Fiduccia Mattheyses Algorithm Implemntation

VLSI Design and Automation Course Project by
Sachin Vernekar(10EC110), Manasij Venkatesh(10EC59), Sai Swaroop(10EC90)

Abstract

In all of the motivating applications for hypergraph partitioning, the size of input is growing and the demand for performance is high. The physical design methodology we apply to these designs must scale to tens of millions of components today and hundreds of millions in the foreseeable future. Because of these large inputs, any partitioning technique used must have near-linear complexity in the worst case in order to be effective. State-of-the-art partitioning tools use local search heuristics to refine a given partitioning solution. The basic technique common to all VLSI partitioning applications is the Fiduccia Mattheyses (FM) algorithm [1], which applies linear-time passes to iteratively improve a given partitioning solution by moving every node exactly once. In our project, we have implemented the flat FM algorithm on C++.

1. The FM Algorithm

```
1.   FM(hypergraph, partitioning)
2.   do
3.   tab initialize gain_container from partitioning;
4.   FMpass(gain_container, partitioning);
5.   while(solution quality improves);
6.   FMpass (gain_container, partitioning)
7.   solution_cost = partitioning.get_cost();
8.   while(not all vertices locked)
9.   move = choose_move();
10.  solution_cost += gain_container.get gain(move);
11.  gain_container.lock_vertex( move.vertex() );
12.  gain_update(move);
13.  partitioning.apply(move);
14.  roll back partitioning to best seen solution;
15.  gain_container.unlock_all();
```

1.1 The FM Heuristic

The Fiduccia Mattheyses (FM) algorithm works by prioritizing moves by gain. A move changes to which partition a particular node belongs, and the gain is the corresponding change to the cost function. After each node is moved, gains for connected modules are updated. The FM algorithm runs in passes wherein each node is moved exactly once. Passes are generally applied until little or no improvement remains. Initial solutions are often produced using a simple randomized algorithm.

1.2 Gain Computation

FM requests the best move from the gain container and can keep on requesting more moves until a move passes legality check(s). As FM applies the chosen move and locks the vertex, gains of adjacent vertices needs to be updated. In performing gain update, the FM must walk through all nets incident to the moving vertex and for each net computes gain updates (delta gains) for each of its vertices due to this net. These partial gain updates are immediately applied to the gain container, and moves of affected vertices may have their priority within the gain container changed.

1.3 FM Passes

During each pass, FM will search the neigh borhood of a given partitioning solution, and record the best seen solution. FM performs successive passes as long as the solution can be improved. At each pass, the FM repetitively chooses one move with the best gain and applies it. Since no vertex can be moved twice in a pass, no moves will be available beyond a certain point (end of a pass). Some best-gain moves may increase the solution cost, and typically the solution at the end of the pass is not as good as the best solutions seen during the pass. FM will then undo a given number of moves to yield a solution with best-seen cost.

2. Implementation on C++

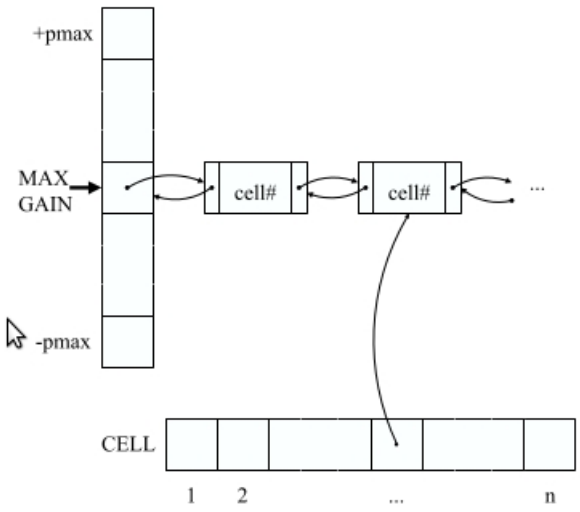

Figure 1: The gain bucket list structure.

2.1 Data Structures

The efficient custom data structures used by the FM algorithm are performance critical. The gain bucket list data structure introduced in [1] is necessary to allow linear-time gain update. Figure 1 shows the gain bucket list structure as originally illustrated. On the left is a structure called a bucket array which stores 2 pmax + 1 pointers to gain buckets. Each bucket is a possibly empty doubly-linked list of gain elements. On the bottom is a repository which stores pointers to the gain elements associated with each cell. The repository allows constant time lookup of the gain for a particular cell. To efficiently find the move with max-gain, the index of the highest-gain, non-empty bucket is maintained. The bucket array structure is efficiently implemented with an array when the magnitude of edge weights is limited by some constant. However, an array based implementation of the bucket array has space complexity dependent upon the magnitude of edge weights. For arbitrary edge weights, the bucket array must be modified in order to efficiently support the operations of a bucket array in the context of an FM gain container (Figure 2).

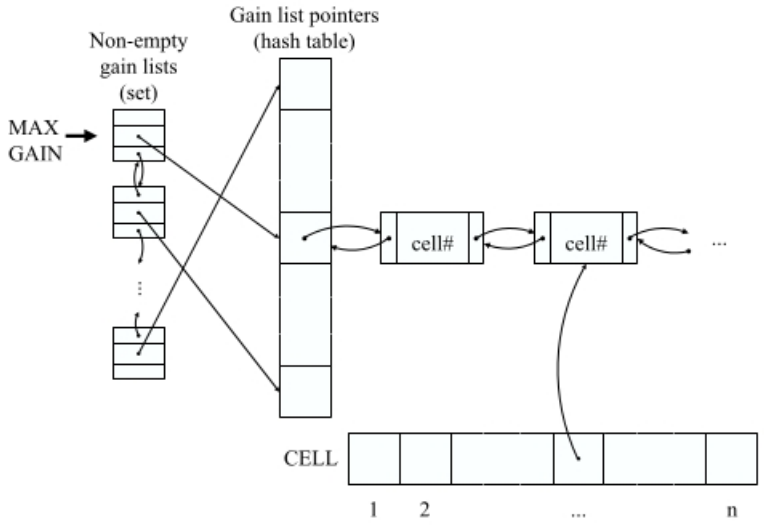

Figure 2: The gain list structure with bucket array for improved FM complexity with arbitrary edge weights.

4. Results

5. Acknowledgement

Our project is based on the work by David A. Papa and Igor L. Markov [2] We would like to thank our course instructor, Dr M.S Bhat for giving us an opportunity to work on this project.

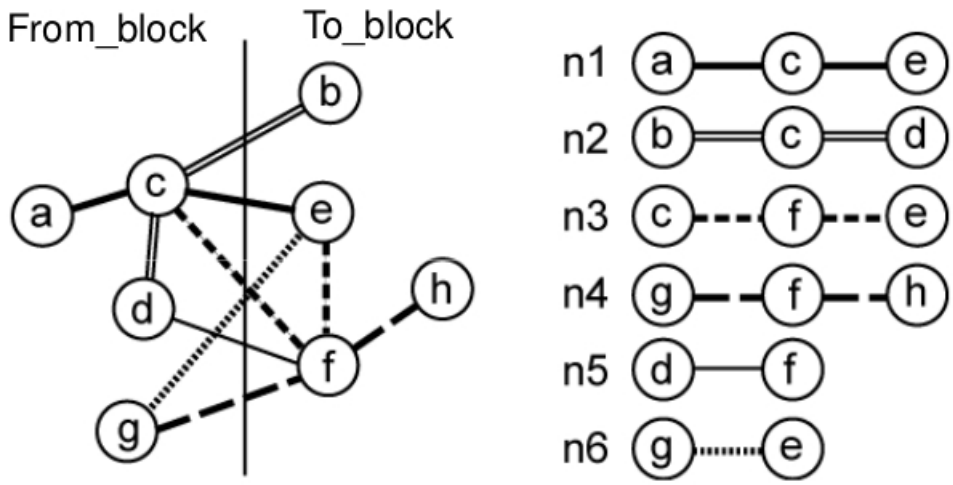

Figure 3: The initial configuration.

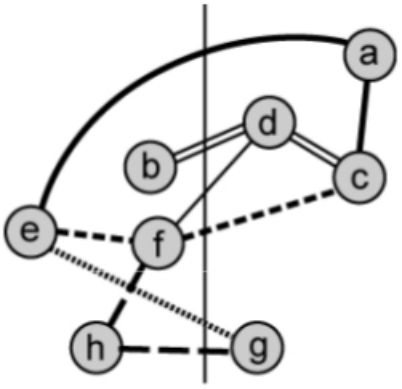

Figure 4: The result after one pass.

Bibliography

[1] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," DAC, 1982, p.175-181.

[2] David A. Papa and Igor L. Markov, "Hypergraph Partitioning," University of Michigan, EECS Department, Ann Arbor