Stony Brook University

ESE 556 VLSI Physical and Logic Design Automation

PROJECT 1 REPORT

FIDUCCIA-MATTHEYSES ALGORITHM

SUBMITTED BY: -

Aniket Deenanath Singh (SBU ID: 115375358)

Animesh Uttekar (SBU ID: 115382330)

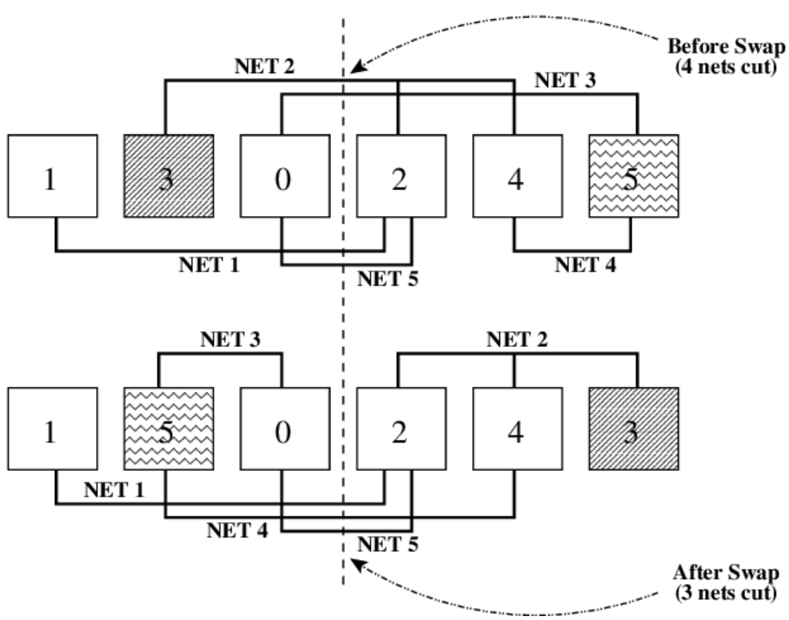
Table of Contents

Introduction:

A chip may contain several million transistors. Due to the limitations of memory space and computation power available it may not be possible to layout the entire chip (or generically speaking any large circuit) in the same step. Therefore, the chip (circuit) is normally partitioned into sub-chips (sub-circuits). These sub-partitions are called blocks. The actual partitioning process considers many factors such as the size of the blocks, number of blocks, and number of interconnections between the blocks. The output of partitioning is a set of blocks and the interconnections required between blocks. In large circuits, the partitioning process is hierarchical and at the topmost level a chip may have 5 to 25 blocks. Each block is then partitioned recursively into smaller blocks.

Partitions can be done at different levels like Board Level or System Level depending upon the need and requirement. It is one of the major factors which plays an important role in the cost of manufacturing the device and thus we try to minimize it. Different algorithms are been used to reduce the partition size like Kernnighan-Lin etc. All did a brilliant job, but every algorithm has drawbacks, leading to the evolution of more optimized and better algorithms. One such optimized algorithm is Fiduccia-Mattheyses.

Fiduccia-Mattheyses algorithm (FM algorithm hereafter) is another heuristic partitioning algorithm which generalizes the concept of swapping of nodes introduced in the KL algorithm. To contrast FM algorithm with KL algorithm, FM algorithm is designed to work on hypergraphs and instead of swapping a pair of nodes as was happening in KL algorithm, FM algorithm swaps a single node in each iteration. The basic essence of FM algorithm is the same as KL algorithm – we define gains for each vertex of the (hyper)graph, select one node according to some criterion, remove it from its present partition and put it to the other partition, lock that vertex, update gains of all other unlocked vertices and iterate these steps until we reach a local optimum configuration. The tool hMETIS implements an augmented version of FM algorithm (please refer to Existing tools for Graph Partitioning section).



Related Work:

From the paper "Gatti, A., Hu, Z., Smidt, T., Ng, E. G., & Ghysels, P. (2022). Deep Learning and Spectral Embedding for Graph Partitioning. In *Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing* (pp. 25-36). Society for Industrial and Applied Mathematics." [1] Popular and widely used graph partitioning codes are METIS and Scotch. Both packages also have distributed memory implementations: ParMETIS and PTScotch, and METIS recently also gained a multithreaded variant called mt-METIS. Both METIS and Scotch use a multilevel graph partitioning framework, where the graph is first partitioned on a coarser representation (recursively)and the partitioning is then interpolated back and re-fined. Many heuristics can be used for the coarse level partitioning and the refinement. Popular choices are for instance Fiduccia-Mattheyses and Kernighan-Lin, diffusion, Gibbs-Poole Stockmeyer, greedy graph growing, etc. Spectral methods make use of global information of the graph, while combinatorial algorithms like Kernighan-Lin and Fiduccia-Mattheyses rely on local node connectivity information. Spectral methods have the advantage that they can be implemented efficiently on high performance computing hardware with graphics accelerators.

From the paper "Sheblaev, M. V., & Sheblaeva, A. S. (2018). A method of improving initial partition of fiduccia–mattheyses algorithm. Lobachevskii Journal of Mathematics, 39(9), 1270-1276." [2] The authors presented a method of initial data generation for Fiduccia–Mattheyses algorithm, which reduces initial dataset with to sparced new one. It allows to find balanced graph or hypergraph partitions with better quality than known approaches and often achives best known solutions on benchmarks set new method is better than randomly generated initial partitions.

From the paper "Song, H. J., & Kim, H. G. (2021). A Study of Adapted Genetic Algorithm for Circuit Partitioning. The Journal of the Korea Contents Association, 21(7), 164-170." [3] The authors proposed a solution space search method that combines a genetic algorithm and a probability evolution algorithm for circuit division that minimizes the interconnection wiring between the central circuits in the physical design process of VLSI. The analysis was compared with the genetic algorithm (GA) and simulated annealing (SA) methods; where Algorithms for obtaining an optimal solution in a circuit division problem include the Kernighan-Lin algorithm, Fiducia Mattheyses heuristic, and simulated annealing.

Proposed Solutions and Implementations Issues:

Proposed Solution:

The proposed solution utilizes the FM algorithm to partition VLSI circuits, aiming to minimize the cut size and optimize the balance between partitions.

The primary problem being addressed is VLSI circuit partitioning, where the goal is to divide a set of cells into two partitions, labeled as A and B. The objective is to minimize the number of interconnections (nets) crossing between the two partitions.

1. Global Variables:
   - Key parameters, such as the sizes of partitions A and B (sizeA and sizeB), cut set variables, and various tracking variables (temp_index, locked_cell_index, etc.), are initialized as global variables.
2. Buckets: Here to replace the traditional linkedlist based buckets present in the original paper, we use a key-value based python dictionary to store the gain values as well as to store the cell maps and net maps.

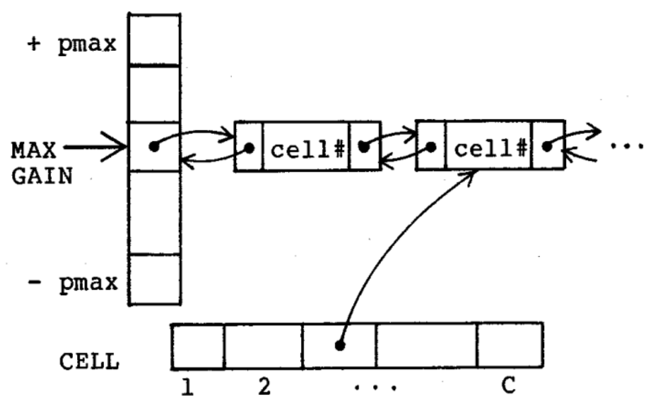
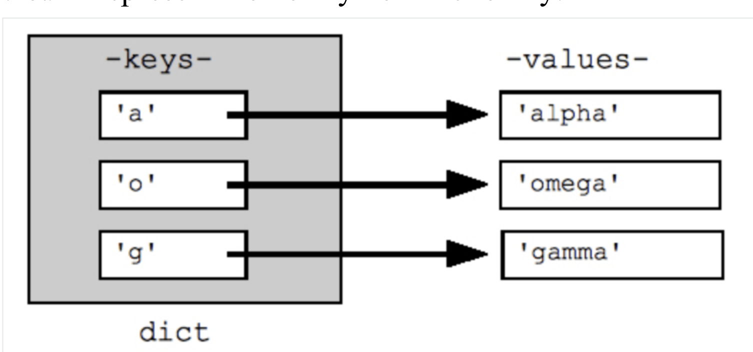


Figure 2. Bucket list structure

Visual Representation of Python Dictionary:



3. Data Representation: Classes Cell and Net
    - The code defines two classes:
        - Cell: Represents an individual cell with attributes such as ID, size, gain, partition assignment, lock status, and a set of connected nets.
        - Net: Represents a net with an ID, a list of cells connected to it, and sizes of partitions A and B.

4. Functions for Data Generation:
    - generate_cell_list(path): Parses a file containing cell information (*.are file) to generate a dictionary of cells (cell_map).
    - generate_net_list(path): Parses a file containing net information (*.net file) to generate a dictionary of nets (net_map).

5. Initial Partitioning: initial_partitions()
    - Randomly assigns cells to partitions A or B, considering size constraints to ensure a balanced initial partitioning.
    - Handles cases where swapping partitions could help achieve better balance.

6. Core FM Algorithm: fiducciaMathAlgo()
    - The core of the proposed approach revolves around the FM algorithm, which employs an iterative process to enhance the partitioning in order to minimize the cut size.
    - Utilizes gain buckets to efficiently identify cells with high gains.
    - Employs balancing checks (isbalanced()) and updates gains accordingly.

7. Update Strategies: update_buckets, update_gains
    - update_buckets(curr_cell, prev_gain): Adjusts gain buckets after a cell's gain is updated.
    - update_gains(target_cell): Updates gains for cells connected to the target cell after a move.

8. Main Execution:
    Iterations over a set of benchmark datasets and performs the following steps for each dataset:
    - Reads the circuit information from input files (.are and .net files).
    - Initializes cell and net maps.
    - Executes the FM algorithm (fiducciaMathAlgo) to find the best cut.
    - Prints the results, including the best cut size and execution time.

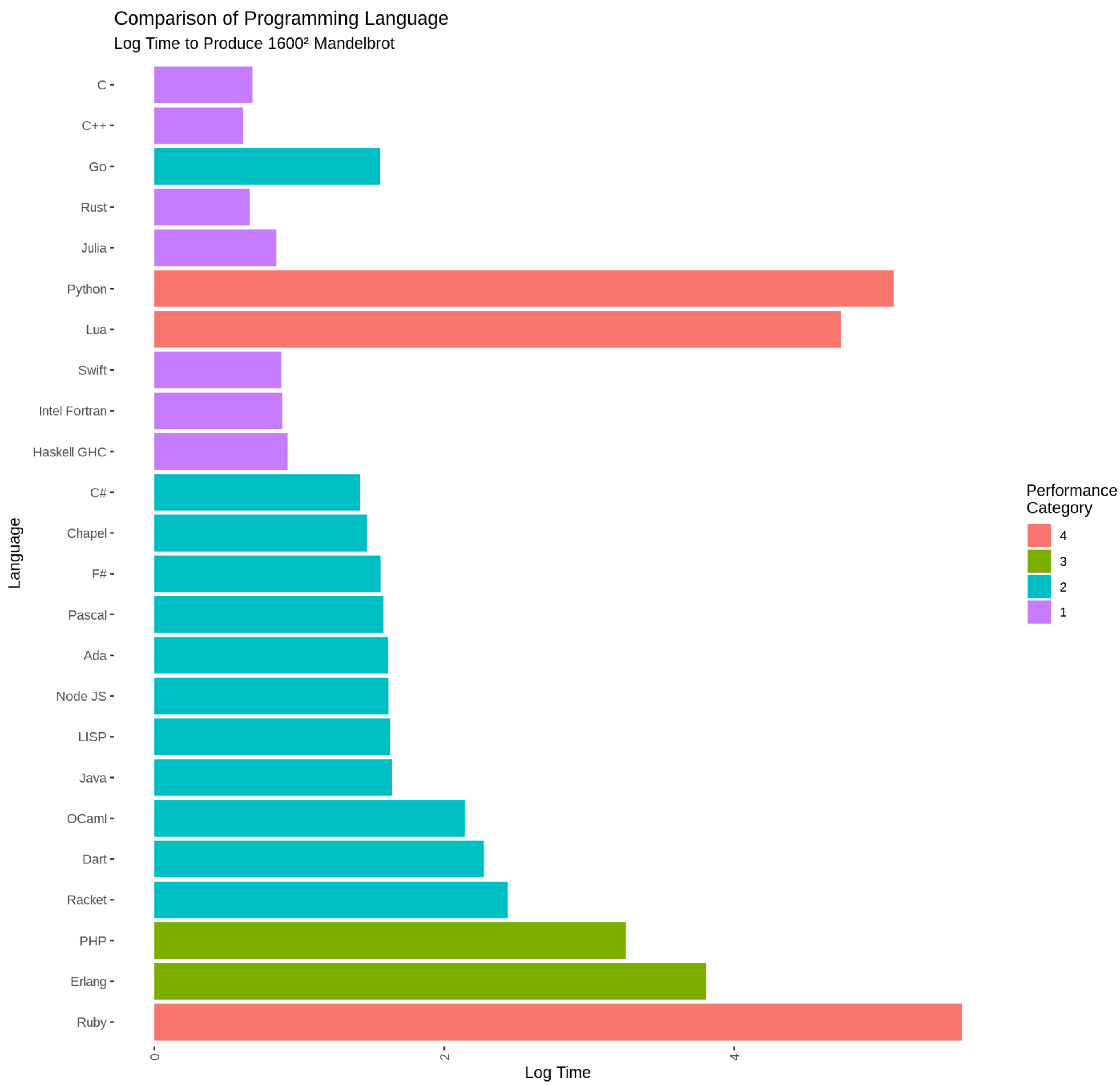
9. Benchmark Datasets:
    - The code demonstrates the algorithm's performance on a predefined set of benchmark datasets (datasets), representing different VLSI circuits.

10. Output and Timing Information:
    - The code prints information about the benchmark being processed, the start and end times of the execution, the best cut size found, and the total execution time.

Issues with Implementations:

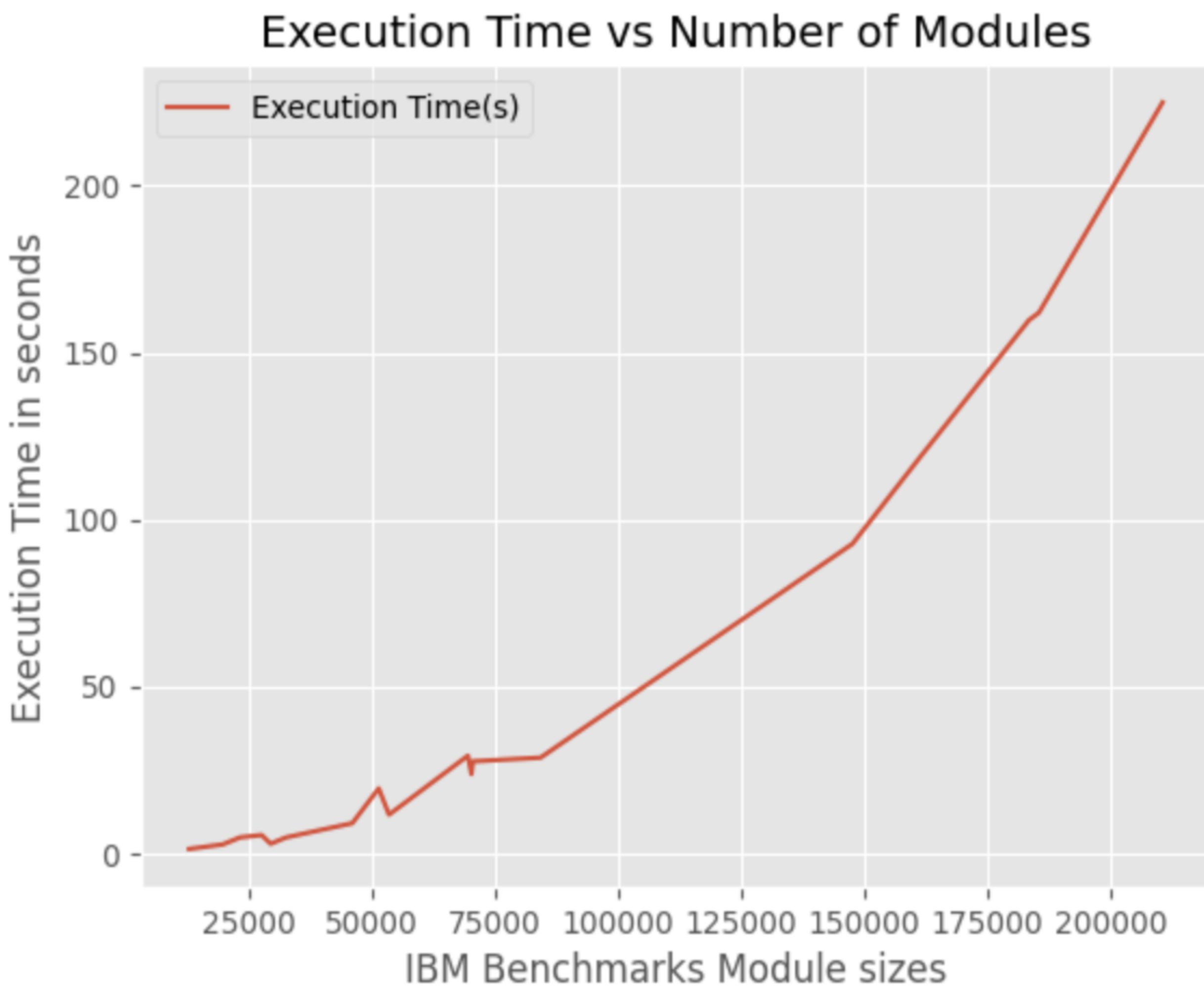
After fetching results, we find out that for larger netlists, Python Programming language becomes slow. Due to its interpreted nature python is slow for larger circuits. A low level language like C would be the better choice here for getting the faster results. Still Python was preferred because of its ability to interpret results and use its vast library functions to generate graphs and its ease of use

.



Experimental Results

We benchmark the algorithm using various ibm netlists in the ISPD 98 benchmark suite which is the standard VLSI benchmark data. While running the benchmark we realized that as the number of modules keep increasing, the execution time also starts increasing manyfolds. Hence smaller netlist like ibm1 and ibm2 were partitioned in less than 2 seconds whereas for bigger netlist like ibm18 it took more than 3 minutes



Code Output:

PROBLEMS 6 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
aniketsingh@Anikets-MacBook-Air FiducciaMattheyses % pythor
[INFO] Current Benchmark: ibm01
[INFO] STARTTIME: 2024-03-04 15:14:20.837588
[INFO] INITIAL CUT SIZE: 9191
[INFO] BEST CUT: 3430
[INFO] ENDTIME :  2024-03-04 15:14:22.126159
[INFO] Total Execution:  0:00:01.288571


[INFO] Current Benchmark: ibm02
[INFO] STARTTIME: 2024-03-04 15:14:22.126194
[INFO] INITIAL CUT SIZE: 13374
[INFO] BEST CUT: 5151
[INFO] ENDTIME :  2024-03-04 15:14:24.767241
[INFO] Total Execution:  0:00:02.641047


[INFO] Current Benchmark: ibm03
[INFO] STARTTIME: 2024-03-04 15:14:24.767265
[INFO] INITIAL CUT SIZE: 17318
[INFO] BEST CUT: 7550
[INFO] ENDTIME :  2024-03-04 15:14:29.521487
[INFO] Total Execution:  0:00:04.754222


[INFO] Current Benchmark: ibm04
[INFO] STARTTIME: 2024-03-04 15:14:29.521571
[INFO] INITIAL CUT SIZE: 20780
[INFO] BEST CUT: 8200
[INFO] ENDTIME :  2024-03-04 15:14:34.986814
[INFO] Total Execution:  0:00:05.465243
```

```
[INFO] Current Benchmark: ibm05
[INFO] STARTTIME: 2024-03-04 15:14:34.986839
[INFO] INITIAL CUT SIZE: 18999
[INFO] BEST CUT: 7191
[INFO] ENDTIME :  2024-03-04 15:14:37.843578
[INFO] Total Execution:  0:00:02.856739


[INFO] Current Benchmark: ibm06
[INFO] STARTTIME: 2024-03-04 15:14:37.843604
[INFO] INITIAL CUT SIZE: 22959
[INFO] BEST CUT: 8739
[INFO] ENDTIME :  2024-03-04 15:14:42.587785
[INFO] Total Execution:  0:00:04.744181


[INFO] Current Benchmark: ibm07
[INFO] STARTTIME: 2024-03-04 15:14:42.587827
[INFO] INITIAL CUT SIZE: 32010
[INFO] BEST CUT: 12176
[INFO] ENDTIME :  2024-03-04 15:14:51.556264
[INFO] Total Execution:  0:00:08.968437


[INFO] Current Benchmark: ibm08
[INFO] STARTTIME: 2024-03-04 15:14:51.556352
[INFO] INITIAL CUT SIZE: 33725
[INFO] BEST CUT: 12544
[INFO] ENDTIME :  2024-03-04 15:15:10.995757
[INFO] Total Execution:  0:00:19.439405
```

```
[INFO] Current Benchmark: ibm09
[INFO] STARTTIME: 2024-03-04 15:15:10.995838
[INFO] INITIAL CUT SIZE: 40214
[INFO] BEST CUT: 16367
[INFO] ENDTIME :  2024-03-04 15:15:22.570327
[INFO] Total Execution:  0:00:11.574489


[INFO] Current Benchmark: ibm10
[INFO] STARTTIME: 2024-03-04 15:15:22.570425
[INFO] INITIAL CUT SIZE: 50800
[INFO] BEST CUT: 20154
[INFO] ENDTIME :  2024-03-04 15:15:51.808181
[INFO] Total Execution:  0:00:29.237756


[INFO] Current Benchmark: ibm11
[INFO] STARTTIME: 2024-03-04 15:15:51.808264
[INFO] INITIAL CUT SIZE: 54150
[INFO] BEST CUT: 20792
[INFO] ENDTIME :  2024-03-04 15:16:15.561283
[INFO] Total Execution:  0:00:23.753019


[INFO] Current Benchmark: ibm12
[INFO] STARTTIME: 2024-03-04 15:16:15.561363
[INFO] INITIAL CUT SIZE: 52236
[INFO] BEST CUT: 22316
[INFO] ENDTIME :  2024-03-04 15:16:43.108916
[INFO] Total Execution:  0:00:27.547553
```

```
[INFO] Current Benchmark: ibm13
[INFO] STARTTIME: 2024-03-04 15:16:43.109013
[INFO] INITIAL CUT SIZE: 66268
[INFO] BEST CUT: 27408
[INFO] ENDTIME :  2024-03-04 15:17:11.764995
[INFO] Total Execution:  0:00:28.655982


[INFO] Current Benchmark: ibm14
[INFO] STARTTIME: 2024-03-04 15:17:11.765078
[INFO] INITIAL CUT SIZE: 101654
[INFO] BEST CUT: 36483
[INFO] ENDTIME :  2024-03-04 15:18:44.507733
[INFO] Total Execution:  0:01:32.742655


[INFO] Current Benchmark: ibm15
[INFO] STARTTIME: 2024-03-04 15:18:44.507843
[INFO] INITIAL CUT SIZE: 125755
[INFO] BEST CUT: 51656
[INFO] ENDTIME :  2024-03-04 15:20:43.652111
[INFO] Total Execution:  0:01:59.144268


[INFO] Current Benchmark: ibm16
[INFO] STARTTIME: 2024-03-04 15:20:43.652203
[INFO] INITIAL CUT SIZE: 129797
[INFO] BEST CUT: 50227
[INFO] ENDTIME :  2024-03-04 15:23:23.445054
[INFO] Total Execution:  0:02:39.792851
```

```
[INFO] Current Benchmark: ibm17
[INFO] STARTTIME: 2024-03-04 15:23:23.445152
[INFO] INITIAL CUT SIZE: 131746
[INFO] BEST CUT: 56903
[INFO] ENDTIME :  2024-03-04 15:26:05.519512
[INFO] Total Execution:  0:02:42.074360
```

```
[INFO] Current Benchmark: ibm18
[INFO] STARTTIME: 2024-03-04 15:26:05.519613
[INFO] INITIAL CUT SIZE: 139301
[INFO] BEST CUT: 46054
[INFO] ENDTIME :  2024-03-04 15:29:40.799027
[INFO] Total Execution:  0:03:35.279414
```

Conclusion

The above algorithm has been coded in python3. Its performance was evaluated by using it to partition all the current benchmarks present on:

https://vlsicad.ucsd.edu/UCLAWeb/cheese/ispd98.html#Benchmark%20File%20Format

In conclusion, the benefits of the heuristics of the Fiduccia-Mattheyses algorithms are observed when the size of the circuit is huge. It was also observed that with increasing size of the number of modules, the time it takes to find better cut sizes goes up significantly. Optimizations in the implementations can lead to even better performance in terms of execution time and memory. Also using a more system friendly programming language like c++ could further improve the results.

Bibliography

[1] A. H. Z. S. T. N. E. G. &. G. Gatti, "Deep Learning and Spectral Embedding for Graph Partitioning," *2022 SIAM Conference on Parallel Processing for Scientific Computing. Society for Industrial and Applied Mathematics,* pp. 25-36, 2022.

[2] M. V. &. S. A. S. Sheblaev, "A method of improving initial partition of fiduccia–mattheyses algorithm," *Lobachevskii Journal of Mathematics,* no. 39(9), pp. 1270-1276, 2018.

[3] H. J. &. K. H. G. Song, "A Study of Adapted Genetic Algorithm for Circuit Partitioning," *The Journal of the Korea Contents Association,* no. 21(7), pp. 164-170, 2021.

[4] C. Alpert, "The ISPD98 Circuit Benchmark Suite," [Online]. Available: https://vlsicad.ucsd.edu/UCLAWeb/cheese/ispd98.html.

Appendix

You can also refer the code on Aniket's githhub: https://github.com/Bruces1998/FM_algorithm

Code:

```
##### IMPORTS ######
import random
import datetime
import pickle as pl
import os

#### GLOBAL VARIABLES ####
sizeA = 0
sizeB = 0
sizeA_minimum_cut = 0
sizeB_minimum_cut = 0
cutset = 0
new_cutset = 0
temp_index = 0
locked_cell_index = 0
minimum_cut = 0
startcut = 0
old_partition = 0

locked_cells = []
gain_bucket = {}
max_gain_index = None  # index for gainA
best_cut_array = []
best_cut = float("inf")
best_copies_cell_map = []
```

```
cell_map = {}
net_map = {}

class Cell:
   def __init__(self, id):
      self.id = id
      self.cell_size = 0
      self.F = 0
      self.T = 0
      self.cell_gain = 0
      self.lock_status = 0  # 1=locked, 0=unlocked
      self.cell_type = ""
      self.cell_partition = 0  # A = 0, B = -1
      self.net_list = set()
      self.cell_size = 0

class Net:
   def __init__(self, id):
      self.id = id
      self.cell_list = []
      self.cutstate = 0  # 0 = uncut, -1 = cut
      self.Asize = 0
      self.Bsize = 0

def get_cut_size(cell_list, net_list):
  cut_size = 0

  first_partition = None

  for net in net_list:
    toggle = 0
    for cell in net_list[net].cell_list:
      if toggle == 0:
        first_partition = cell_list[cell].cell_partition
        toggle = 1
```

```
    else:
     if cell_list[cell].cell_partition != first_partition:
      cut_size += 1
      break

  return cut_size



def generate_cell_list(path):
  Cell_list = {}
  with open(path) as file:
   file_data = file.readlines()

   for i, cell_data in enumerate(file_data):
    cell_data = cell_data.split(" ")
    cell_id = cell_data[0]
    new_cell = Cell(cell_id)
    new_cell.cell_size = int(cell_data[1])

    Cell_list[new_cell.id] = new_cell



  return Cell_list

def generate_net_list(path):
  Net_list = {}
  net_id = 0
  i = 5
  global cell_map



  with open(path) as file:
   file_data = file.readlines()
   length = len(file_data)
   while i < length:
```

```
    net_id += 1
    new_net = Net(net_id)
    cell_id = file_data[i].split(" ")[0]
    new_net.cell_list.append(cell_id)
    cell_map[cell_id].net_list.add(net_id)
    if cell_map[cell_id].cell_partition == 0:
      new_net.Asize += 1
    else:
      new_net.Bsize += 1
    i+= 1

    while (i< length and (file_data[i].split(" ")[1] != "s")):

     cell_id = file_data[i].split(" ")[0]
     new_net.cell_list.append(cell_id)
     cell_map[cell_id].net_list.add(net_id)
     if cell_map[cell_id].cell_partition == 0:
       new_net.Asize += 1
     else:
       new_net.Bsize += 1
     i+= 1

    Net_list[net_id] = new_net


  return Net_list

def initial_partitions():
  global cell_map, sizeA, sizeB
  for node in cell_map:
    partition = random.choice([0, -1])
    if partition == 0:
      if sizeB + cell_map[node].cell_size - sizeA <= int(len(cell_map) * (0.1)):
        partition = ~partition
        sizeB += cell_map[node].cell_size
    elif partition == -1:
```

```
        if sizeA + cell_map[node].cell_size - sizeB <= int(len(cell_map) * (0.1)):
            partition = ~partition
            sizeA += cell_map[node].cell_size
    cell_map[node].cell_partition = partition
    cell_map[node].lock_status = 0


def isbalanced(target_cell):
    global sizeA, sizeB, cell_map
    temp_sizeA = sizeA
    temp_sizeB = sizeB
    if cell_map[target_cell].cell_partition == 0:
        temp_sizeA -= cell_map[target_cell].cell_size
        temp_sizeB += cell_map[target_cell].cell_size
    elif cell_map[target_cell].cell_partition == -1:
        temp_sizeA += cell_map[target_cell].cell_size
        temp_sizeB -= cell_map[target_cell].cell_size

    if abs(temp_sizeA - temp_sizeB) <= int(len(cell_map) * (0.1)):
        sizeA = temp_sizeA
        sizeB = temp_sizeB
        return True
    else:
        return False

def update_buckets(curr_cell, prev_gain):
    global gain_bucket
    new_gain = cell_map[curr_cell].cell_gain
    gain_bucket[prev_gain].remove(curr_cell)
    if not gain_bucket[prev_gain]:
        del gain_bucket[prev_gain]
    gain_bucket.setdefault(new_gain, []).append(curr_cell)

def update_gains(target_cell):
    global old_partition, gain_bucket
    for curr_net in cell_map[target_cell].net_list:
```

```
        if old_partition == 0:
            cell_map[target_cell].F = net_map[curr_net].Asize
            cell_map[target_cell].T = net_map[curr_net].Bsize
        elif old_partition == -1:
            cell_map[target_cell].F = net_map[curr_net].Bsize
            cell_map[target_cell].T = net_map[curr_net].Asize

        if cell_map[target_cell].T == 0:
            for curr_cell in net_map[curr_net].cell_list:
                if not cell_map[curr_cell].lock_status:
                    prev_gain = cell_map[curr_cell].cell_gain
                    cell_map[curr_cell].cell_gain += 1
                    update_buckets(curr_cell, prev_gain)

        elif cell_map[target_cell].T == 1:
            for curr_cell in net_map[curr_net].cell_list:
                if cell_map[target_cell].cell_partition == cell_map[curr_cell].cell_partition:
                    if not cell_map[curr_cell].lock_status:
                        prev_gain = cell_map[curr_cell].cell_gain
                        cell_map[curr_cell].cell_gain -= 1
                        update_buckets(curr_cell, prev_gain)
                        break

        cell_map[target_cell].F -= 1
        cell_map[target_cell].T += 1

        if old_partition == -1:
            net_map[curr_net].Asize = cell_map[target_cell].T
            net_map[curr_net].Bsize = cell_map[target_cell].F
        elif old_partition == 0:
            net_map[curr_net].Bsize = cell_map[target_cell].T
            net_map[curr_net].Asize = cell_map[target_cell].F

def fiducciaMathAlgo():
    global best_cut, sizeA, sizeB, cutset, minimum_cut, startcut, old_partition, locked_cells, gain_bucket,
max_gain_index, sizeA_minimum_cut, sizeB_minimum_cut, locked_cell_index
```

```
passes = 1

cutset = get_cut_size(cell_map, net_map)
startcut = cutset
minimum_cut = cutset
best_cut = float("inf")

print("[INFO] INITIAL CUT SIZE:", cutset)



for pass_num in range(1, passes + 1):
    if pass_num > 1:
        cutset = minimum_cut
        sizeA = sizeA_minimum_cut
        sizeB = sizeB_minimum_cut
        for i in range(len(locked_cells)):
            curr_cell = locked_cells[i]
            cell_map[curr_cell].lock_status = 0
            if locked_cell_index <= i:
                cell_map[curr_cell].cell_partition = ~cell_map[curr_cell].cell_partition

    gain_bucket.clear()
    for curr_cell in cell_map:
        cell_map[curr_cell].cell_gain = 0
        for curr_net in cell_map[curr_cell].net_list:
            if cell_map[curr_cell].cell_partition == 0:
                cell_map[curr_cell].F = net_map[curr_net].Asize
                cell_map[curr_cell].T = net_map[curr_net].Bsize
            elif cell_map[curr_cell].cell_partition == -1:
                cell_map[curr_cell].F = net_map[curr_net].Bsize
                cell_map[curr_cell].T = net_map[curr_net].Asize
            if cell_map[curr_cell].F == 1:
                cell_map[curr_cell].cell_gain += 1
            if cell_map[curr_cell].T == 0:
                cell_map[curr_cell].cell_gain -= 1
```

```
    gain_bucket.setdefault(cell_map[curr_cell].cell_gain, []).append(curr_cell)

locked_cells.clear()
max_gain_index = max(gain_bucket.keys())

while True:
    toggle = True
    if not gain_bucket:
        break
    else:
        target_cell = gain_bucket[max_gain_index][-1]
        temp_index = 0
    while toggle and not isbalanced(target_cell):
        if target_cell == gain_bucket[max_gain_index][0]:
            if max_gain_index == min(gain_bucket):
                break
            max_gain_index = max_gain_index - 1
            while max_gain_index > min(gain_bucket.keys()) and gain_bucket.get(max_gain_index) is None:
                max_gain_index -= 1
            temp_index = 0
            toggle = False
            continue
        temp_index = temp_index + 1
        target_cell = gain_bucket[max_gain_index][-1 - temp_index]

    if not toggle:
      continue

    cell_map[target_cell].lock_status = 1
    locked_cells.append(target_cell)
    new_cutset = cutset - cell_map[target_cell].cell_gain
    gain_bucket[max_gain_index].remove(target_cell)
    if not gain_bucket[max_gain_index]:
        del gain_bucket[max_gain_index]
    cutset = new_cutset
    old_partition = cell_map[target_cell].cell_partition
```

```
            cell_map[target_cell].cell_partition = ~cell_map[target_cell].cell_partition
            update_gains(target_cell)
            if 0 < new_cutset <= minimum_cut:
                minimum_cut = new_cutset
                sizeA_minimum_cut = sizeA
                sizeB_minimum_cut = sizeB
                locked_cell_index = len(locked_cells)
            if gain_bucket:
                max_gain_index = max(gain_bucket.keys())
        # current_cut = get_cut_size(cell_map, net_map)
        if minimum_cut < best_cut:
          best_cut = minimum_cut
        else:
          break
    return best_cut

if __name__ == "__main__":

    datasets = [
        "ibm01", "ibm02", "ibm03", "ibm04", "ibm05", "ibm06",
        "ibm07", "ibm08", "ibm09", "ibm10", "ibm11", "ibm12",
        "ibm13", "ibm14", "ibm15", "ibm16", "ibm17", "ibm18",
    ]
    current_directory = os.getcwd()

    for ibm_file in datasets:
        print("[INFO] Current Benchmark:", ibm_file)

        starttime = datetime.datetime.now()
        print("[INFO] STARTTIME:", starttime)

        cell_map = generate_cell_list(current_directory+'/dataset/'+ibm_file+'.are')
        initial_partitions()
        net_map = generate_net_list(current_directory+'/dataset/'+ibm_file+'.net')

        best_cut = fiducciaMathAlgo()
```

```
print("[INFO] BEST CUT:", best_cut)

endtime = datetime.datetime.now()
print("[INFO] ENDTIME : ", endtime)

print("[INFO] Total Execution: ", endtime - startime)

print("\n")
```

References:

https://www.semanticscholar.org/paper/Experimental-study-of-a-novel-variant-of-Fiduccia-Sinha-Mohanty/779f33210fd53da5867f37b979538df7e1c63e81

https://www.researchgate.net/figure/Circuit-Partitioning-Overview_fig2_4145666

http://vlsicad.ucsd.edu/UCLAWeb/cheese/ispd98.html

https://limsk.ece.gatech.edu/course/ece6133/project/FM.pdf

https://www.datacamp.com/tutorial/python-dictionaries

https://books.google.com/books/about/A_Guide_to_Programming_Languages.html?id=OxYpAQAAMAAJ