EEE6406 EDA Challenge: Scalable Partitioning Algorithm for Large-Scale Circuit Graphs

Marcus Simmonds

In this project, I attempted to implement the Fiduccia-Mattheyses (FM) algorithm for hypergraph partitioning. This algorithm is a greedy heuristic for selecting nodes to be separated into two partitions based on the concept of node "gains" which quantify the number of connections reduced by moving a node from one partition to the next.

Algorithm Details

The FM algorithm takes as input a set of hyperedges (referred to as nets) with each hyperedge containing at least 2 different nodes (referred to as cells). In my case, I utilized the .hgr file specified on the pace challenge website (pacechallenge.org/2019/htd/htd_format/). My code also takes as input another .hgr file format not consistent with this specified format.

The goal of the algorithm is to minimize the cut size of the input graph. The size of a graph cut in this case is the number of edges that cross partitions. This has practical applications in EDA and circuit design, as small cuts can prevent congestion and minimize trace area.

First, the algorithm takes these inputs and initializes two instances of a bucket list, which is a unique data structure utilizing a doubly linked list for quick insertion and deletion of items:

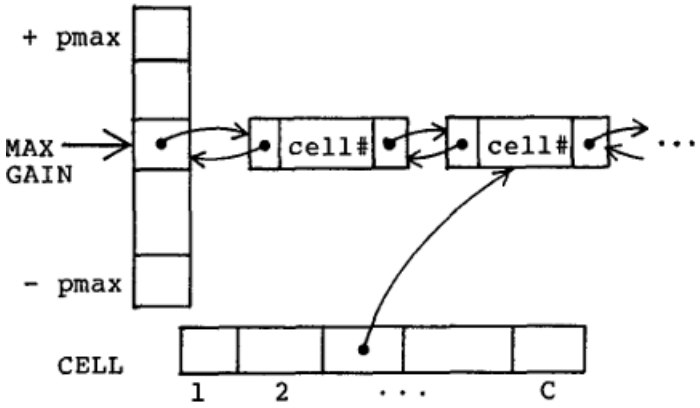

Figure 1: Bucket list structure

Each entry in the bucket list is a doubly linked list at an index corresponding to the cell's "gain". The gain is defined as the difference between outgoing nets and incoming nets. (the nets connected to cells outside of the current cell's partition minus the nets connected to cells within the current cell's partition) The algorithm does not specify a specific manner of cell insertion, but it has been observed that initialization can influence the quality of the solution. In addition to the bucket lists, the algorithm also initializes a net list and cell list for use in later traversals and calculations. For my specific implementation, I used a python dictionary and my own custom node structure.

Python dictionaries have a constant-time lookup, which when paired with lists with an O(1) or O(n) insert time depending on the location, provide the same benefits as using this bucket list structure.

Once the cells have been placed in an initial partition, the algorithm re-organizes the inserted cells by calculating their gains and placing each cell in the index corresponding with it's gain. This allows for constant time lookup and removal of indexes within the bucket list structure. The gain calculation is as follows:

```
/* compute cell gains */
FOR each free cell i DO
   g(i) ← 0
   F    ← the "from block" of cell(i)
   T    ← the "to block"   of cell(i)
   FOR each net n on cell i DO
      IF F(n) = 1 THEN increment g(i)
      IF T(n) = 0 THEN then decrement g(i)
   END FOR
END FOR
```

Figure 2: Gain update pseudocode

Where F is the block (partition) the base cell (the cell being updated) is in, and T is the block that the cell isn't in. F(n) is the number of nets that have the base cell (n) as it's only cell, and T(n) is the number of nets which contain the base cell but don't have any other cells in the T block. This is essentially calculating:

$$g(i) = FS(i) - TE(i)$$

Where g(i) is the gain of base cell i. FS(i) and TS(i) are F(n) and T(n) but for base cell i. This is a faster method of calculating the gain for a cell, which the creators of the algorithm have proved is more time efficient than manually counting nets.

Upon initialization, the main optimization process begins:

```
// Optimization loop pseudocode

cumulative_gain = 0
previous_gain = 0
best_gain = -inf
// select the node with the largest gain
while(node = get_largest_gain_node()){

    // Check if moving the node violates the balance constraint:
    if(partitions_balanced(node)){
        bucket_list.move_node(node)
        bucket_list.recalculate_gain()
        node.lock() // Prevent this node from being moved again
        cumulative_gain += node.gain()
     }

    if(cumulative_gain > best_gain){
        // Save the gain and our state
        best_gain = cumulative_gain
        save_moves()
    }
}

previous_gain = best_gain
cumulative_gain = 0
best_gain = -inf
unlock_nodes() // Reset locks
undo_to_best_gain() // Undo moves until you get back to the best gain
}
```

Figure 3: Single pass of the FM algorithm

The pseudocode shown is a single pass of the FM algorithm. It will continually move nodes until all nodes are locked or no more valid moves are available. Some key points:

- Balance Constraint:
    - The algorithm determines if moving a node will unbalance the partition. Most implementations aim for a 50-50 balance of nodes, with a beta parameter being used to adjust this.
- Node Lock:
    - To prevent nodes from being moved more than once, the algorithm “locks” them. This is typically done by maintaining a list of locked nodes and updating it when a node is moved.
- Gain recalculation:
    - To prevent recalculating the gain for every node, the gains of the neighbors of the moved node are updated. This prevents expensive iterations. After each pass, the gain for the entire bucket list is recalculated using the same method as the pseudocode shown before.
- Cumulative gain:
    - The cumulative gain of each pass is collected and stored. This is the heuristic that is used to optimize the cut size. Since a node’s gain is *positive* when moving it reduces the total graph cut size and negative when a node adds to the graph’s cut size, moving the node with the largest gain first will minimize the cut size.
- Hill-Climbing:
    - This algorithm uses a hill-climbing approach to optimize the cut size. The passes will continue until the previous gain is zero or negative, indicating a lack of improvement.
- Node Roll-Back:
    - The algorithm saves the state of the graph when it’s largest gain was observed and rolls back to it after every pass. This guarantees that each iteration will start with the best observed configuration.

This algorithm runs in linear time with respect to the number of hyperedges and nodes, mostly due to the efficient gain updates via the bucket list data structure and the gain recalculation technique.

Benchmarking and Results:

The files for my benchmarking and comparisons were obtained from this github repository: https://github.com/TILOS-AI-Institute/HypergraphPartitioning/tree/main

I attempted to match the benchmarking environment by introducing an additional 2% balance constraint in either direction. My results can be seen here, with comparisons to the lowest edge cut on the available leaderboard. My runtime for each of these graphs was in the order of seconds, which reinforces the linear time claim made by the original creators. These graphs have tens of thousands of nodes, so I'll only include the cut size and partition balance here.

Benchmark File	Cut Size	Partition Balance	Best Edge Cut
Ibm01.hgr	9201	50-50	202
Ibm02.hgr	13345	50-50	336
Ibm03.hgr	17378	50-50	954
Ibm05.hgr	18817	50-50	1719
Ibm06.hgr	22989	50-50	1719
Ibm07.hgr	31868	50-50	904
Ibm09.hgr	40260	50-50	620
Ibm10.hgr	50649	50-50	1313
Ibm11.hgr	53658	50-50	1062

Despite the fast runtime, the performance of my implementation performs *terribly* in comparison to the best edge cuts seen on the benchmark. I believe this may be due to several errors on my end, as the algorithm did not appear to come close to the best edge cut.

References:

C. M. Fiduccia and R. M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," 19th Design Automation Conference, Las Vegas, NV, USA, 1982, pp. 175-181, doi: 10.1109/DAC.1982.1585498. keywords: {Partitioning algorithms;Iterative algorithms;Data structures;Research and development;Computer networks;Approximation algorithms;Polynomials;Design automation;Pins},

Gemini Deep Research – *For finding benchmark dataset*