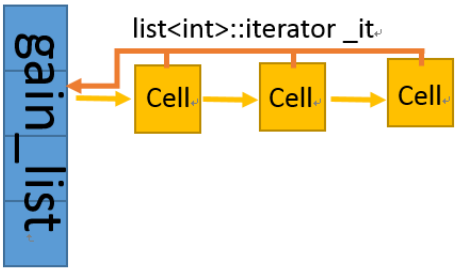
(4) Data Structure:

```
class Net {
private:
  int _Acnt;
  vector<int> _cells;
};
```

```
class Cell {
private:
  int _gain;
  bool _inA, _isFree;
  vector<int> _nets;
  list<int>::iterator _it;
};
```

A Net class stores which cells belong in this cell by the cell's id. There is also an integer variable _Acnt which keeps track of the quantity of the cells in group A. The quantity of cells in group B can be obtained by ((int)_cell.size()-_Acnt).

A Cell class stores which nets does this cell connects to by the net's id. It also remembers its state by the variable _inA (true if in group A else false), _isFree (is unlocked or not), _gain (current gain value). The iterator _it just speeds up the algorithm by constant time removal from the gain_list when this cell's gain is changed, as shown below.



(5) Issues with input_0.dat :

The input_0.dat is very different from all the other input_x.dat. I print out some information about the netlist which definitely show the difference. The information is shown as below:

For input_0.dat →

Net with minimum number of connected cell: 1 (which is quite awkward)
Net with maximum number of connected cells: 30506 (which is very very very large)
Average number of connected cells: 4.3
Cell with minimum number of pins: 2
Cell with maximum number of pins: 31
Average number of pins: 4.8

For input_[1~5].dat →

Net with minimum number of connected cell: 2 (which is reasonable)
Net with maximum number of connected cells: 3 (input_[1, 2].dat), 4 (input_[3, 4, 5].dat)
Average number of connected cells: 2.5 (input_[2, 3].dat), 3(input_[4, 5].dat)
Cell with minimum number of pins: 3 (input_[2, 4, 5].dat), 4 (input_[1, 3].dat)
Cell with maximum number of pins: 4 (input_[2, 4, 5].dat), 5 (input_[1, 3].dat)
Average number of pins: 3.3 ~ 4.2

After examining the information above, it is obvious that input_0.dat has some strange characteristic.

i. There exists nets that connect to only a single cell.
ii. The maximum number of connected cells within a net is only 3 for input_[1~5].dat, but this number is more than 10000 times for input_0.dat.
iii. The situation in (ii.) makes the average number of connected cells within a net for input_0.dat is almost twice as high as for input_[1~5].dat.
iv. The gap between maximum and minimum number of pins on a cell is precisely 1 for input_[1~5].dat, but is 29 for input_0.dat.
v. The situation in (iv.) makes the average number of pins on a cell for input_0.dat is apparently higher that as input_[1~5].dat.

Not only that, but input_0.dat has duplicate connections: a net may connect to a cell more than once. If the F-M Heuristic don't take care of this and remove these duplicate connections, there will be mismatch between the gain calculated and the actual cutsize reduced.

All of these extraordinary attributes of input_0.dat makes it special. Usually, the four different kind of initial split method I mentioned in (3) performs about the same (within 10%). But for input_0.dat, random generated split usually obtains a cutsize of about 10000, but the (ii.) split method gets a staggering cutsize of 1949. However, with the (iii.) split method, my program can reach a jaw-dropping cutsize of 832.

(6) Improvement :

I wants to improvement the quality of my solution, so I searched online, and then I saw this particular paper: [http://web.eecs.umich.edu/~imarkov/pubs/jour/j004.pdf]. I made two big difference according to this paper.

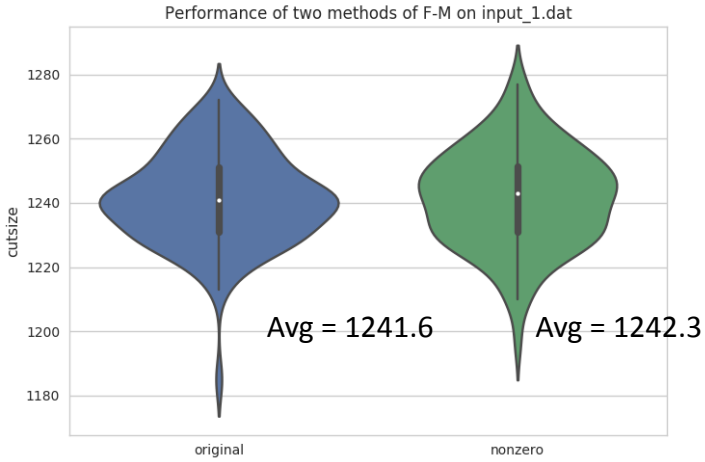
LIFO vs. FIFO →

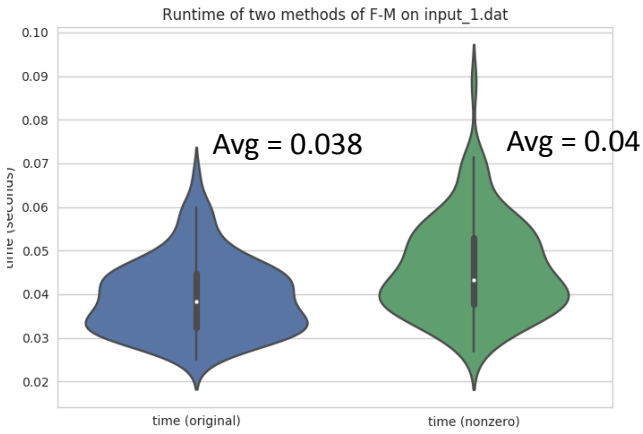
The paper suggests that Last-In-First-Out (LIFO) is better than First-In-First-Out (FIFO). I tried it

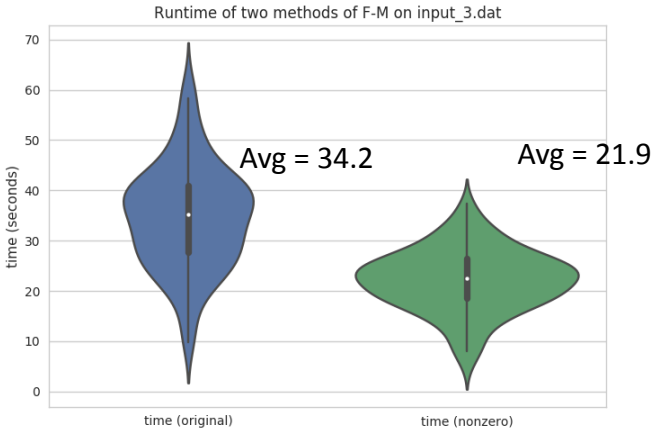
and it did worked.

All-delta update vs. nonzero update →

Originally, in every swap of a cell, the gain is updated immediately, and the updated cell will be moved to a new gain bucket concurrently. However, there is possibility that the total gain difference is zero because the base cell influences the other cell with more than one net and may results in zero changes. In this case, the original program will change the position of the other cell in that same gain bucket since it is moved out and then moved in. In my new nonzero-version program, the other cell will just do nothing. I used a global counter with a global tracker and a local list to perform the operations above in constant time, thus the total complexity is the same. In the paper, nonzero out performs the original one by a large margin taking a slightly shorter time span. I also experimented and visualize the result by python myself to check out if this is true or not.







I run the algorithm 100 times and then plotted the results as above for input_1.dat and input_3.dat. We can see that the difference of cutsize is not as big as the paper shown. However, the nonzero idea did make a difference on the runtime. When the size of the netlist is small, nonzero spends about the same time. But when the total number of cells increased, the runtime significantly reduced as we can see since input_3.dat has 66666 cells and input_1.dat has only 3000 cells. Since we used to pay attention for larger netlist, I would prefer nonzero over the original (all-delta) one.