EDA HW2

江若綾 113062640

(1) How to compile and execute program

Compile program:

Step 1: Enter "HW2/src/"

Step 2: Enter command: $ make

Remove compile program:

Step 1: Enter "HW2/src/"

Step 2: Enter command: $ make clean

Execute program:

Step 1: Enter "HW2/bin/"

Step 2: Enter command: $ ./hw2 <.txt file> <.out file>

Execution example: $ ./hw2 ../testcase/public1.txt ../output/public1.out

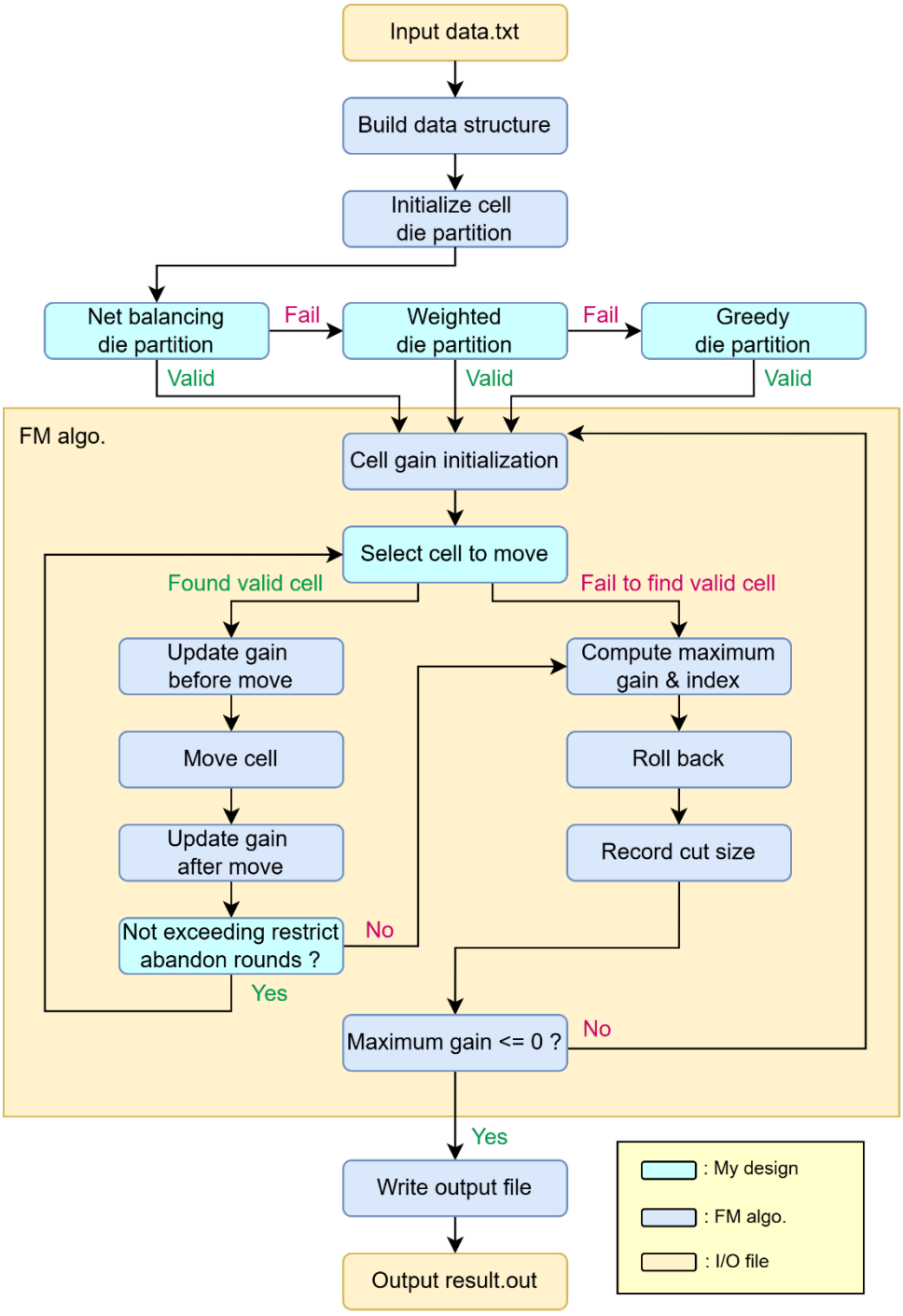
(2) Final cut size and runtime of each testcase

```
checking item          | status
-----------------------|--------
correct tar.gz         | yes
correct file structure | yes
have README            | yes
have Makefile          | yes
correct make clean     | yes
correct make           | yes

 testcase |   cut size |    runtime | status
----------|------------|------------|--------
   public1 |       1216 |       0.23 | success
   public2 |        102 |       0.26 | success
   public3 |      32399 |       2.86 | success
   public4 |     138558 |       2.46 | success
   public5 |     265160 |      39.42 | success
   public6 |     196456 |      36.98 | success
+-----------------------------------------------+
|                                               |
|   Successfully write grades to HW2_grade.csv  |
|                                               |
+-----------------------------------------------+
```

Testcase	public1	public2	public3	public4	public5	public6
Cut size	1216	102	32399	138558	265160	196456
Run time	0.23	0.26	2.86	2.46	39.42	36.98

(3) Algorithm



My design is similar to **Fiduccia-Mattheyses (FM) algorithm**[1] but with some changes. I modify some parts of the algorithm to speed up the execution time and make the final cut size become lower.

[1] Fiduccia, C., & Mattheyses, R. (1982). A Linear-Time Heuristic for Improving Network Partitions. In 19th Design Automation Conference (pp. 175-181).

Modification to minimize the cut size:

The final cut size is heavily dependent on the quality of the initial die partition. Due to this, I implemented three different methods, and out of all three, the best way to initialize die partition is the method called **net balancing**.

(1) Net balancing

In which is a heuristic technique commonly used in VLSI partitioning to reduce initial cut size by assigning all cells connected to the same net into the same partition. But there are some catches, because of the area utilization constraints, these methods may not always work. Thus, I prepared the other two initialized methods just in case.

(2) Weighted

In which I sort the cells based on their area differences when placed on DieA compared to DieB. In other words, I want to assign the die to the cell that has bigger area differences first. Then I assign the cell to the die which the cell has a smaller area in it. This method successfully initializes all the public testcases' die partition, but I still implement the last initialized method to be safe.

(3) Greedy

In which it assigns the die to the cell based on the ascending order of the area of DieA. This method perfectly assigns die to all the cells but can't land on a good cut size compared to the other two methods.

Modification to speed up the execution time:

(1) Selecting cell to move (remove cell in bucket list)

When selecting which cell to move, this process will become faster along with the rounds go by. This is because I rebuild the bucket-list every time a pass starts. In this way, I can simply remove the cell in the bucket-list when the cell is picked up to move. Thus, reducing the time a cell is being constantly checked but not chosen. This method can help speed up the process of a pass.

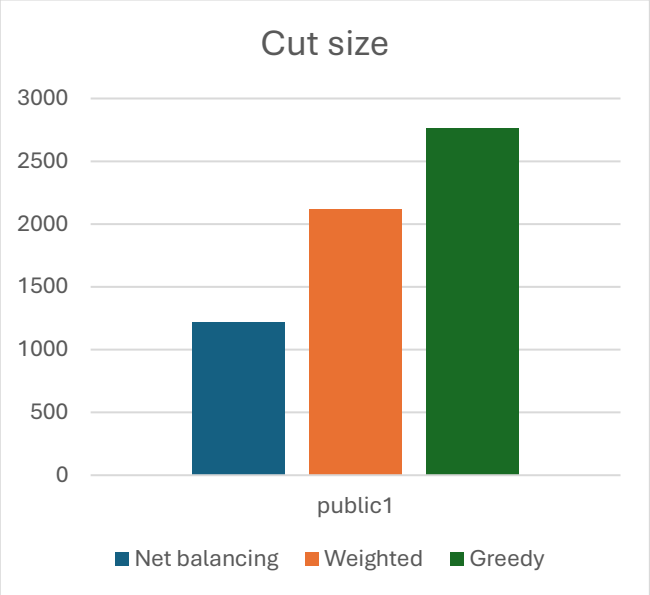
(2) Restrict abandon rounds

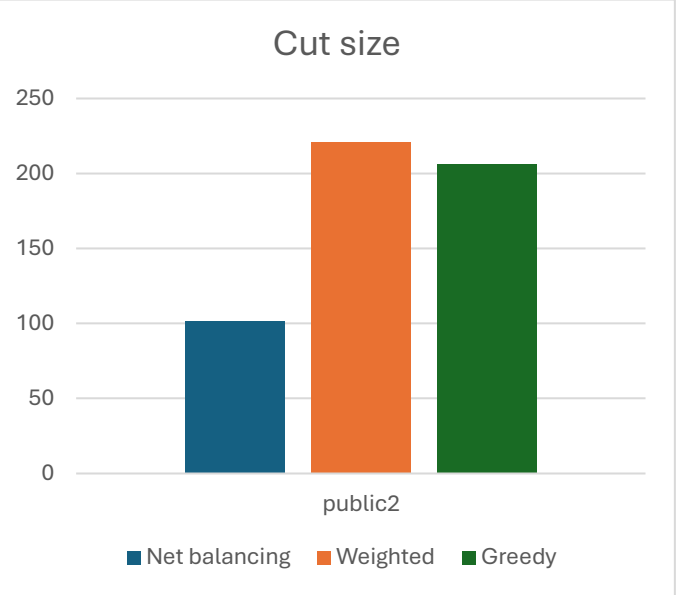
During my implementation, I've noticed that the abandoned rounds of a pass are significantly more if the number of cells becomes bigger. In which makes the time of a pass unbearably longer. To restrict the unused rounds of a pass, I add a feature to control the maximum abandoned rounds of a pass. If a pass has a consecutive downfall that reaches the same number of the restricted maximum abandoned rounds, then the pass will be forced to stop and enter the next new pass. In this way, the abandoned rounds will be forced to lower down which speeds up the time without sacrificing much quality (cut size).

(4) Techniques to improve solution's quality

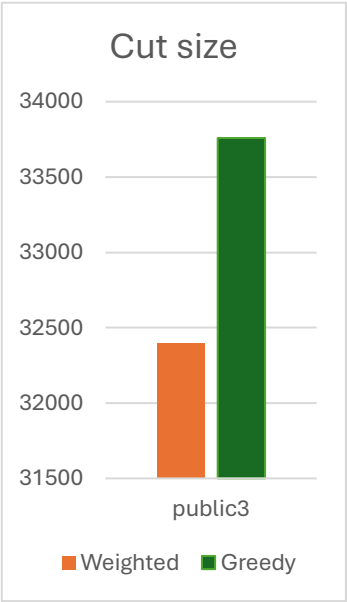
The initial die partition deeply affects the final cut size. I implemented three methods in which to make my solution quality better.

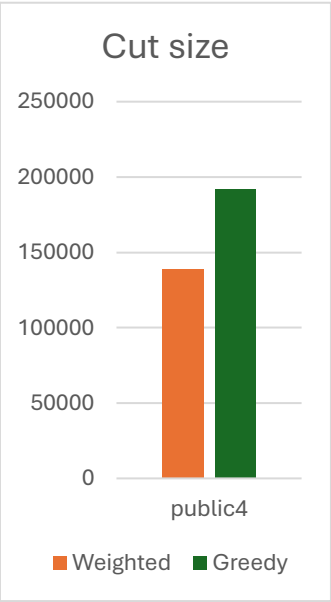
Techniques to improve cut size:

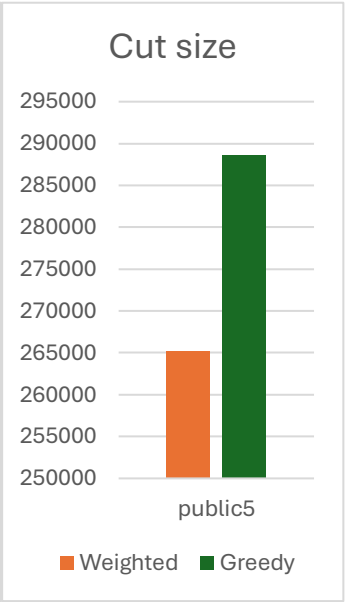


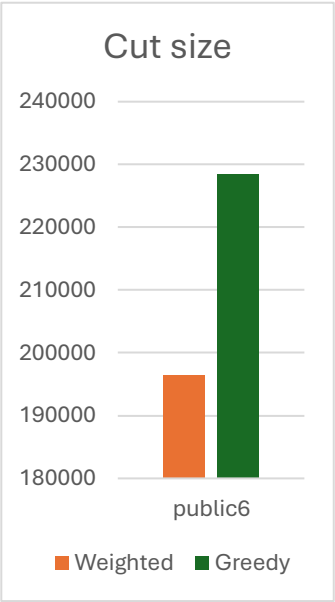


Because net balancing may not work in certain test cases, the following graph only shows the comparison between weighted and greedy die initialization.



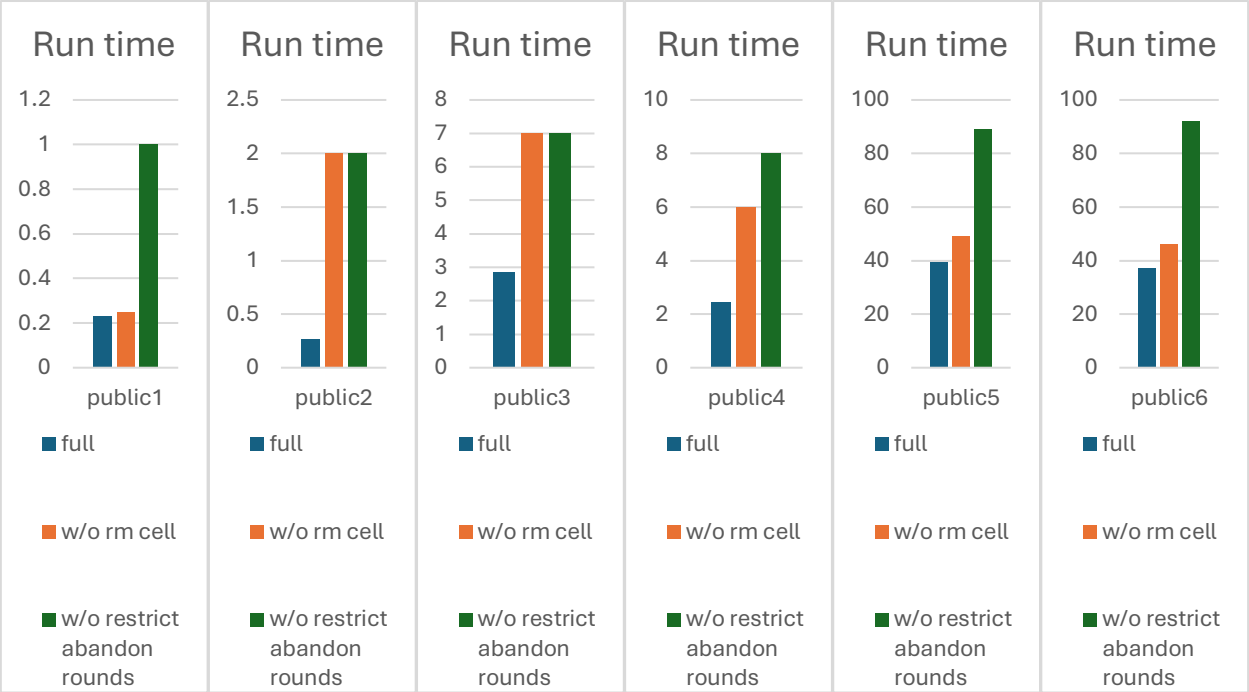






The above graphs show that the method applied into the algorithm can significantly improve the solution quality. The reason why net balancing works so well is because this method promotes locality by grouping strongly connected cells together early, which can lead to better partitioning convergence in FM algorithm.

Techniques to improve run time:



After both techniques being applied onto the algorithm, not only did the run time become significantly lower, but the cut size also performed better. The reasons why these techniques can speed up the execution time are elaborated in the previous section. The possible reason why stopping early can actually lower the cut size might due to the implementation of my design, I see the consecutive down fall of partial gains as abandon rounds (which in some way it is), force the pass to find the maximum partial gains no matter how many rounds being abandoned could possibly lead the algorithms harder to find better movement in the next pass. It might be a good idea to move to the next pass before it abandons too many rounds. And the results support my idea as well.

(5) Learned & encountered problems

Learned:

The FM algorithm offers many possible implementation approaches, and I found the process of implementing it quite challenging. Through experimenting with different methods, I realized the significant impact of the initial state. However, to be honest, I probably learned even more about how to build functional and easily expandable code. To optimize the algorithm, it's crucial for the code to be modular, readable, and easy to modify. I put a lot of effort into keeping my code as clean as possible, and that has greatly helped during the modification process.

Encounter problems:

The FM algorithms show concepts but not a lot of implementation details. During the implementation, the number one thing I faced is how do I build the data structure? How can I make the execution time below the required time limits but also make my code look clean and easy to expand? Before any

accelerations, the execution time is unbearably high, that also took me some time to fix.

The full structure of the FM algorithm is quite big, I change my data structure constantly during implementation, which makes some errors pop out. And that also took me some time to fix.