Student Name: LUO QIN
Student ID: 1155184128
Department: Computer Science & Engineering

CENG 5270: EDA for Physical Design of Digital Systems

Homework 1

1. **Principles**

The overall implementation is based on the Fiduccia and Mattheyses (FM) algorithm. The algorithm is aimed at moving single node with the largest cut size decrease each time and getting the largest maximum partial sum of the cell movements. That is to find a proper cell movement sequence to maximize the partial sum G,

$$G = \sum_{i=1}^{m} g_m \tag{1}$$

where g_m is the gain (cut decrease) when moving the node i to the other partition. In addition, the results of the partitioning should satisfy the balance criteria. Suppose the sum of all cell sizes in A and B are notated as $area(A)$ and $area(B)$, then the balance condition are described as:

$$|area(A) - area(B)| \leq \frac{n}{10} \tag{2}$$

where $n = area(A) + area(B)$. Note that the balance condition is considered only when comparing different partial sum of the cell movements.

The implementation of the FM algorithm is similar as that in the slides. In the implementation, the bucket list data structure is used to obtain the nodes with the largest gain, and only the critical nets are considered when updating the gain for all the free nodes.

- **Bucket Set**

  In each iteration in FM algorithm, we need to find the node with largest gain, and update the gain for each node. Bucket Set is an efficient data structure to use because we don't need to traverse all the nodes when getting the largest gain. In my implementation, the bucket list is structured as a vector with the *Move* object defined as:

```
class Move
{
    public:
        Move(int _gain): gain(_gain) {}
        void addCandidateNodes(shared_ptr<NodeCluster> node) {candidateNodes.insert(node);}
        void deleteCandidateNodes(shared_ptr<NodeCluster> node) {candidateNodes.erase(node);}
        int NodeSetSize() {return candidateNodes.size();}
        set<shared_ptr<NodeCluster>> candidateNodes;
        int gain;
};
vector<Move> MoveCol;
```

where two critical elements in the class are *gain* and *candidateNodes*. *gain* denotes the calculated gain value when moving the node, and *candidateNodes* contain all the nodes with the same gain value. We use data structure *set* to represent the sets of the nodes with the same gain value, because we can find, add and delete the nodes in $\mathcal{O}(1)$. Suppose the maximum number of pins for all the cells is denoted as P_{max}, the size of the $MoveCol$ vector is equal to $2P_{max} + 1$.

- **Gain Update**

  In the gain update, we only consider the critical nets connecting to the cells. A net is considered critical when the number of pins in one partition is smaller than 1. In my implementation, *partitionPin* represents the pins in each partition for the net, and *partitionPinCnt* represents the number of the pins in each partition.

```
        vector<int> partitionPinCnt;
        vector<set<shared_ptr<NodeCluster>>> partitionPin;
```

  When the nodes are moved across the partitions, the *partitionPinCnt* and the *partitionPin* would update.

- **Maximum Partial Sum**

  In each movement, we would sum the gain of the cell movements the Partial Sum, and then compare them with the current maximum of the partial sum.

```
        // update the best partitions

        gainsum += largestGain;

        if (checkBalance() && gainsum >= bestgain)
        {
            bestgain = gainsum;
            bestcellPartitions.assign(cellPartitions.begin(), cellPartitions.end());
        }
```

  Note that in this process, the balance criteria would be checked.

There are no other introduced techniques to improve the cut size. Meanwhile, no other parallellism are introduced to speed up the algorithm.

2. **Running the program**

In our implementation, we use CMake to compile the overall program. I write the python script *build.py* in the scripts folder to help with the compilation as:

```
./scripts/build.py -o release
```

Then the executable file $hw1$ would be generated in the *bin* folder. Then we could run the executable file with the following command as:

```
cd bin
time -p ./hw1 <node_file> <net_file>  <out_file>
```

For example, if we want to run the FM partitioner on the p2-1.cells and p2-1.nets file, we could run:

```
time -p ./hw1 ../testcases/p2-1.cells ../testcases/p2-1.nets ../output/p2-1.out
```

The partitioning result for the case would be generated at ../output/p2-1.out file.

3. **Results**

We run the FM partitioner on the p2-1 to p2-5 testcases, and the results are as Table 1:

Table 1: Cut size and Runtime for p2-1 to p2-5 testcases with the implemented FM Partitioner

Testcases	Nodes	Nets	Cut	Time(s)
p2-1	375	357	10	0.02
p2-2	6049	4944	338	0.08
p2-3	104213	106374	6042	1.48
p2-4	172597	169350	45861	3.43
p2-5	172597	169350	133951	7.63

Table 2: Runtime Analysis for p2-1 to p2-5 testcases with the implemented FM Partitioner

Testcases	Nodes	Nets	$T_{IO}(s)$	$T_{computation}(s)$
p2-1	375	357	0.0025	0.0060
p2-2	6049	4944	0.0174	0.0448
p2-3	104213	106374	0.2769	1.1942
p2-4	172597	169350	0.5057	2.7292
p2-5	172597	169350	1.1683	5.2804

- **Comparison with the top 5 results**

  From the aspect of the cut size, our FM partitioner ranks 4th in the p2-2 and p2-4 cases, while the solution produced by our partitioner could not perform better in the other cases. From the aspect of the running time, our FM partitioner could beat the top-5 FM partitioners because of the introduced techniques like bucket set and the critical nets in the gain update.

  Some techniques would help if we would like to reduce the cut size of our partitioner.

  – Better Initialization: it has significant influence on the performance of the partitioning. In our implementation, we initialize the bipartitioning by divding the cells into two parts with similar cell sizes with the order of id. We could try multiple randomized initialization of the partitioning.

  – Coarsening: proper coarsening would help to reduce the scale of the partitioning problem. We could try to coarsen the nodes with strong connectivity.

- **Analysis of the runtime**

  The runtime is composed of the time spent on I/O and the time for FM partitioning. Here, we give the runtime analysis in Table 2. The time for FM partitioning dominates the overall time of running the program.

4. **Reflections**

   In this homework, I implement the FM partitioning algorithm from scratch. I have better understandings of this partitioning algorithm and practice my coding skills. In addition, I learn some knowledge of how to write CMakeFile to help the compilation of a complex project. Although the cut size is not good as the top-5 solution, the speed of our FM partitioner is relatively better than the others. In the future, we would consider some techniques like better initialization and multi-level to improve our cut size.