CS6135 VLSI Physical Design Automation
Homework 2: Two-way Min-cut Partitioning

111062625 蔡哲平

1. **How to compile and execute my program.**
   - Compile: Enter *src/* and make, it'll generate the executable file to *bin/*.
     $ cd src
     $ make
   - Execute
     $ ./bin/hw2 [CELL_FILE] [NET_FILE] [OUT_FILE]
     e.g.
     $ ./bin/hw2 ../testcases/p2-1.cells ../testcases/p2-1.nets ../output/p2-1.out

2. **The final cut size and the runtime of each test case.**

	p2-1	p2-2	p2-3	p2-4	p2-5
cut size	225	2221	22653	80473	161653
runtime	0.07	1.01	105.76	291.12	292.36

3. **Time profile of each test case.**

	p2-1	p2-2	p2-3	p2-4	p2-5
I/O	0.005	0.039	0.454	1.231	2.590
Computation	0.014	0.946	104.902	289.319	288.767

4. **The details of your implementation containing explanations of the following questions:**

   I. **Where is the difference between your algorithm and FM Algorithm described in class? Are they the same?**

   No, they are slightly different. In the FM algorithm described in class, *update_gain* updates only the free cells' gains. However, in my implementation, it also updates the locked cells' gains because I've found that when restoring the result after the maximum partial sum is determined, it also needs the correct info such as the gain of the locked cells to restore correctly. More information about my implementation of *restore_result* will be discussed in III. Moreover, because there is a time constraint in every test case and the runtime won't be graded in this assignment, I limited the runtime to 290 seconds to make sure it won't TLE and used as much time as possible to reduce the cut size, which is different from the FM algorithm that keeps running until the cut size converges.

   II. **Did you implement the bucket list data structure?**

Yes, there is a bucket list for each set, which is 2 in sum. It's similar to the one described in class, however, I used a *std::vector* instead of a doubly linked list and a *std::unordered_map* instead of an array recording cells' positions in the bucket list, which will increase the time complexity of finding cells' positions to O(logN) but reducing some coding overhead. As a C++ class, there are some other operations in the bucket list data structure such as *insert_cell*, *remove_cell*, *update_cell*, and *get_top_kth_cell* to maintain it.

III. How did you find the maximum partial sum and restore the result?

I've implemented a function called *select_base_cell* to return the valid cell with max gain to move to the other set. It'll push the selected cell to an array called *selectedBaseCells* and push its gain to an array called *maxGains*. Once *select_base_cell* returns a null pointer, it means that there are no cells available, and hence starts calculating the maximum partial sum with a function going through *maxGains* to find the maximum partial sum with O(N) time complexity. After the maximum partial sum is found, the function will also record the found index of *selectedBaseCells*, denoted as *maxPartialSumIndex*. This way, we'll know where to restore the result. Finally, the way I restore is to keep choosing the element of *selectedBaseCells* from the rear as the new base cell and move it back to its original set and update cells gain after the movement until reaching *maxPartialSumIndex*.

IV. What else did you do to enhance your solution quality (you are required to implement at least one method to enhance your solution quality) and to speed up your program?

After some experiments and paper research, I've found that the final cut size your program can get is determined by the initial partition, as well as many other local search heuristic algorithms. However, the algorithm for finding the initial partition in the paper is too difficult to implement, as a result, I used the try-and-error method to try to come up with the best initial partition for the provided test cases. After tons of experiments, the best initial partition method I came up with is in follow:

- Push all cells to set A.
- Sort cells by their size B.
- Keeps pushing cells with the smallest size B to set B until it's balanced.
- After balanced, keeps pushing cells with the smallest size B to set B until it's unbalanced.

V. If you implement parallelization (for FM algorithm itself), please

describe the implementation details and provide some experimental results.

No, I didn't implement parallelization.

5. **What have you learned from this homework? What problem(s) have you encountered in this homework?**

In this assignment, I've learned how to implement a complex algorithm and deeply felt the importance of object-oriented programming. Although all the needed algorithms' pseudo codes are written in the PPT, it was still challenging to transform them into actual code. The most stressful problem I've encountered is that I followed the *update_gain* pseudo code in the PPT but it caused incorrect results in my implementation. Because the last thing you will suspect while debugging is the original FM algorithm, it took me two full days to find that I should also update the locked cells' gains. In a nutshell, it was a quite difficult assignment but it sure did improve my coding skill.