Report for Programming Assignment #1
2-Way Fiduccia–Mattheyses Circuit Partitioning

Student Name: `Yi-En Wu`
Student ID: `B11901188`
Course: Physical Design for Nanometer ICs
Instructor: Prof. Yao-Wen Chang

April 2025

---

1. Introduction

This project implements the Fiduccia–Mattheyses (FM) heuristic for 2-way circuit partitioning. The goal is to partition a set of cells into two disjoint, balanced groups while minimizing the number of cut nets (cut size). The balance constraint is determined by a user-defined balance factor d.

2. Key Algorithm

The FM heuristic moves one cell at a time, selecting the one with the highest gain (i.e., the largest reduction in cut size). After each move, it locks the cell, updates the gain of neighboring cells, and tracks cumulative gain to potentially revert to the best partial solution found during the pass.

3. Data Structures

3.1 Bucket list

The most important component in my implementation is the `Bucketlist` class, which allows:

- O(1) insertion and deletion of cells (if not operating on max gain cell)
- O(max_num_of_net) deletion for maximum gain cell
- O(1) access to the cell with the maximum gain
- Fast gain updates after each move

3.2 Net

The `Net` class represents a hyperedge connecting multiple cells. It is a lightweight class that stores:

- **Net Name:** A unique identifier for the net.
- **Cell List:** A list of integers representing the IDs of connected cells.
- **Partition Counts:** An array `_partCount[2]` that keeps track of how many connected cells are in partition A (0) and B (1).

3.3 Cell and Node

Each cell is associated with a `Node` structure that is used in the bucket list. A node contains:

- Cell ID
- Pointers to the previous and next node in a doubly-linked list

3.4 Bucket list Design

The `Bucket list` class maintains a mapping from gain values to linked lists of nodes. Below is a random example, where each dummy node represents a specific gain value, and the doubly-linked list will connect all the cells with the same gain.

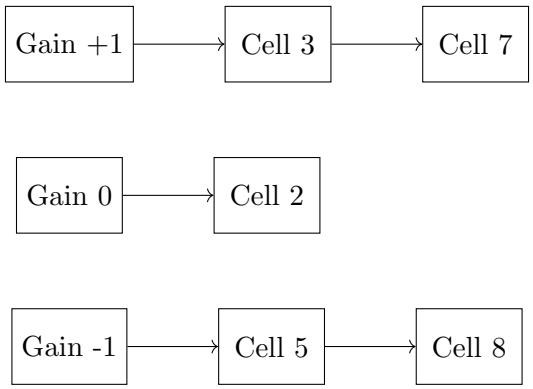

Figure 1: Visualization of the Bucketlist structure with dummy head nodes and cell nodes for each gain level.

- **Gain Range:** From $-P_{max}$ to $+P_{max}$, where P_{max} is the maximum number of nets a cell is connected to.
- **Structure:** Uses a `std::map<int, Node*>` called `head_list_` where each gain value points to a dummy head node of a linked list.
- **Maximum Gain Tracking:** An integer `max_gain_` is maintained to allow constant-time retrieval of the highest-gain cell.

3.5 Important Functions

Constructor

```
Bucketlist(int max_pin_num) {
    for (int i = -max_pin_num; i <= max_pin_num; ++i)
        head_list_[i] = new Node();
    max_gain_ = -max_pin_num;
}
```

Allocates dummy head nodes for each possible gain value.

insertNode()

Inserts a node to the front of the linked list corresponding to its gain. Also updates `max_gain_`.

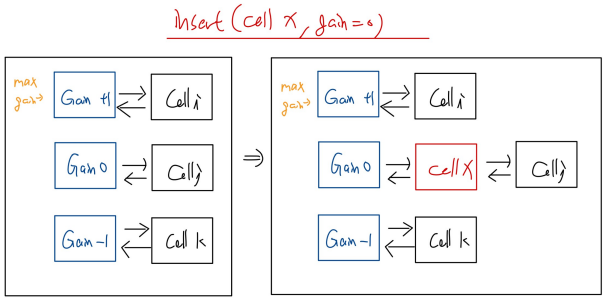

Figure 2: Bucketlist structure with insertion

deleteNode()

Removes a node from its list and updates `max_gain_` if necessary. Edge cases like removing the last node at the current max gain are handled. Only when deleting the node with maximum gain, and it should also be the only node with such gain, will cost linear time instead of constant time.

Deleting with the node which doesn't have the maximum gain

In this case, the deleted node (Cell X) is *not* the only node at the current maximum gain level. It lies between other nodes in the same bucket (e.g., Cell j). The deletion operation simply updates the `prev` and `next` pointers of the surrounding nodes. Since `max_gain_` remains unchanged, the operation runs in constant time.

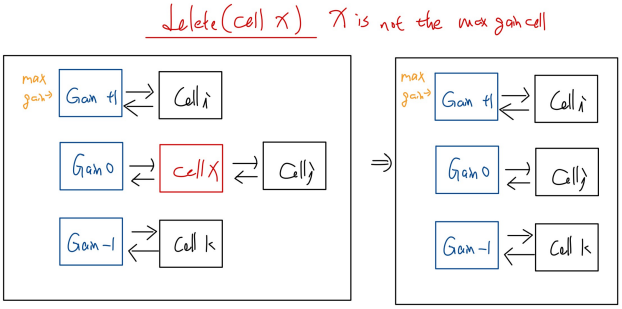

Figure 3: Bucketlist structure with deletion (not deleting the node with maximum gain)

Deleting the node with maximum gain

Here, the node being deleted (Cell X) is the *only* node in the bucket with the current `max_gain_`. After its deletion, the bucket becomes empty. The algorithm must then search downward to find the next non-empty gain level, updating `max_gain_`. This edge case requires up to $O(P_{max})$ time, where P_{max} is the maximum number of pins, but is still efficient in practice due to the limited range of gain values.

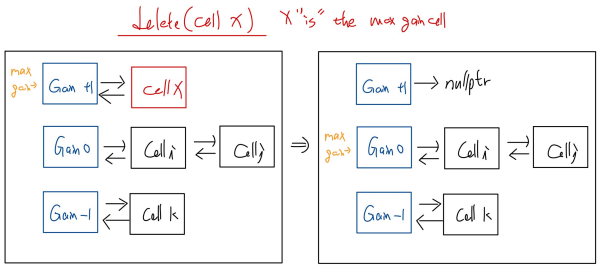

Figure 4: Bucketlist structure with deletion (deleting the node with maximum gain)

3.6 Advantages

- Efficient gain-based selection using `max_gain_`
- O(1) insertion and deletion using linked list node pointers
- Avoids linear search through gains
- Clean modular interface that integrates with FM core logic

4. Findings and Observations

I found out that the `Bucketlist` structure is crucial in implementing this FM partitioning algorithm. Since the algorithm updates the cells connected to the critical net in each iteration, it should be efficient to update the gain for the affected cells. Class `Bucketlist` can insert and delete nodes in constant time and only needs linear time when deleting the node with the maximum gain.

For runtime improvement, I have tried multiple ways, but none of them have significant improvement. What I have changed is that when sometimes I need to access a vector or some address frequently, I will create a local reference to store the parameter, which can slightly improve the runtime since the program will not need to fetch the same data repeatedly.

Lastly, for the quality of the solution, i.e., the number of the cutsize, I believe that the initial partition plays an important role because the initial decision will separate the circuit to two parts, and the randomness of this cut can affect the final solution a liitle. However, during the partition process, there are still several ways can be tested whether effective in improving the solution quality. For example, if I try this method again, I would try to treat not all the nets equally, because some of the nets may be more critical than others. By treating the nets differently, maybe the partition can be more effective.

5. Conclusion

This project demonstrates an efficient implementation of the FM heuristic using well-designed data structures. The `Bucketlist` class enables fast gain lookups and updates, allowing the algorithm to scale well for large input instances.