ESE 556 - VLSI Physical and Logic Design Automation

Fiduccia–Mattheyses Algorithm

Submitted by

Thomas Karl (111558962)

Felix Chimbo Cungachi (113470260)

Karan Rajendra (115375145)

.

1. Abstract

In the domain of Very Large Scale Integration (VLSI) design, the partitioning of circuits at the gate level plays a pivotal role in addressing the complexities associated with modern semiconductor devices. This report delves into the implementation and thorough examination of the Fiduccia-Mattheyses algorithm, a heuristic technique esteemed for its effectiveness in refining partitioning outcomes. Our investigation adapts the FM algorithm to the specific challenges of gate-level design partitioning, offering a comprehensive overview of its implementation nuances and executing a series of empirical analyses to assess its performance. The essence of the FM algorithm lies in its ability to iteratively enhance the division of a circuit's netlist into two distinct subsets. Our algorithm employs the creation of several structures and classes along with the inclusion of vectors in C++ programming that acts as the gates and buckets that allows for the development of a dynamic strategy that evaluates the "gain" of moving individual gates between partitions. This is where the gain metric quantifies the potential reduction in cutset size resulting from such moves to be updated before it gets iterated to the next one. This approach facilitates a nuanced balance between reducing the interconnection complexity and maintaining the area balance across partitions, thus optimizing the overall design for both performance and scalability. Through this exploration, we aim to underscore the algorithm's capacity to markedly improve partitioning efficiency within VLSI design workflows, providing valuable insights into its operational dynamics and the broader implications for tackling contemporary circuit design challenges that proved to be more challenging than what we expected due to our overall results underperforming.

2. Introduction:

Partitioning within VLSI circuits represents a fundamental challenge, particularly when striving to achieve a minimal cut, which is essential. The purpose of partitioning is to organize the cells into blocks in such a manner that connections across blocks are minimized. This can also be achieved recursively by arranging the blocks to reduce both block density and wire length.

When it comes to graph partitioning in the context of electronic design, various methods were employed to be able to do the following. One of which was introduced in the 1970s by Brian W. Kernighan and Shen Lin[1] which is known as the Kernighan-Lin (KL) algorithm. The KL algorithm begins with a given initial partition of the graph and iteratively improves the partitions by swapping pairs of nodes between the two partitions ultimately exhibiting a sort of gain that allows the algorithm to identify the reduction in its cost functions that leads to it having minimal cut size in the end. The KL algorithm was limited in terms of scalability to larger graphs, it had a significant impact on graph partitioning where it led to further developments that extended its approaches in the early 1980s.

In 1982, C.M. Fiduccia and R.M. Mattheyses[2] would publish a graph partitioning algorithm that addressed the problems that faced the KL algorithm when it came to large-scale hypergraphs. The Fiduccia Mattheyses (FM) algorithm aimed to achieve a better partitioning of the nodes in the graph, having it consider the weights of the nodes and edges it has. This information would then lead to the introduction of buckets that allowed for the management of the gains of nodes that occur during the partitioning process. It would then iterate the process of being able to calculate the gains, updating the buckets, and balancing the partitions until there are no more improvements needed, allowing it to be the most efficient algorithm to date. The partitioning strategy leverages hypergraphs and is notable for its linear time complexity, making the FM algorithm exceptionally efficient with an operational time complexity of O(n)[3]. This stands in contrast to the method proposed by Kernighan and Lin, which exhibits a time complexity of O(n^2)[3] demonstrating how it can function better when dealing with large-scale hypergraphs. Unlike its predecessors, the FM algorithm is adept at handling multi-pin nets beyond the simple two-terminal nets. It accommodates components of varying sizes and allows for certain cells to be pre-assigned or "locked" into a partition at the outset. These characteristics render the FM algorithm particularly advantageous for applications that necessitate bipartitioning.

3. Problem Statement:

Given a set of benchmarks our end goal is to implement and experiment the Fiduccia-Mattheyses partitioning algorithm for a gate-level design to minimize the cutset size while meeting the given area constraints that are fixed for the partitions.

The given benchmark consists of a netlist file of 3 files each with its own characteristics that must be taken into consideration. The format of the first file includes five entries in its header that includes: ignored, pins, nets, modules, and pad offset. These are then followed by the list of nets that are numbered from 1 to the pad offset that represents the hypergraph where the pads as the start/end of the nodes and cells are the nodes that are between the pads. The following file consists of the list of cells and pads that are followed by its area. The final file gives a directional I/O for the cells that allows it to be an input, an output, or in some cases be bidirectional.

4. Related Work:

As previously discussed there have been many other algorithms that have been developed to be able to reach minimal cut size of the partitions such as the KL algorithm and that further expanded what the FM algorithm is capable of. This section will review relevant literature and related algorithms, that further explains the concept of the partitioning techniques and how it impacts recent developments.

In addressing the challenges on partitioning, K. Shook[4] tackled a similar project with the goal of testing the KL algorithm and the FM algorithm where the end result would be comparing the execution times. Shook discusses the two algorithms as well as his methods of reading the files that led to the conclusion that despite the longer execution time on small inputs, when it comes to multiterminal nets, the FM algorithm is far superior to the KL in most cases. This is further backed up by the statement Shook made of how the operational time complexity is O(n) for the FM algorithm compared to the O(n^2) for the KL algorithm.

M.V. Sheblaev and A.S. Sheblaeva[5] followed the FM algorithm that involves a new method for finding the initial partitioning that makes it possible to to balance the hypergraph when partitioning. They proposed a method that uses geometrical properties and dimensional reduction methods to combat the large dimensions that are imposed by hypergraphs. As a result it led to partitions with better quality than known approaches and achieved the best known solutions based on its benchmarks for more than 50% of the tests that have been done.

S.A. Mim, M. Zarif-Ul-Alam, R. Reaz, M.S. Bayzid, and M.S. Rahman[6] further expanded the FM algorithm via improving the Quartet FM (QFM) algorithm for better running time as it is excessively useful for its tree qualities however it was lacking in run time. This resulted in them being able to improve the algorithm to exceed previously used QFM by 400 times faster than what was previously used on larger datasets.

5. Proposed Solution:

In order to apply a Fiduccia-Mattheyses algorithm to a netlist file, a starting point is needed with the creation of object types that can define either a cell or a net. Once we have our ‘Cells’ and ‘Nets’ representations, the next step is to apply data structures that will save and organize our objects. These data structures are to include some list type to hold all existing cells, and another to hold all existing nets. Another structure is the bucket structure, which would give the ability to organize cells by their potential gain, making it easier to find the highest gain cell. An additional object we are going to add is a ‘state’ object, so a sort of checkpoint can be made when a minimum cut size is found during the algorithm. The final structure needed will be a vector to keep track of each found optimal state during each pass of the FM algorithm. At that point, all the tools to execute the Fiduccia-Mattheyses are present.

The Gate class will be constructed as follows:

```
class Gate {
private:
    std::string name;
    std::vector<Net*> nets;
    bool Part;
    int GainTot;
    int area;
    bool isLocked = false;
```
Figure 1. *Gate object class.*

We referred to each cell as a 'Gate.' Each gate contains it's name, what partition it is in as a bool, the gain score if the gate was to be moved, an area, whether it's locked, and most importantly a vector of pointer to several net objects. The vector contains any nets that the gate is attached to. The 'Part' bool which when true represents that the gate is present in partition 1, and partition 2 when false.

The other object is the Net struct. A struct was used since there are far less changes being made to a net. Technically the net and gate object could have been made to be either a class or struct. The Net struct is as follows:

```
struct Net {
    Net() = default;
    std::string name; // Name of the net
    std::vector<Gate*> gates; // Pointers to Gates connected to this net
    int p1cnt = 0; // Count of gates in partition 1 connected to this net
    int p2cnt = 0; // Count of gates in partition 2 connected to this net
    bool cut = false;

    Net(const std::string& name) : name(name) {}
};
```
Figure 2. *Net object struct.*

The Net struct includes its name, two integers that represent the number of gates in partition 1 and 2, and a bool indicating whether the net is to be cut. Like the Gate class, the struct contains a vector of pointers to Gate objects that the net connects.

To store these objects two different maps are used for their respective object type. A map comes with the preprogrammed library <map>. It is similar to the python dict, where both hold a list of elements, but these elements are not accessed by an index, but by a key of some type decided by the programmer. In the case of both of these maps the key will be a string representing the name of either the gate or the net. The only advantage of the map was to make the scanner faster, since for the rest of the operations, the gates and nets had pointers to each other.

The scanner was defined as ‘void read()’ and took in a .net filename, a .are filename, and both the gate and net map. The method opens and reads each line of the .net file first. Each time a new gate name is seen a new gate object will be created and added to the gate map. Each time an ‘s’ character is seen in a line, a new net object is created and any gates seen from this point until another ‘s’ is read will be added to that nets vector of gates, and that gate will add said net to its vector of nets. This is why the maps are used so that when a brand new net at the end of the scanner hold a gate from the beginning it doesn’t need to waste time iterating through the list of gates to find the object it needs to add to it’s vector, and can instead access that object immediately since the name is already known. When the gates are added to nets, they are put into the second partition by default. The second major step of the scanner is to iterate through each line of the .are file, where the first word in the line is the name of the gate, thus having a key for the gate map, then an area that we can set that object's area to attribute to.

After the scanner runs, random initial partitions need to be generated. This method will generate a random iterator index somewhere in the range of the gate map, move that gate to the other partition, and add that index to a set of numbers. Each time a new index is generated it will have to be checked if that index is contained in the set, and if it is a different index will have to be generated. The logistics of what happens when a gate is swapped from one partition to the other will be covered later on in this section. It also has to be made sure that the partition size is updated in our ‘state’ object, which I will describe now.

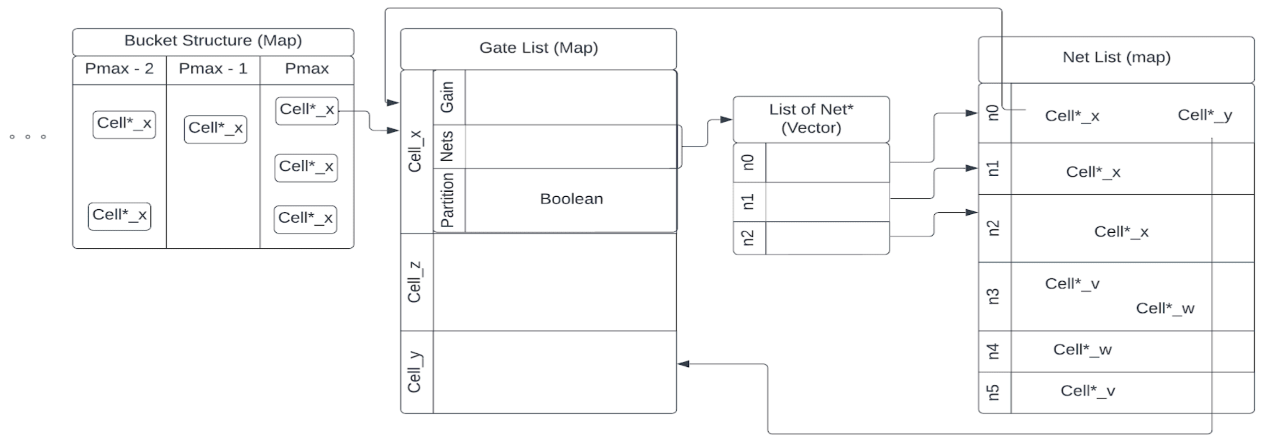
The goal of the state object is to keep a record of what gates are in what partition, the partition total area sizes, and the cutsize of the entire circuit at any given time. The struct is set up as follows:

```
struct state {
    std::vector<Gate*> P1; // Partition 1 list
    std::vector<Gate*> P2; // Partition 2 list
    int cutsize;
    int A1; // Area of partition 1
    int A2; // Area of partition 2
    std::map<int, std::set<Gate*>> buckets; // Buckets based on gain

    state() : cutsize(0), A1(0), A2(0) {}
};
```

Figure 3. *State object struct.*

This will keep two vectors of gate pointers, to represent which gates belong to what partition at any one time, a cut size attribute, and an integer area attribute to represent the total area in each partition at any time. The bucket structure is also found in the state object, labeled as ‘buckets.’ Buckets is another map, however instead of a string referring to an object, the key is now an integer representing a gain referring to a set of gate pointers pointing to gates with that gain. At this point we have all the data structures needed to execute our Fiduccia-Mattheyses algorithm:



At this point the remaining issue is the functionality of our algorithm. To go back to the random initialization function, for that to work we needed to include a class function for gates to control what happens when a gate is moved. When a gate is moved, its partition has to be set to !partition to reflect change in partition. After a gate is changed to another partition, the amount of nets being cut will also be changed, and the gain of that gate and any gate in all the affected nets need to be updated.

The swapping of the gate is handled by the class functions setPart() and unsetPart(). These functions are both contained in the togglePartitionAndRecalculateGain() method, which simply calls one or the other based on the gate partition. The objective is to iterate through each net the gate is connected to, increment/decrement p1cnt and p2cnt of each net depending on which partition the gate was swapped to, and then set the bool cut to represent whether or not the net is still to be cut.

After this is achieved the updateGainsAfterMove() function is called. The goal at this point is to iterate through all affected gates, remove their current entry in state.buckets, recalculate its gain, then add it back to the bucket structure at the appropriate index. This is all handled by the calculateInitialGain() gate class function, where a for loop of all nets the gate is included in is iterated in.

```
void Gate::updateGainsAfterMoveNew(Gate* movedGate, state & s) {
    set<Gate*> done;
    for (Net* net : movedGate->getNets()) {
        for (Gate* gate : net->gates) {
            if (done.count(gate) == 0) {
                s.buckets[gate->getGT()].erase(s.buckets[gate->getGT()].find(gate));
                gate->calculateInitialGain();
                s.buckets[gate->getGT()].insert(gate);
                done.insert(gate);
            }
        }
    }
}
```

Figure 4. *Called after a gate is moved. Update bucket location and updates gains cells.*

The first step before gates can be moved is to fill our bucket structure. The advantage of using a map as a bucket structure rather than an array of linked lists is that on creation there are no empty buckets. It also takes away the need to find the maximum possible gain, and maintain a MaxGain pointer. It will take shorter, if not the same amount of time to find a Gate pointer in the highest index bucket as it would to iterate a pointer through an array and find the first non-empty index, especially since the pointer would have to iterate through more empty buckets than with the map. After these buckets are filled, we can begin our algorithm.

How the FM function works is that it will iterate through the buckets starting with the highest gain index. Then if the bucket is empty it will continue, if not it will begin iterating through the pointer in the bucket and check if it is locked. If it is it will continue, if not it must be checked if the gate can be swapped based on partition area sizes. There are several qualifications that determine whether a gate can be swapped.

When the random initial partitions are generated, the partition sizes are random as well and are likely to be outside of the tolerance. Therefore if the current difference in areas is outside the tolerance and the absolute value of the difference between the partition area total will decrease then it’s a valid move. If the gate is in a partition with a smaller total area than the other partition, it is a valid move. If none of these are met then it will be checked if the move results in an acceptable difference in partition area. If any condition met the gate will be swapped, if not then the gate will be skipped and remain unlocked, since after moving other gates there should eventually be room to move that gate. This causes the execution time to suffer a bit, however should result in better functionality than locking them.

After it is determined that a gate is acceptable to be moved, the gate class function togglePartitionAndRecalculateGain() will be called as described previously. The program will then call a new function updatePartitionSizes() which does as the title says, and updates the sizes to the state object attributes, and removes the gate pointer from one partition vector and adds it to the other. The gate is then locked. The state is then checked to see if it has a smaller cutsize than the current minimum, and if it is the state will be saved to a holding variable called ‘result.’ This entire process is looped until either the current cut size is greater than the cut size of the state at the beginning of the function, or every gate is locked. Once finished the method will return ‘result’ as an output.

This concludes one pass of the FM algorithm. The main() function will call the algorithm twice at first and then push those ‘result’ objects to a state vector that will keep track of every resulting state. Then a while loop will begin and check that the last state in that vector has a smaller cut size than the previous state until this condition is broken. We will then have our results in the second to last state in the ‘results’ vector<state>.

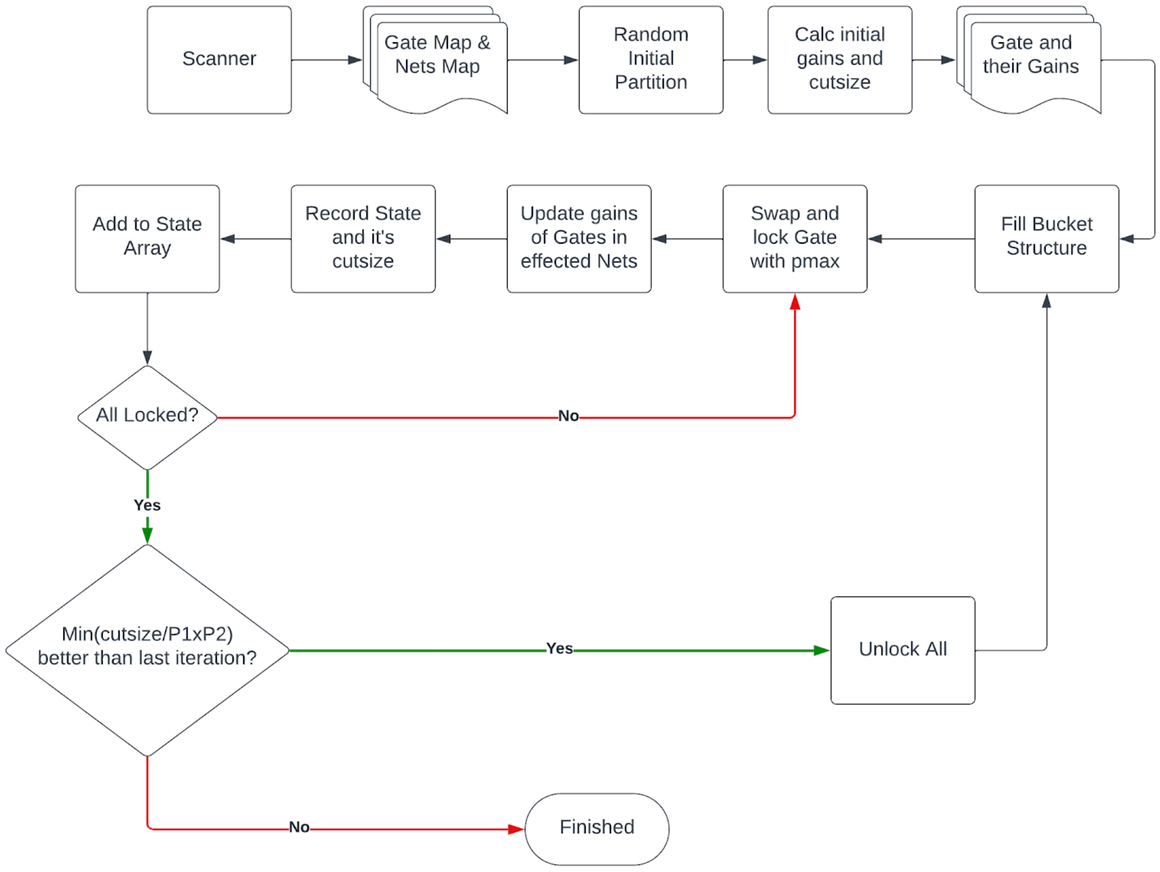

Scanner
Gate Map & Nets Map
Random Initial Partition
Calc initial gains and cutsize
Gate and their Gains
Fill Bucket Structure
Swap and lock Gate with pmax
Update gains of Gates in effected Nets
Record State and it's cutsize
Add to State Array
All Locked?
No
Yes
Min(cutsize/P1xP2) better than last iteration?
Yes
Unlock All
No
Finished

Figure 5. *Flowchart of the entire program.*

6. Implementation Issue:

The first main issue found after initial implementation was due to speed. Instead of updating the buckets as cells were moved, each time a move occurred the buckets map would be cleared and then refilled. The change here was instead of calling fillBuckets in the performFMAlgorithm loop, we added the functionality of removing the gate at its original index and then adding it to its new bucket after recalculating gain in the updateGainsAfterMove() method.

The second issue found when implementing was with the pointer values not being changed after the Fiduccia-Mattheyses algorithm function finished, however the exact causes remain unclear. The two methods that were causing issues with this were updateGainsAfterMove() and then CanSwap().

The issue in updateGainsAfterMove() was that our team was trying to optimize the bucket structure even further by deleting the set at that index, however it would effectively remove all other pointers in that set that were still needed and causing runtime exceptions. Now typing it out, I'm realizing that we should have been checking whether the set was empty, however we opted to just remove that feature since it was the clearer solution at the time.

The issue in CanSwap() proved to be equally as difficult as debugging; both of these functions were time consuming, even if the solution was somewhat straightforward. The result of the issue was that the state was saving the initial state at the beginning of the loop, therefore when debugging we were looking at our net and gate maps trying to keep track of the changes, which due to the large amount of both seemed to be a tedious task. However, backtracking to the fact that the original problem came from the initial addition of the canSwap() we began debugging that function and realized a couple of our variables should have been declared as a double instead of an int, and this caused the if condition to act strangely, and the behavior of the FM loop to behave even stranger.

More problems arose when getting to larger .net files such as ibm05 and onward. This created significant speed problems. Our solution to cut off a constant amount of time was tweaking the converge condition. Now rather than waiting for the total cut size to reach the cut size seen when originally seen, the algorithm will end if the current cut size is about a quarter of the difference between the original and the minimum cut size, since during debugging no improvement is seen past this point. The event of the cut size reducing back to minimum is very small and not worth the time spent moving low gain gates. Another improvement made for this issue is not saving the state every time a minimum is seen, but only after a minimum sets a flag and then a new cut size is greater than the minimum. This way the true minimum is not saved but the move right after is saved which is relatively the same in large netlists. A final addition to increase the speed is to not save two full vectors filled with a pointer to every object every time a minimum is found. We also fixed the fact that each gate's netlist would be iterated through twice each time the gains needed to be updated.

After adding these changes to our program, execution time still proved significantly high, so we decided to limit the amount of times we applied our algorithm to the netlist. We felt this to be a valid option in that the cut size improvement made to the netlist was logarithmic with each iteration. The following is a figure showing minimal cut size with the first ten iterations for ibm01:

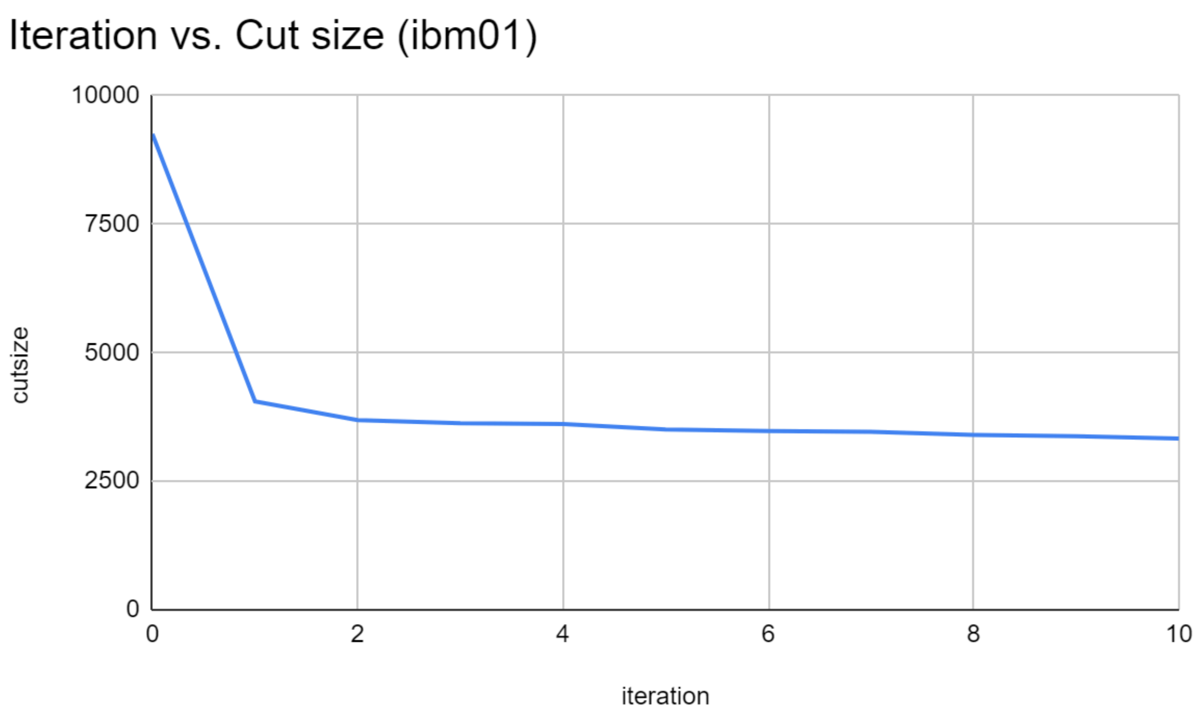


Figure 6. *Visual of cut size value with every iteration.*

Iteration	Cutsize
0	9257
1	4052
2	3689
3	3627
4	3616
5	3509
6	3475
7	3463
8	3404
9	3375
10	3332

Table 1. *Raw data for Figure 6. (ibm01)*

7. Experimental Results:

Benchmark	Cells	Nets	C_0	C_f	C improvement (%)	Partition Areas	Execution Time
ibm01	12752	14111	9257	3332	62.6816%	2.07168, 2.15834	33s
ibm02	19601	19584	13337	6170	53.7377%	5.89981 2.55853	78s
ibm03	23136	27401	17331	7085	59.1195%	4.19811 5.64477	109s
ibm04	27507	31970	20789	8222	60.4502%	5.04077, 4.25418	157s
ibm05	29347	28446	18842	6760	64.12%	2.30896, 2.163424	170s
ibm06	32498	34826	22986	10180	55.7122%	5.46041, 3.11738	193s
ibm07	45926	48117	32117	14192	55.8116%	5.85875, 5.9711	364s
ibm08	51309	50513	33478	14026	58.1038%	8.90454 4.54534	508s
ibm09	53395	60902	40385	17244	57.301%	1.05226, 7.00672	466s
ibm10	69429	75196	50480	23798	52.8566%	1.99813, 2.7553	757s
ibm11	70558	81454	54299	22719	58.1595%	9.49366,1.17437	860s
ibm12	71076	77240	52097	25686	50.6958%	1.79571,: 1.90178	865s
ibm13	84199	99666	66063	28278	57.1954%	1.35112e,: 1.15504	1278s

Figure 7. *Resulting data of all tested benchmarks.*

We plotted two line graphs using some metrics to validate our experiment results.

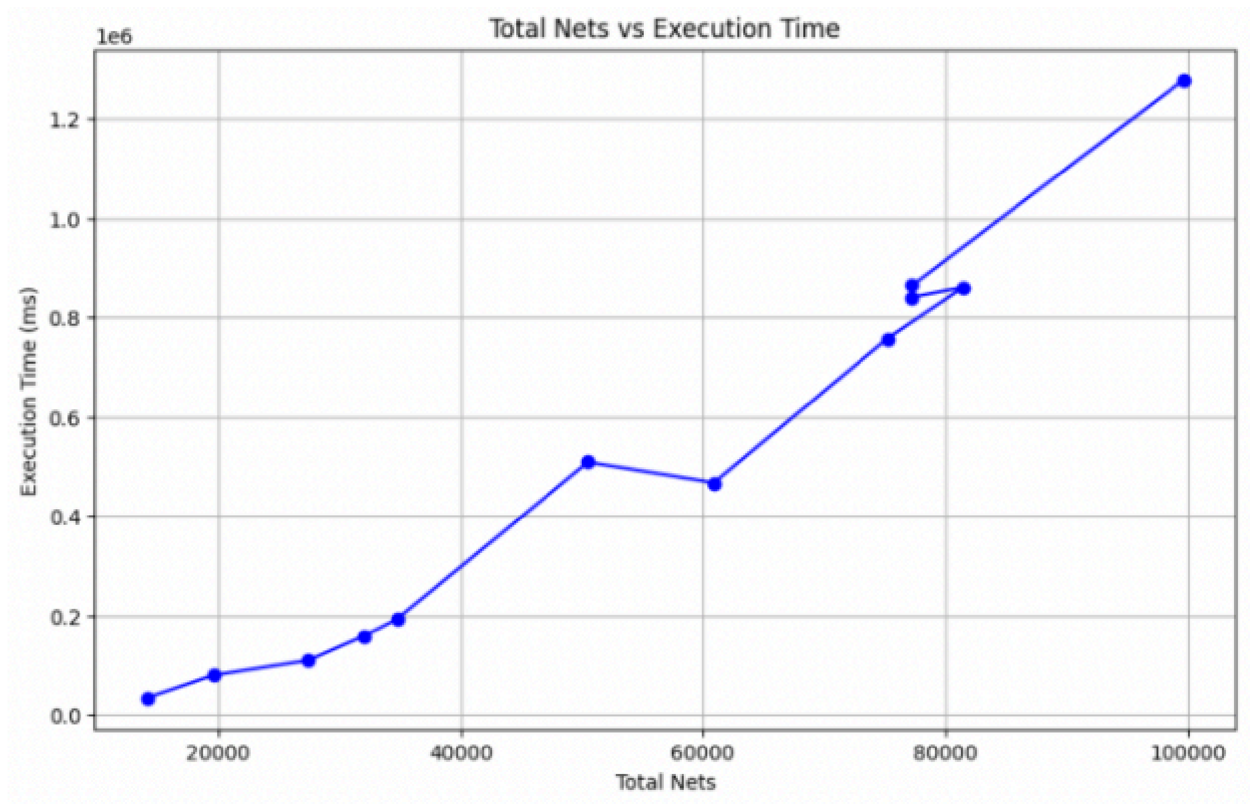


Figure 8. *No. of Nets Vs Execution Time (ms)*

In the Fig 8., we can see each scatter point represents an input circuit file which is ibm# .net file from ISPD98 benchmark suite. Here we plotted execution time of the partitioning algorithm for 13 input files (ibm01 to ibm013) against the number of nets in each circuit. We observed an increase in the time taken to execute as the number of nets increased further. The linear growth in time is attributed to the smart heuristic used by Fiduccia-Mattheyses Algorithm.

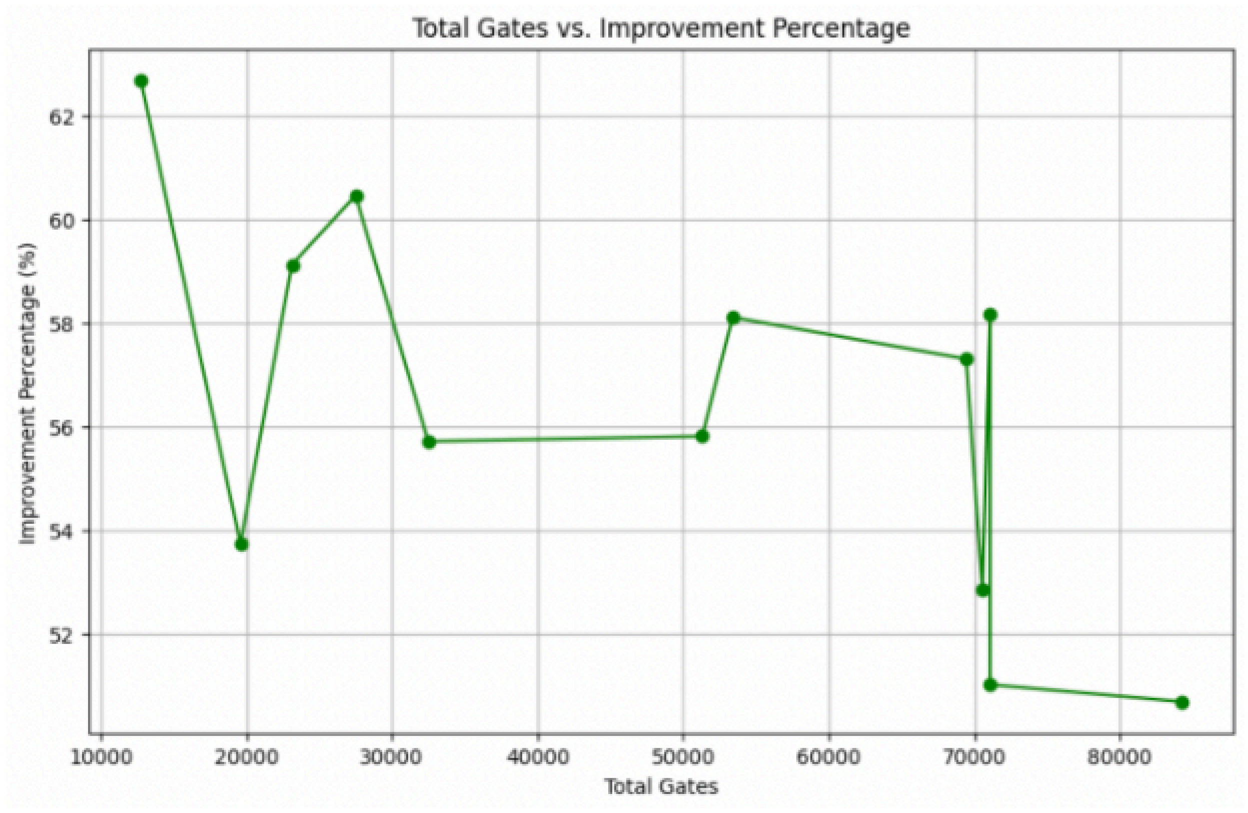


Figure 9. *No. of Modules Vs Cutsize Reduction Percentage*

In the Fig 9., we can see each scatter point represents an input circuit file which is ibm# .net file from ISPD98 benchmark suite. Here we plotted the cut size reduction percentage for 13 input files (ibm01 to ibm013) against the number of modules in each circuit. Basically, the cutsize is calculated from subtracting the resulting cutsize from the initial cutsize. We observed a decline over the graph from ibm01 to ibm13 from roughly 65% to 48% which is mainly due to the increase in number of modules, as the number of modules increase, the algorithm got difficult to make a large difference from initial number to final cutsize numb

8. Conclusion:

After our final implementation, we weren't able to reach the cut sizes listed in the hmetis table results, and our code proved to be significantly slow despite our best efforts to increase execution speed, so much so that we had to implement a runtime limit and take the best cut size at that limit, or else we would not get results for every benchmark. The largest contributing factor to the slow speed of the algorithm is the fact that for each gate, each net is iterated through and updated, and then each gate within each net must have their gain updated. This gives us a time complexity of O(cn^2), however our constant c is significant. Some possible solution would be to avoid updating these gains until it would affect the move of another gate, referred to as ‘lazy’ updates. This could be done by flagging a gate as stale and if it needs an update we can check for that flag. Our solution of limiting the amount of iterations was easier to implement.

9. Bibliography:

[1] Lin, S., and B. W. Kernighan. “An Effective Heuristic Algorithm for the Traveling-Salesman Problem.” *Operations Research*, vol. 21, no. 2, Apr. 1973, pp. 498–516, https://doi.org/10.1287/opre.21.2.498.

[2] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In "19th Design Automation Conference," pp 175-181, 1982.

[3] S. M. Sait and H. Youssef. VLSI Physical Design Automation: Theory and Practice. World Scientific Publishing Co., 1999.

[4] K. Shook. *Linear Time Fiduccia- Mattheyses Bipartitioning*, 2004

[5] Sheblaev, M.V., Sheblaeva, A.S. A Method of Improving Initial Partition of Fiduccia–Mattheyses Algorithm. *Lobachevskii J Math* **39**, 1270–1276 (2018). https://doi.org/10.1134/S1995080218090196

[6] Sharmin Akter Mim, Md Zarif-Ul-Alam, Rezwana Reaz, Md Shamsuzzoha Bayzid, Mohammad Saifur Rahman, Quartet Fiduccia–Mattheyses revisited for larger phylogenetic studies, *Bioinformatics*, Volume 39, Issue 6, June 2023, btad332, https://doi.org/10.1093/bioinformatics/btad332

10. Appendix:

DataStructures.cpp:

```
#pragma once
#include <iostream>
#include <vector>
#include <map>
#include <set>
#ifndef DATASTRUCTURES_HPP
#define DATASTRUCTURES_HPP

class Gate; //forward declaration

struct Net {
    Net() = default;
    std::string name; // name
    std::vector<Gate*> gates; // pointers to Gates connected to this net
    int p1cnt = 0; // count of gates in partition 1 connected to this net
    int p2cnt = 0; // count of gates in partition 2 connected to this net
    bool cut = false; // whether this net is cut
    Net(const std::string& name) : name(name) {}
};

struct state;

class Gate {
private:
    std::string name;
    std::vector<Net*> nets;
    bool Part;
    int GainTot;
    int area;
    bool isLocked = false;

public:
    Gate();
    Gate(std::string name, std::vector<Net*> nets, bool part, int gain, int
area, bool lock);
    ~Gate();
    std::vector<Net*> getNets() const;
    void addNet(Net* net);
    void removeNet(Net* net);
    void setPart(int& cutsize, std::map<std::string, Net>& netmap);
    void setPartNew(int& cutsize, state& s, std::map<std::string, Net>&
netmap);
    void unsetPart(int& cutsize, state& s, std::map<std::string, Net>&
netmap);
    bool getPart() const;
    int getGT() const;
    void incGT();
    void decGT();
    std::string getName() const;
    int getArea() const;
    void setArea(int a);
    void lock();
    void unlock();
    void calculateInitialGain();
    static void updateGainsAfterMove(Gate* movedGate);
    void togglePartitionAndRecalculateGain(int& cutsize, state& s,
std::map<std::string, Net>& netmap);
    friend std::ostream& operator<<(std::ostream& os, const Gate& g);
    bool getIsLocked() const;
    void updateGainsAfterMoveNew(Gate* movedGate, state& s, Net* &net);
};
struct state {
    std::vector<Gate*> P1; //part 1 list
    std::vector<Gate*> P2; //part 2 list
    int cutsize;
    int A1; //partition 1
    int A2; //partition 2
    std::map<int, std::set<Gate*>> buckets; //bucket structure
    state() : cutsize(0), A1(0), A2(0) {}
};

#endif // DATASTRUCTURES_HPP
```

DataStructures.cpp:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <map>
#include <cmath>
#include "DataStructures.hpp"
#include <sstream>
using namespace std;

Gate::Gate() : name(""), Part(false), GainTot(0), area(0), isLocked(false)
{}
```

```
Gate::Gate(string name, vector<Net*> nets, bool part, int gain, int area,
bool lock)
    : name(name), nets(nets), Part(part), GainTot(gain), area(area),
isLocked(lock) {}

Gate::~Gate() {

}
std::vector<Net*> Gate::getNets() const {
    return nets;
}
void Gate::addNet(Net* net) {
    nets.push_back(net);

    // Adjust p1cnt or p2cnt based on the gate's current partition
    if (Part) {
        net->p1cnt++;
    }
    else {
        net->p2cnt++;
    }
    // Recalculate gains for all gates in this net
    Gate::updateGainsAfterMove(this);
}

void Gate::removeNet(Net* net) {
    nets.erase(std::remove(nets.begin(), nets.end(), net), nets.end());
    for (size_t i = 0; i < nets.size(); ++i) {
        if (nets[i] == net) {
            nets.erase(nets.begin() + i);
            // Adjust p1cnt or p2cnt based on the gate's current partition
            if (Part) {
                net->p1cnt--;
            }
            else {
                net->p2cnt--;
            }
            // Recalculate gains for all gates in this net
            Gate::updateGainsAfterMove(this);
            break;
        }
    }
}
```

```
void Gate::setPart(int& cutsize, map<string, Net>& netmap) {
    if (!Part) {
        Part = true;
        for (Net* net : nets) {
            auto it = netmap.find(net->name);
            it->second.p1cnt++;
            it->second.p2cnt--;
            if (net->p2cnt == 0 && net->cut == true) {
                net->cut = false;
                netmap[net->name].cut = false;
                it->second.cut = false;
                cutsize--;
            }
            else if (net->p1cnt == 1 && net->cut == false) {
                net->cut = true;
                netmap[net->name].cut = true;
                it->second.cut = true;
                cutsize++;
            }
        }
        // Recalculate gains for all gates in affected nets
        Gate::updateGainsAfterMove(this);
    }
    else {
        cout << "Trying to setPart when already set." << endl;
    }
}

void Gate::setPartNew(int& cutsize, state& s, map<string, Net>& netmap) {
    if (!Part) {
        Part = true;
        for (Net* net : nets) {
            auto it = netmap.find(net->name);
            it->second.p1cnt++;
            it->second.p2cnt--;
            if (net->p2cnt == 0 && net->cut == true) {
                net->cut = false;
                netmap[net->name].cut = false;
                it->second.cut = false;
                s.cutsize--;
                if (cutsize == 0) {
                    cout << "stop" << endl;
                }
            }
            else if (net->p1cnt == 1 && net->cut == false) {
                net->cut = true;
                netmap[net->name].cut = true;
                it->second.cut = true;
                s.cutsize++;
            }
            if (cutsize == 0) {
                cout << "stop" << endl;
            }
            Gate::updateGainsAfterMoveNew(this, s, net);
        }
    }
    else {
        cout << "Trying to setPart when already set." << endl;
    }
}

void Gate::unsetPart(int& cutsize, state& s, map<string, Net>& netmap) {
    if (Part) {
        Part = false;
        for (Net* net : nets) {
            auto it = netmap.find(net->name);
            it->second.p1cnt--;
            it->second.p2cnt++;
            if (net->p1cnt == 0 && net->cut == true) {
                net->cut = false;
                netmap[net->name].cut = false;
                it->second.cut = false;
                s.cutsize--;
                if (cutsize == 0) {
                    cout << "stop" << endl;
                }
            }
            else if (net->p2cnt == 1 && net->p1cnt == 0) {
                net->cut = true;
                netmap[net->name].cut = true;
                it->second.cut = true;
                s.cutsize++;
                if (cutsize == 0) {
                    cout << "stop" << endl;
                }
            }
            Gate::updateGainsAfterMoveNew(this, s, net);
        }
    }
}


bool Gate::getPart() const {
    return Part;
}

int Gate::getGT() const {
    return GainTot;
}

void Gate::incGT() {
    GainTot++;
}

void Gate::decGT() {
    GainTot--;
}

string Gate::getName() const {
    return name;
}

int Gate::getArea() const {
    return area;
}

void Gate::setArea(int a) {
    area = a;
}
bool Gate::getIsLocked() const {
    return isLocked;
}

void Gate::lock() {
    isLocked = true;
}

void Gate::unlock() {
```

```cpp
    isLocked = false;
}


void Gate::calculateInitialGain() {
    int gain = 0;
    for (Net* net : nets) {
        int p1cnt = net->p1cnt;
        int p2cnt = net->p2cnt;
        if (Part) { // Gate is in partition 1
            if (p1cnt == 1) gain++;
            if (p2cnt == 0) {
                gain--;
                net->cut = false;
            }
            else net->cut = true;
        }
        else { // Gate is in partition 2
            if (p2cnt == 1) gain++;
            if (p1cnt == 0) {
                gain--;
                net->cut = false;
            }
            else net->cut = true;
        }
    }


    GainTot = gain;
}


void Gate::updateGainsAfterMove(Gate* movedGate) {
    for (Net* net : movedGate->getNets()) {
        // Update the gains of all gates on those nets
        for (Gate* gate : net->gates) {
            gate->calculateInitialGain();
        }
    }
}

void Gate::updateGainsAfterMoveNew(Gate* movedGate, state& s, Net* &net) {
    set<Gate*> done;
    for (Gate* gate : net->gates) {
        if (done.count(gate) == 0) {
s.buckets[gate->getGT()].erase(s.buckets[gate->getGT()].find(gate));
            gate->calculateInitialGain();
            s.buckets[gate->getGT()].insert(gate);
            done.insert(gate);
        }
    }
}

void Gate::togglePartitionAndRecalculateGain(int& cutsize, state& s, 
map<string, Net>& nets) {
    if (!isLocked) {
        if (Part) {
            this->unsetPart(cutsize, s, nets);
        }
        else this->setPartNew(cutsize, s, nets);
    }
    else {
        cout << "Gate is locked and cannot change partitions.\n";
    }
}
ostream& operator<<(ostream& os, const Gate& g) {
    os << "Gate: " << g.getName() << ", Area: " << g.getArea() << ", Part: 
" << (g.getPart() ? "1" : "2");
    return os;
}
```

Scanner.cpp:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <map>
#include <random>
#include <set>
#include <chrono>
#include "DataStructures.hpp"
#include <sstream>
using namespace std;
```

```cpp
void read(const string& netfile, const string& arefile, map<string, Gate>&
gates, map<string, Net>& nets, int& netTot, int& nodeTot) {
    gates.clear(); // Clear existing gates
    nets.clear(); // Clear existing nets

    string line;
    fstream myFile;
    myFile.open(netfile, ios::in);
    if (myFile.is_open()) {
        int iter = 0;
        int netcount = 0; // To keep track of nets
        string netname;
        while (getline(myFile, line)) {
            if (line.empty()) continue; // Skip empty lines
            if (iter == 2) {
                netTot = stoi(line);
                iter++;
            }
            else if (iter == 3) {
                nodeTot = stoi(line);
                iter++;
            }
            else if (iter == 0 || iter == 1 || iter == 4) {
                iter++;
            }
            else {
                // Parsing line
                vector<string> words;
                string temp;
                stringstream ss(line);

                while (ss >> temp) {
                    words.push_back(temp);
                }

                if (words.size() < 2) continue; // Skip invalid lines

                if (words[1] == "s") {
                    // Start of a new net
                    Net newNet("n" + std::to_string(netcount));
                    nets[newNet.name] = newNet;
                    netname = newNet.name;
                    netcount++;
                }

                if (gates.find(words[0]) == gates.end()) {
                    // Gate not found, create new
                    gates[words[0]] = Gate(words[0], {}, false, 0, 0, 0);
                }
                Net* currentNet = &nets[netname];
                Gate* currentGate = &gates[words[0]];

                currentNet->gates.push_back(currentGate);
                currentGate->addNet(currentNet);

            }
        }
        myFile.close();
    }
    else {
        cerr << "Failed to open " << netfile << endl;
    }

    ifstream areFile(arefile);
    if (areFile.is_open()) {
        while (getline(areFile, line)) {
            stringstream ss(line);
            string gateName;
            int area;
            if (ss >> gateName >> area) {
                gates[gateName].setArea(area);
            }
        }
        areFile.close();
    }
    else {
        cerr << "Failed to open " << arefile << endl;
    }
}


void randInitParts(map<string, Gate>& gm, int gtotal, state& s, int&
cutsize, map<string, Net>& netmap) {
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(0, gtotal - 1);
    set<int> chosen;
    int index;
    for (int i = 0; i < (gtotal / 2); i++) {
        do {
            index = dis(gen);
        } while (chosen.count(index) > 0);
        chosen.insert(index);
        auto it = std::next(gm.begin(), index);
        it->second.setPart(cutsize, netmap);
        s.P1.push_back((&it->second));
        s.A1 += it->second.getArea();
    }
    for (auto it = gm.begin(); it != gm.end(); it++) {
        if (chosen.count(distance(gm.begin(), it)) == 0) {
            s.P2.push_back(&it->second);
            s.A2 += it->second.getArea();
        }
    }
    s.cutsize = cutsize;
}



void initializePartitionSizes(state& s, const map<string, Gate>& gates) {
    s.A1 = s.A2 = 0;
    for (const auto& gatePair : gates) {
        const Gate& gate = gatePair.second;
        if (gate.getPart()) {
            s.A1 += gate.getArea();
        }
        else {
            s.A2 += gate.getArea();
        }
    }
    std::cout << "Initial Partition Sizes => P1: " << s.A1 << ", P2: " <<
s.A2 << std::endl;
}


void fillBuckets(state& s, std::map<std::string, Gate>& gates) {
    s.buckets.clear(); // Reset the buckets

    for (auto& gatePair : gates) {
        Gate& gate = gatePair.second;
        gate.calculateInitialGain(); // Make sure this updates GainTot
correctly
        s.buckets[gate.getGT()].insert(&gate);
    }
}

int calculateCutSize(const std::vector<Net>& nets) {
    int cutSize = 0;
    for (const auto& net : nets) {
        if (net.p1cnt > 0 && net.p2cnt > 0) {
            cutSize++;
        }
    }
    return cutSize;
}

bool canSwap(const Gate* gate, const state s, double thresh) {
    int gateArea = gate->getArea();
    double newA1 = s.A1, newA2 = s.A2;
    if (newA1 == newA2) { // have to allow some changes
        return true;
    }
    else if (gate->getPart() && newA1 < newA2) {
        return true;
    }
    else if (!(gate->getPart()) && newA1 > newA2) {
        return true;
    }
    else if (gate->getArea() == 0) {
        return true;
    }
    else {
        int flag = 0;
        int origdiff = 0;

        if (abs(newA1 - newA2) > (newA1 + newA2) * thresh) {
            flag = 1;
            origdiff = abs(newA1 - newA2);
        }

        if (gate->getPart()) { // Moving from Partition 1 to Partition 2
            newA1 -= gateArea;
            newA2 += gateArea;

        }
        else { // Moving from Partition 2 to Partition 1
            newA1 += gateArea;
            newA2 -= gateArea;
        }

        // Define a threshold for imbalance (e.g., one partition should not
be more than X% larger than the other)
        int totalArea = newA1 + newA2;
        if (flag == 1) {
            if (abs(newA1 - newA2) < origdiff) {
                return true;
            }
        }
        else return std::abs(newA1 - newA2) <= totalArea * thresh;
    }

}
void printAreaConstraints(const Gate* gate, const state& s, double thresh)
{
    double gateArea = gate->getArea();
    double newA1 = s.A1, newA2 = s.A2;

    if (gate->getPart()) {
        newA1 -= gateArea;
        newA2 += gateArea;
    }
    else {
        newA1 += gateArea;
        newA2 -= gateArea;
    }

    cout << "Current Partition Areas - P1: " << s.A1 << ", P2: " << s.A2 <<
endl;
    cout << "New Partition Areas if Swapped - P1: " << newA1 << ", P2: " <<
newA2 << endl;
    double totalArea = newA1 + newA2;
    double imbalance = abs(newA1 - newA2) / totalArea;
    cout << "Imbalance Threshold: " << thresh << ", New Imbalance: " <<
imbalance << endl;
```

```
    bool wouldSwap = (imbalance <= thresh);
    cout << "Would swap under current threshold? " << (wouldSwap ? "Yes" :
"No") << endl;
}

void updatePartitionSizes(Gate* gate, state& s) {
    int gateArea = gate->getArea();
    if (!gate->getPart()) { //move to part 2
        s.A1 -= gateArea;
        s.A2 += gateArea;
        s.P1.erase(remove(s.P1.begin(), s.P1.end(), gate));
        s.P2.push_back(gate);
    }
    else { //move to part 1
        s.A1 += gateArea;
        s.A2 -= gateArea;
        s.P2.erase(remove(s.P2.begin(), s.P2.end(), gate));
        s.P1.push_back(gate);
    }
}


void unlockAllGates(map<string, Gate>& gates) {
    for (auto& gatePair : gates) {
        gatePair.second.unlock();
    }
}

int getCutSize(map<string, Net> netlist) {
    int cutsize = 0;
    for (auto pair : netlist) {
        if (pair.second.cut) cutsize++;
    }
    return cutsize;
}

state performFMAlgorithm(state& s, map<string, Gate>& gates, map<string,
Net>& nets, double thresh) {
    state result;
    bool improvement = true;
    int mincut = s.cutsize;
    int originalCutsize = s.cutsize;
    int track = 0;
    int swapflag = 0;
    int flag = 0;
    int convergeflag = 0;
    while (improvement) {
        improvement = false;
        for (auto it = s.buckets.rbegin(); it != s.buckets.rend(); ++it) {
            if (it->second.empty()) {
                improvement = false;
                continue;
            }

            for (auto gatePtr : it->second) {
                Gate* gate = gatePtr;
                if (gate->getIsLocked()) {
                    continue;
                }
                if (canSwap(gate, s, thresh)) {
                    gate->togglePartitionAndRecalculateGain(s.cutsize, s,
nets);
                    updatePartitionSizes(gate, s);
                    gate->lock();
                    track++;
                    cout << track << "/" << gates.size() << endl;
                    if (s.cutsize < mincut) {
                        mincut = s.cutsize;
                        flag = 1;
                    }
                    else if (flag == 1) {
                        result = s;
                        flag = 0;
                        convergeflag = 1;
                    }
                    if (abs(s.cutsize - mincut) > abs(originalCutsize -
mincut) * 0.25 && convergeflag == 1) {
                        improvement = false;
                        break;
                    }
                    if (track == 500 || track == 1000 || track == 5000) {
                        cout << getCutSize(nets);
                    }

                    improvement = true;
                    break; // Exit the loop to start from the highest gain
again
                }
                else {
                    cout << "couldnt swap" << endl;
                    swapflag = 1;
                    improvement = false;
                    continue;
                }

            }
            if (improvement) {
                break;
            }
            else if (swapflag == 1) {
                continue;
            }
            else goto converged;
        }
    }
converged: unlockAllGates(gates); //prepare for the next pass
    cout << "inner cutsize" << getCutSize(nets) << endl;
    return result;
}


void printVerificationOutput(const state& s, const map<string, Gate>&
gates) {
    cout << "Partition 1 Size: " << s.A1 << endl;
    cout << "Partition 2 Size: " << s.A2 << endl;
    int cutSize = 0;
    cout << "Cut Size: " << cutSize << endl;
}


void updateNets(map<string, Net>& netmap, map<string, Gate> gates) {
    for (auto pair : gates) {
        for (Net* net : pair.second.getNets()) {
            Net dummybef = netmap[net->name];
            netmap[net->name] = *net;
            Net dummy = netmap[net->name];
        }
    }
}
```

```
int main() {
    state currentState;
    int maxAreaDiff = 10000;
    std::map<string, Net> nets;
    std::string netfile = "ibm10.net";  //adjust path as necessary
    std::string arefile = "ibm10.are";  //adjust path as necessary
    double thresh = 0.0480;
    std::map<std::string, Gate> gates;
    int netTot = 0, nodeTot = 0;
    int cutsize = 0;
    auto startTime = std::chrono::high_resolution_clock::now();

    // Step 1: Load the nets and gates data from files
    read(netfile, arefile, gates, nets, netTot, nodeTot);
    std::cout << "Files read successfully. Total gates: " << gates.size()
<< ", Total nets: " << nets.size() << std::endl;

    std::cout << "Calculating initial cut size..." << std::endl;
    // Step 2: Randomly initialize partitions for gates
    randInitParts(gates, gates.size(), currentState, cutsize, nets);
    int initialCutSize = currentState.cutsize;
    // Step 3: Initialize the state object for the partitioning algorithm
    std::cout << "Initial Partition Sizes => P1: " << currentState.A1 << ",
P2: " << currentState.A2 << std::endl;
    fillBuckets(currentState, gates);
    // Step 4: Perform the FM partitioning algorithm
    map<string, Net> checkpoint = nets;
    state result = performFMAlgorithm(currentState, gates, nets, thresh);
    for (auto pair : nets) {
        if (checkpoint[pair.second.name].p1cnt != pair.second.p1cnt) {
            cout << "change" << endl;
            break;
        }
    }
    int cuts = getCutSize(nets);
    cout << "Rresults 1: " << result.cutsize << " where intital was " <<
initialCutSize << endl;
    cout << "New cutsize" << cuts << endl;
    vector<state> results;
    result.cutsize = getCutSize(nets);
    results.push_back(result);
    fillBuckets(currentState, gates);
    result = performFMAlgorithm(currentState, gates, nets, thresh);
    cout << "Result 2: " << result.cutsize << endl;
    result.cutsize = getCutSize(nets);
    int index = 1;
    results.push_back(result);
    while ((results.at(index).cutsize < results.at(index - 1).cutsize) &&
index <= 10) {
        fillBuckets(currentState, gates);
        result = performFMAlgorithm(currentState, gates, nets, thresh);
        result.cutsize = getCutSize(nets);
        results.push_back(result);
        index++;
        cout << "Result " << index + 1 << ": " << result.cutsize << endl;
    }

    // Step 5: Output some results for verification
    cout << "Initial Cut Size: " << initialCutSize << endl;
    std::cout << "Final Cut Size: " << result.cutsize << std::endl;

    double totalAreaP1 = currentState.A1;
    double totalAreaP2 = currentState.A2;
    std::cout << "Total Area P1: " << totalAreaP1 << ", Total Area P2: " <<
totalAreaP2 << std::endl;

    double percentageImprovement = 0.0;
    if (initialCutSize > 0) { //avoid zero division
        percentageImprovement = 100.0 * (initialCutSize - result.cutsize) /
initialCutSize;
    }
    cout << "Improvement Percentage: " << percentageImprovement << "%" <<
endl;

    auto endTime = std::chrono::high_resolution_clock::now();
    auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);

    std::cout << "Execution Time: " << duration.count() << " ms" <<
std::endl;
    std::cout << "Improvement Percentage: " << percentageImprovement << "%"
<< std::endl;
    ;
    std::ofstream outFile("results.txt", std::ios::app);

    if (outFile.is_open()) {
        outFile << "Total Gates: " << gates.size() << ", Total Nets: " <<
nets.size() << endl;
        outFile << "Initial Cut Size: " << initialCutSize << std::endl;
        outFile << "Final Cut Size: " << result.cutsize << std::endl;
        outFile << "Improvement Percentage: " << percentageImprovement <<
"%" << std::endl;
        outFile << "Total Area P1: " << totalAreaP1 << ", Total Area P2: "
<< totalAreaP2 << std::endl;
        outFile << "Execution Time: " << duration.count() << " s" <<
std::endl;
        outFile << "----------------------" << std::endl;
        outFile.close();
    }
    else {
        std::cerr << "Unable to open file for writing." << std::endl;
        outFile.close();

    }

    return 0;
}
```