Karlsruhe Institute of Technology

Smart Local Search in Hypergraph Partitioning

Diploma Thesis of

Orlin Kolev
Matr.-Nr: 1341048

At the Department of Informatics
Institute for Theoretical Informatics

Reviewer:	Prof. Dr. rer. nat. Peter Sanders
Advisor:	M.Sc. Sebastian Schlag

Duration: April 2017 – September 2017

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt habe. Alle verwendeten Quellen sind im Literaturverzeichnis aufgeführt.

Karlsruhe, den 30. September 2017

Abstract

Hypergraphs are a generalisation of graphs where each (hyper)edge can connect more than two vertices. Partitioning hypergraphs in disjoint, balanced blocks with low connectivity in between is a problem with many applications, for example VLSI CAD and scientific computing. Since it is NP-hard, it is solved using various heuristics. State of the art partitioners use a multilevel approach, in which the partitioning is done in three stages: coarsening, when vertices are contracted together in a way that preserves the structure of the graph, initial partitioning, and finally uncoarsening and refinement. During the refinement stage, a local search heuristic is used that improves the current partitioning each time after an uncontraction. The most commonly used heuristic for the local search is the Fiduccia-Mattheyeses algorithm. Various improvements have been made to it over time, but most of them focus on improving runtime. In this work we explore ways to improve the quality of solutions by modifying FM to take into account more information about the graph. We applied and modified some algorithms from the literature and developed a new one, integrating all into KaHyPar, a high-quality universal hypergraph partitioner. Our experiments showed a pronounced improvement against the original implementation of FM and our new method, Interval Soft Gain, achieves some of the best overall results and outperforms other approaches for some instance classes.

Zusammenfassung

Hypergraphen sind Generalisierungen von Graphen, bei denen eine (Hyper-)Kante mehr als zwei Knoten verbinden kann. Partitionierung von Hypergraphen in disjunkten, balancierten Blöcken mit schwacher Verbindung zwischeneinander ist ein oft auftretendes Problem mit zahlreichen Anwendungen, z.B. in VLSI CAD un wissenschaftliches Rechnen. Da es NP-schwer ist wird in der Praxis zu heuristischen Lösungen zugegriffen. Heutige Partitionierer verwenden ein Multilevel-Partitionierungsschema, das aus drei Hauptphasen besteht. Zuerst wird der Hypergraph verkleinert durch das Zusammenführen von Knoten, sodass die Struktur erhalten bleibt. Danach wird nach eine initiale Partitionierung gesucht. Zuletzt werden die zusammengeführte Knoten wieder expandiert und es wird dabei mittels eine lokale Suche die Lösung verfeinert. Die meistbenutzte Heuristik für die lokale Suche ist der Algorithmus von Fiduccia-Mattheyeses. Es sind zahlreiche Verbesserungen davon veröffentlicht, die sich meistens auf die Laufzeit beziehen. In dieser Arbeit untersuchen wir Methoden, die Qualität der Lösung zu verbessern, indem wir weitere Informationen über den Hypergraphen berücksichtigen. Wir haben Algorithmen aus der Literatur implementiert und angepasst, sowie einen eigenen entwickelt, und diese in KaHyPar integriert — ein universelles Partitionierungsframework für Hypergraphen. Unsere Experimente zeigen wesentliche Verbesserungen im Vergleich zu FM, und unser neuer Algorithmus, Threshold Soft Gain, erreicht einer der besten Ergebnissen, für manche Instanzklassen zeigt er sich sogar als der optimale.

Contents

1. Introduction

Hypergraphs are a generalisation of graphs in which an edge can connect more than two vertices — sometimes up to thousands. They are more flexible than graphs and allow the more natural representation of many systems. Consequently, applications can be found in many fieds within computer science, for example computer-aided chip design, efficient data storage, data mining [1], parallel computing [2] [3], machine learning [4], and more.

Hypergraph partitioning is a common problem for many applications of hypergraphs. Sometimes it arises as a standalone problem, for example in VLSI (Very Large Scale Integration), where it affects the quality of the final result, and in other cases it is a pre-processing step that allows splitting prohibitively large instances into manageable chunks, e.g. in SAT solving and scientific computing. Since hypergraph partitioning is an NP-hard problem [5], it is crucial to find effective heuristics that find good-quality solutions in reasonable time.

All commonly used partitioning tools use a multilevel approach, which consists of three phases: coarsening, initial partitioning and refinement. In the coarsening phase, vertices are merged together for the purposes of reducing the size of the hypergraph. To ensure a high-quality partition, this must be done in a way that preserves the topology of the hypergraph, i.e. the set of vertices that are eventually contracted into one should represent a densely connected component of the hypergraph. The result of the coarsening phase is a small hypergraph, which is then processed during the initial partitioning phase. It can use a multitude of approaches to find a good partition of the coarsened hypergraph. Due to its small size, it is possible to use algorithms that are too slow for the original hypergraph, or to try several different algorithms (or multiple times with a randomised algorithm) and proceed with the best partition. Effectively, due to the contraction of vertices, the initial partitioning phase partitions components of the hypergraph, each consisting of multiple vertices. During the final stage, the refinement, the contraction is undone in order to restore the original hypergraph. Local search is successively applied in order to improve the cut in the now more detailed hypergraph.

In order to produce a partition in k blocks, there are two main approaches — directly producing k partitions, or recursively bisecting the hypergraph until the desired number of partitions is achieved. Direct k-way partitioning has the potential to achieve better results, but is more complex.

Typically, designing a heuristic is a trade-off between generality and quality. For example, one of the best known partitioners for road networks heavily relies on some properties of the networks to find much better partitions than general-purpose algorithms [6], but it would fail on graphs that do not exhibit similar properties. The Karlsruhe Hypergraph Partitioner, KaHyPar, strives to manage the balancing act and provide fast and high-quality partitions for diverse hypergraph types. To do this, it combines many individual algorithms in each of the phases of the partitioning, which are enhanced, configured and tailored to seamlessly integrate together.

The goal of this work is to identify, develop and evaluate different ways to improve the local search phase of KaHyPar in recursive bisection mode. While the state of the art in terms of both runtime and solution quality is direct k-way partitioning, it is a lot more complicated, so we are using recursive bisection as a testbed for different methods so that the most promising ones can later be integrated in the k-way approach. Furthermore, recursive bisection is more universal, because k-way is not well suited for minimising the number of hyperedges in the cut, as opposed to the total blocks they connect.

1.1 Contributions

The main contribution of this work is identifying methods and parameters for them that alleviate some weaknesses of the Fiduccia-Mattheyeses local search algorithm, especially apparent on hypergraphs with large hyperedges. The Interval Soft Gain, described in section 4.4, is a new development which produces the best overall results, along with Lookahead FM from [7], which it outperforms for some instance classes. Unlike most work on hypergraph partitioning, which is closely related to certain applications, we use a diverse benchmark set consisting of hypergraphs with widely varying properties. We examine the strengths and weaknesses of different algorithms for different instance classes and suggest configurations that provide high-quality solutions in all cases. To the best of our knowledge, there have been no previous efforts to develop a universal local search algorithm, so we adapted methods from VLSI design and SAT solving to make them more robust.

1.2 Outline

The work is structured as follows: chapter 2 introduces concepts and notation used throughout the work; chapter 3 lists previous work on hypergraph partitioning, and introduces in detail the Fiduccia-Mattheyeses algorithm (section 3.1) and KaHyPar (section 3.3), which are central to this work. In chapter 4 we introduce the various improvement methods that we implemented. Chapter 5 describes the experimental results, with section 5.2 concerned with determination of optimal parameters for each algorithm, and section 5.3 compares the different methods with each other and the baseline. Finally, in chapter 6 we summarise the results and outline the future work on improving local search.

2. Preliminaries

This chapter introduces terminology and notation used later. It assumes two-way partitioning.

A *hypergraph* $H = (V, E, c, \omega)$ consists of a set of *vertices* V, a set of *hyperedges* E such that each $e \in E$ is a subset of V with $|c| \geq 2$, a vertex weight function $c : V \rightarrow \mathbb{R}$ and a hyperedge weight function $\omega : E \rightarrow \mathbb{R}$.

A vertex v is *incident* to a hyperedge e if $v \in e$. For a hyperedge e, the vertices incident to it are called its *pins*. For a vertex v, the number of its incident hyperedges is its *degree*, and the number of vertices in a hyperedge e, i.e. $|e|$ is the *size* of e. Two vertices are *neighbours* if there exists a hyperedge that contains both of them.

A *k-way partition* of a hypergraph $H = (V, E, c, \omega)$ is a partition of V in k *blocks* $V_1, ..., V_k$ such that $V_1 \cup ... \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for every $(i, j) : 1 \leq i, j \leq k; i \neq j$. For a parameter ϵ, a k-way partition is ϵ*-balanced* when $|V_i| \leq (1+\epsilon)\lceil \frac{|V|}{k} \rceil$ for every $i : 1 \leq i \leq k$. A block V_i is *overloaded* if $|V_i| > (1+\epsilon)\lceil \frac{|V|}{k} \rceil$, and *underloaded* if $|V_i| < (1+\epsilon)\lceil \frac{|V|}{k} \rceil$. Thus, an ϵ-balanced k-way partition does not have overloaded blocks. For a given k-way partition, a hyperedge e is *internal* to a block V_i if $e \subset V_i$. If a hyperedge e is not internal to any block, it is a *cut hyperedge*, and the set of all cut hyperedges of a partition is *the cut set* C of that partition.

In the context of a partition, for a hyperedge e and a pin $v \in e$, $T_e(v)$ denotes the number of pins of e that are not in the same block as v, and $\Theta_e(v)$ denotes the proportion of the pins in e, excluding v, that are not in the same part as v, i.e. $\Theta_e(v) = \frac{T_e(v)}{|e-1|}$.

A partition of a hypergraph can be assigned a cost. It is the metric that is the target for optimisation (minimisation) during partitioning. The most commonly used cost functions are *cut* and *connectivity*. Cut is the sum of the weights of the hyperedges in the cut (in the common case of unit weights, this corresponds to the number of cut hyperedges). Connectivity is the total number of blocks to which all hyperedges are connected, minus the number of hyperedges. For the cut metric a cut hyperedge is counted once, regardless of how many blocks it is connected to, whereas for the connectivity metric it also matters how many blocks it spans (the less, the better).

The *k-way partitioning problem* is to find an ϵ-balanced k-way partition of a hypergraph with minimal cost, with ϵ, k, and the cost function as parameters. It is NP-hard. [5]

Recursive bisection is a method of solving the k-way partitioning problem by repeatedly dividing the hypergraph in two blocks and then recursively dividing each of them until the desired number of partitions is achieved.

3. Related Work

In this chapter we introduce other publications related to hypergraph partitioning and local search.

3.1 Fiduccia-Mattheyses

The Fiduccia-Mattheyeses, or FM, is a widely used heuristic for the local search phase. It was introduced by [8].

A central concept in the FM algorithm is the *gain* of a vertex. It is defined as the (possibly negative) total weight of hyperedges that would be removed from the cut if the vertex were moved to the other block. If a vertex v is in block A, then a hyperedge e incident to v can be removed from the cut by moving v if and only if v is the only pin of e in block A. Let P_v be the set of hyperedges that satisfy this criterion. Conversely, e would be added to the cut by moving v from block A to block B if and only if $e \in A$. Let N_v denote the set of all edges incident to v that are entirely in the same block as v. Then $gain(v) = |P_v| - |N_v|$.

During a local search, the algorithm repeatedly selects the vertex with the highest gain and moves it to the other block, then updates the gain values of its neighbours, if necessary. The highest gain may be negative, so the move does not necessarily decrease the cut, but it is done regardless, in order to allow the algorithm to escape local minima. The vertex that has just been moved is *locked* in its new block and cannot be moved for the remainder of the pass, to prevent the algorithm from getting stuck moving one vertex back and forth and to ensure linear runtime of each pass. Choosing the highest-gain vertex and moving it is repeated until all vertices are locked. Finally, the state when the lowest cut was encountered is restored. This is one *pass* of the local search. If the latest pass has improved the cut, an additional one is made until no further improvements are achieved.

In order to satisfy the balance constraint, it is necessary to put some restrictions in the choice of the vertex to be moved. By definition of the balance constraint, a block is not allowed to have more than $(1 + \epsilon) \cdot \frac{|V|}{2}$ vertices, in which case it is overloaded. In case one block is overloaded, it is only permissible to move a vertex from that block to restore the balance between the blocks. If neither block is overloaded, the vertex with the highest gain of either block is chosen.

Choosing the vertex to be moved next can be performed efficiently by exploiting the fact that the gain values are limited to the interval $[-deg_{max}, deg_{max}]$, where deg_{max} is the maximal degree of a vertex in the hypergraph, and in practice often span only a small fraction of this interval. The vertices can be stored in a bucket queue which maintains a linked list of vertices for each possible gain value. In addition, an array points to the current location in the data structure of each vertex. Retrieving the vertex with the highest gain can be done in amortised constant time by removing an element from the first non-empty list. Changing the gain of a vertex can be done in constant time — it is accessed using the pointer in the vertex array, removed from the list where it is located, and inserted at the head of the list that corresponds to its new gain. Since locked vertices are not included in the lists of the bucket array, their gains can be kept track of in a separate list so that they can be re-inserted in the bucket array at the end of the pass in constant time.

The other major concern regarding the efficiency, in addition to choosing the vertex with the highest gain, is updating the gains of the neighbours. As a consequence of the definition of the gains, a hyperedge can contribute to the gain of a vertex only in two cases: if all its pins are in the same block, in which case its contribution is -1, and if there is one pin in one block and all its other pins are in the other block, then the contribution is 1 to the gain of the stray pin. [1] (A divided hyperedge of size two would have one "stray" pin in each block, so it would contribute positively to the gains of both its pins.) If a hyperedge has at least two pins in either block, it does not affect the gains of any vertices. After a move, it is sufficient to check for each hyperedge incident to the moved vertex if it has changed state between these three (positive, negative, and zero gain), and increment or decrement its pins' gains accordingly. In practice, this happens seldom to large hyperedges, and small ones are quick to update.

Each pass moves each vertex only once, and both choosing the highest gain vertex and updating its neighbours are done in constant time for realistic instances [8], each pass is completed in $O(n)$ time. The number of passes until no improvements are made is in practice also constant, although it is theoretically bounded by the number of hyperedges (because each pass until the last one removes at least one hyperedge from the cut).

3.2 Improvements on FM

As discussed in the previous section, a hyperedge contributes to the gain of its pins only if it is outside the cut or has only one pin in one of the blocks. While this reduces the number of gain updates that have to be made, it proves ineffective for improving the quality of the solution in some circumstances. In particular, large hyperedges with a significant number of pins in each of the blocks are difficult to remove from the cut, because it would take a lot of moves until they can have a non-zero contribution to the gains. If all hyperedges are in this state, there will be a lot of vertices with zero gain. FM will keep moving them around randomly, which is unlikely to yield significant improvements of the cut. Multiple approaches have been proposed to alleviate this situation. Some of them are listed below, and chapter 4 provides additional details about the algorithms and our implementation as part of KaHyPar (see section 3.3).

Mann and Papp [9] examine hypergraph partitioning as a preprocessing step for solving SAT instances, which commonly have large hyperedges and are vulnerable to the problem described above. They suggest the "Soft Gain" method, in this work referred to as "Threshold Soft Gain" (see section 4.3). It abandons the strict binding of the gain values to the decrease in cut and instead uses a definition of gain that encourages moving pins of a hyperedge to the part where the majority of its pins are. [9] also propose moving entire hyperedges across the cut, instead of individual pins. They also experiment with a

[1] Actually -1 and 1 are factors that are multiplied by the edge weight to get the final gain; for clarity we assume unit weight for all hyperedges.

relaxation of the locking of vertices, allowing them to be moved $k > 1$ times per FM pass, but this fails to yield significant improvement.

Cong et. al. [10] propose avoiding the aimless moving of vertices by prioritising neighbours of pins that have already been moved. In a situation where a lot of vertices have zero (or otherwise equal) gains, FM will choose a random one to move. The gains of its neighbours that are in the same part where it has been moved from are then increased to encourage following up this move with others in the same direction, eventually moving all pins of some hyperedges across. These gain bonuses depend on the past moves done within the current pass and reset at the beginning of the next one. Section 4.1 describes our implementation of this approach.

Krishnamurthy [7] keeps the strict binding of cut change to gains, but examines potential gains a few moves ahead. Thus, the gain becomes a vector, its first element being the same as the gain in the original FM algorithm. In case of a tie, the second-level gain values decide which vertex is to be moved next, and so on.

3.3 KaHyPar

KaHyPar [11] is a universal partitioner, designed to provide high-quality partitions regardless of use case.

It supports two methods of partitioning — recursive bisection and direct k-way partitioning, as well as two objectives — cut and connectivity. While it is possible to use direct k-way partitioning to optimise cut, it is designed with the connectivity metric in mind, whereas the recursive bisection is suitable for both. During a single bisection it is irrelevant what the current metric is. When there are only two blocks the values for cut and connectivity are identical, because a hyperedge that is in the cut set is inevitably connected to exactly two blocks and contributes with 1 to the connectivity metric. The difference between the two is in between bisections. If cut weight is used, hyperedges that are in the cut set after a bisection cannot be subsequently removed from it, so they can be removed from consideration altogether (deleted or deactivated). If the objective is connectivity, then each cut hyperedge must be split in two — one for each block of the bisection — to reduce the number of blocks it is spanning in the subsequent bisections.

A distinguishing feature of KaHyPar is that it takes the multilevel approach to the extreme and contracts or uncontracts one vertex at a time. This fine-grained strategy has the potential to achieve very high-quality partitions, but care must be taken to keep the runtime at an acceptable level. For this reason KaHyPar uses numerous techniques for performance optimisation, such as adaptive stopping rules, gain cache and gain updates during uncoarsening. It is essential to adapt and use them as much as possible in any modification of the local search, because for the n-level approach it is invoked a lot of times and can easily become a performance bottleneck.

For more details the reader is referred to [11]; we will describe here the optimisations that concern the local search — the limited scope of the search and the gain cache.

The local search is invoked after uncontraction of a single vertex and there is only a small part of a hypergraph that has changed since the last local search. Thus, it would be wasteful to initialise it with all vertices, as done in the traditional FM algorithm. Instead, for KaHyPar it starts with the just uncontracted vertex and the one it has been uncontracted from. In order to keep the search in the affected area, KaHyPar uses a stopping rule that terminates it after certain conditions are met, e.g. a certain number of moves have been performed without any improvements in the cut.

Since each local search visits only a fraction of the hypergraph, it is unnecessary to perform a tedious recomputation of the gains of all vertices. In fact, KaHyPar minimises recomputation of gains by maintaining a *gain cache*. Whenever a vertex is moved, its gain must be recomputed, but the gains of its neighbours can be adjusted with increments or decrements as appropriate. These changes are reflected in the gain cache so that it remains up to date and recomputation of the neighbours' gains can be avoided. Note that it is also necessary to update the values in the priority queue for active vertices.

3.4 Other Partitioning Tools

In this section we list other partitioning tools and their distinguishing features.

- PaToH (Partitioning Tools for Hypergraph) is one of the first multilevel partitioning tools. It is focused on efficiency and is one of the fastest sequential partitioners; [12]
- hMETIS is the hypergraph partitioner of the METIS partitioning family, coming from the VLSI application and producing high-quality partitions for such instances; [13]
- the Mondriaan partitioner is designed for partitioning a rectangular sparse matrix for parallel sparse matrix-vector multiplication; [14]
- MLPart is hypergraph partitioner for the min-cut metric; [15]
- Zoltan is a toolkit providing algorithms for partitioning, among other tasks, intended for load-balancing of parallel applications; [16]
- Parkway is a parallel multilevel partitioner; [17]
- UMPa is a multi-objective partitioner from the scientific computing domain. [18]

4. Improvements on Local Search

In the FM algoritm, the gain of a vertex is defined as the exact reduction in cut that would be achieved when moving it. The heuristic assumption is that making the locally optimal move at each step (i.e. moving the vertex with the highest gain) will result in a good solution globally. All methods presented here except Lookahead FM go a step further and replace or extend the cut change with a different heuristic each. Lookahead FM keeps the relation between gain and cut, but examines the potential gains for several moves.

Common for all methods and the original FM is that for each vertex, the gain is the sum of the *gain contributions* of all its incident hyperedges, i.e.

$$gain(v) = \sum(cont(e, v) : e \text{ incident to } v)$$

$cont(e, v)$ istelf breaks down in a gain factor multiplied by the edge weight. For clarity, in the remainder of this chapter we assume all edge weights are 1.

In FM, $cont(e, v)$ can be -1, 1, or 0, depending on whether moving v would add, remove, or keep e in the cut, respectively. Soft Gain, Threshold Soft Gain and Interval Soft Gain replace this function with heuristic ones that are more fine-grained. They are no longer related to what would be immediately “gained” by moving a vertex, hence the name “soft gain”. Loose Hyperedge Removal retains the original gain function, but augments it with temporary additions during the course of each local search pass. Of the algorithms that use different gain functions, Soft Gain uses a fixed, fine-grained one, whereas Interval Soft Gain and Threshold Soft Gain employ customisable intervals and gain additions, each in a different way.

4.1 Loose Hyperedge Removal

In the course of the multilevel partitioning algorithm, the local search is invoked every time a vertex is uncontracted. During each local search, a vertex can only be moved once. A vertex that has not been moved in the current search is called *free*, whereas one that has been moved is *locked.* For hyperedges, there are three cases:

- no pins of the hyperedge are locked, in which case it is *free*;
- the hyperedge has locked pins, but they are all in the same block — in this case, the hyperedge is *loose*, and *anchored* to the block where its locked pins are, which are themselves called *anchors*;

- a hyperedge is *locked* if it has locked pins in both blocks.

A locked hyperedge cannot be removed from the cut during the current local search pass. A loose hyperedge, on the other hand, is a suitable candidate for being completely moved to the block where it is anchored, since some of its pins have already been moved and it is appropriate to increase the priority of its other pins. This approach has been used in [10].

In the unmodified FM algorithm, the gain values of the vertices are used as a heuristic to decide which ones to move (in addition to describing the exact change of the cut). In the loose hyperedge removal approach, a “loose hyperedge bonus” is added to the gain of each vertex to determine its priority for moving. The original formula for the gain increases proposed in [10] is as follows:

$$incr(e) = \left\lfloor \frac{MAX\_SIZE}{|e|} \cdot \frac{\sum_{c \in L_e} deg(v)}{\sum_{c \in F_e} deg(v)} \right\rfloor$$

where $incr(e)$ is the value added to the gain, MAX_SIZE is the size of the largest hyperedge in the hypergraph, L_e is the set of locked pins of e, and F_e is the set of free pins of e that are in the part without locked pins. $deg(v)$ denotes the degree of a vertex v. This function favours:

- small hyperedge size;
- strong connectivity of the hyperedge in its anchor block;
- weak connectivity in the other block.

The first term of this function, $\frac{MAX\_SIZE}{|e|}$ is meant to prioritise smaller hyperedges. If a small hyperedge becomes loose (after it has been free and one pin has been moved), it takes less moves to remove it from the cut than a large one. However, the size of the hyperedge is in part implicitly included in the second term — large hyperedges have many pins and the sum of their degrees is correspondingly greater. Initially, L_e is empty, and as pins get added to it one by one, the second term increases faster for small hyperedges than large ones. To avoid over-prioritising small hyperedges, we introduce a variation that uses only the second term and relies on its implicit affinity to small hyperedges:

$$incr_1(e) = \frac{\sum_{c \in L_e} deg(v)}{\sum_{c \in F_e} deg(v)}$$

Note that our function does not round the result. Cong et. al. use a bucket queue for their implementation, which requires integer-valued added gains, but we use a regular heap that can handle floating point.

We also introduce a hybrid variant that does explicitly favour small hyperedges, but without referring to the maximal hyperedge size:

$$incr_2(e) = \frac{1}{|e|} \cdot \frac{\sum_{c \in L_e} deg(v)}{\sum_{c \in F_e} deg(v)}$$

It offers three potential advantages:

- it puts stronger emphasis on small hyperedges than $incr_1$;
- it does not refer to the maximal edge size, which potentially makes it more robust than $incr$;
- its values are smaller, giving higher priority to actual cut changes over loose hyperedge additions.

It is important to note that the loose hyperedge bonus has a fundamental difference from the gain. While the gain depends only on the current state of the hypergraph, the loose hyperedge bonus also depends on the state of the current local search pass. This means that the loose hyperedge bonuses must be reset in between local search invocations, because the new invocation starts with all vertices and hyperedges in the free state. This necessitates keeping track of the loose hyperedge bonus separately from the gain. While this incurs a small overhead, it is advantageous that the gain values remain bound to the actual cut change, so there is no need to explicitly compute the cut change after a vertex is pulled from the priority queue.

4.2 Soft Gain

The FM algorithm works directly with the cut improvements, thus failing to "notice" hyperedges that are "almost" completely in one part, but still have two or more vertices in the other. Moving each of these vertices is insufficient to remove the hyperedge from the cut, so they will have the gain value of 0. Heuer [19] proposes an alternative gain function that determines the gain based on the distribution of the vertices of a hyperedge between blocks.

In the FM algorithm for a vertex v and a hyperedge e:

$$cont_{FM}(e,v) = \begin{cases} -1\text{, if moving } v \text{ would add } e \text{ to the cut set} \\ 1\text{, if moving } v \text{ would remove } e \text{ from the cut set} \\ 0 \text{ otherwise} \end{cases}$$

The above gain contibution function can be expressed in terms of $\Theta_e(v)$:

$$cont_{FM}(e,v) = \begin{cases} -1 \text{ if } \Theta_e(v) = 0 \\ 0 \text{ if } \Theta_e(v) \in (0,1) \\ 1 \text{ if } \Theta_e(v) = 1 \end{cases}$$

The *soft gain* function is an interpolation of the above in the interval in which $cont_{FM}(e,v)$ is 0:

$$cont_{SG}(e,v) = 2\Theta_e(v) - 1$$

Recall that $\Theta_e(v)$ is the fraction of the pins of e, excluding v, that are not in the same part as v. $cont_{SG}(e,v)$ is effectively a linear mapping of the interval of $\Theta_e(v)$, which is $[0,1]$, to $[-1,1]$, which is the interval of the gain contribution. The purpose is to avoid the situation when no moves could change the cut — in this case FM performs random moves, whereas the soft gain method prioritises edges that could potentially be removed from the cut in several moves.

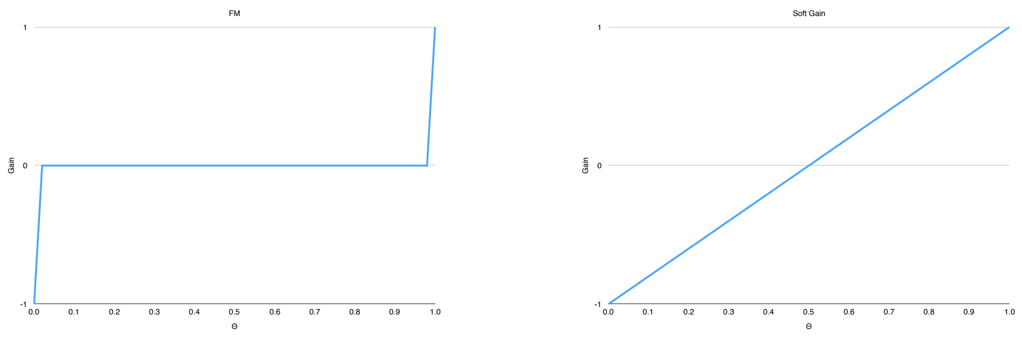


Figure 4.1: The FM (left) and Soft Gain (right) gain contribution functions.

4.3 Threshold Soft Gain

The Threshold Soft Gain method is suggested by [9] as an improvement to FM. The aim is to effectively partition SAT formulas, the corresponding hypergraphs of which have large hyperedges and vertices with high connectivity, so the unmodified FM typically encounters a situation where most gain values are zero and cannot find reasonable moves. To solve this problem, they devised a gain function that is not bound to the decrease in cut, but also favours moves that are likely to result in future increases in the cut.

The algorithm uses two parameters: a list of thresholds $0.5 < t_1 < t_2 < ... < t_l = 1$ and corresponding gain values $0 < g_1 < g_2 < ... < g_l = 1$. The thresholds refer to the proportion of the pins of a hyperedge that are in the other block, relative to a particular vertex v. If moving v to the other part would result in the proportion exceeding a threshold, the gain of v is increased by the corresponding gain factor. If v is in the block with the majority of pins, then it gets a gain decrease if moving it to the other block would result in the proportion of the majority sinking below a threshold. Formally, for a vertex v and hyperedge e, let $\delta_e = \frac{1}{|e|-1}$ denote the change of Θ_e when one vertex is moved. If $\Theta_e(v) < t_i$ and $\Theta_e(v) + \delta_e \geq t_i$, the gain of v is increased by g_i for every i that satisfies this criterion. Conversely, if $\Theta_e(v) \geq t_i$ and $\Theta_e(v) - \delta_e < t_i$, it is decreased by g_i.

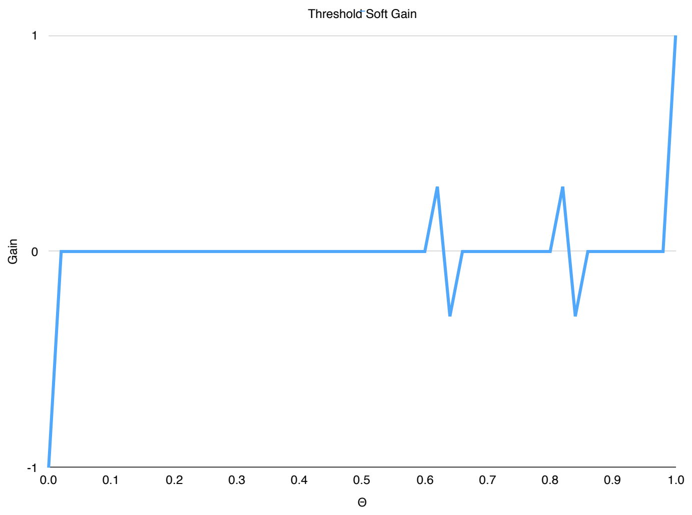
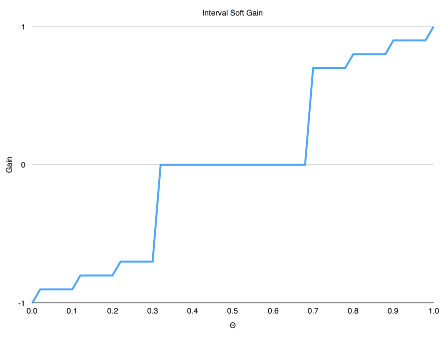


Figure 4.2: The Threshold Soft Gain gain contribution function.

4.4 Interval Soft Gain

The threshold soft gain, described in the previous section, has a disadvantage that is obvious from the gain plot. It is possible that $t_i < \Theta_e(v) - \delta_e < \Theta_e(v) + \delta_e < t_{i+1}$, i.e. the proportion of pins in the other part falls in between two thresholds, such that neither adding nor removing a vertex would trigger crossing a threshold. In this case there are no gain additions. To avoid this situation the thesholds could be placed densely, but this would alter the behaviour with small hyperedges where moving one vertex would result in crossing multiple thresholds and corresponding excessive gain increases. In order to make it possible to set the thresholds so that they work effectively for hyperedges of all sizes, we developed Interval Soft Gain.

We define a vector of thresholds $(0 = t_0 < ... < t_l = 1)$ that mark the interval borders, and interval gains $(-1 = g_0 < ... < g_l = 1)$, with g_i being the (possibly negative) gain addition for vertices for which $\Theta_e(v) \in [t_i, t_{i+1})$ (i.e. g_i refers to the interval which starts at t_i). Since $\Theta_e(v) \in [0, 1] = [t_0, t_l]$, there always exists $i \in [0, l]$ with $t_i \leq \Theta_e(v)$. The thresholds span the entire interval of possible values for $\Theta_e(v)$ and cover the "hard gain" cases from the original FM: when $\Theta_e(v) = 0 = t_0$, the "active" gain addtition is $g_0 = -1$, and when $\Theta_e(v) = 1 = t_l$, it is $g_l = 1$. In the configurations that we used we always restricted the +1 and -1 gain values to the cases when there is a change in cut and used smaller gain additions in the other cases, i.e. for every $i \in [1, l-1] |g_i| < 1$. Further, it was necessary to set $t_1 < \frac{1}{MAX\_SIZE}$ and $t_{l-1} > 1 - \frac{1}{MAX\_SIZE}$ to avoid situations where the gain is 1 despite there being more than one vertex on the current side, or the gain is -1 even though there are still one or more vertices on the other side.



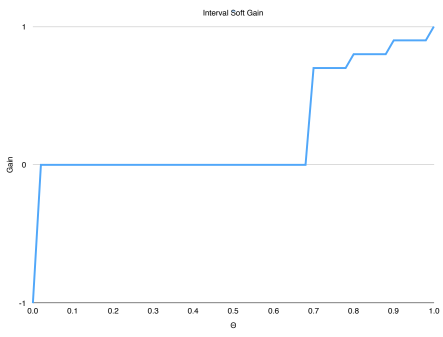


Figure 4.3: The Interval Soft Gain gain contribution function, symmetric (left) and asymmetric (right).

4.5 Lookahead FM

Lookahead FM by [7] aims to avoid the situation when FM executes aimless moves because of equal gains while retaining the strict relation of the gain values to the cut change. To do this, it keeps track of higher-level gains which describe the potential cut change after several moves in the same direction. For example, if a hyperedge has two vertices in one part and more than two in the other, it can be removed from the cut by two moves. Assuming these two vertices will not affect any other hyperedges' cut state, they will both have zero gains, possibly among many other in the hypergraph. FM would overlook them and would move them only if one gets randomly selected; the other one would then have its gain increased and thus its priority for moving as well. Lookahead FM is designed to recognise that moving one of these vertices would result in the gain of the other one being increased and prioritises such moves.

Specifically, the gain of a vertex is a vector $(g_1, g_2, ..., g_l)$. A hyperedge contributes positively to the i-th level gain of a vertex if i moves of a pin from the block of that vertex to the other would result in the hyperedge being removed from the cut, i.e. if it has exactly i pins in that block. In the event of a pin move in the opposite direction, the hyperedge will no longer contribute positively to the i-th level gain. To discourage such moves, the hyperedge has negative contribution to the $(i+1)$-th level gain of vertices on the opposite side. Formally, for a hyperedge e and a pin v, if V denotes the block v is in, $\overline{V}$ the opposite block, the contribution of e to the i-th level gain of v is:

$$cont_i(e,v) = \begin{cases} -1 \text{ if } |e \cap \overline{V}| = i-1 \text{ and } |e \cap V| > 0 \\ 1 \text{ if } |e \cap V| = i \text{ and } |e \cap \overline{V}| > 0 \\ 0 \text{ otherwise.} \end{cases}$$

Note that g_1 is exactly the gain of the original FM. Analogously to it, the gain of a vertex is the sum of the contributions of all its incident hyperedges, this time on a per-component basis:

$$gain(v) = (\sum_{e \in E_v} cont_1(e,v), \sum_{e \in E_v} cont_2(e,v), ..., \sum_{e \in E_v} cont_l(e,v))$$

where E_v denotes the set of hyperedges incident to v.

In the gain priority queue, the vector gains are compared lexicographically, so that the i-th level gain for $i > 1$ only serves to break ties at level $i-1$. The size of the vector, i.e.

the lookahead depth l, is set as a parameter at the start of execution. Having higher-level gains necessitates more frequent gain updates — every time a hyperedge remains with l or less pins in one part, as opposed to 1 in the original FM.

5. Experiments

In this chapter we describe our experimental results. In section 5.2, we compare different configurations of each algorithm to determine its most suitable parameters. Soft Gain is not included, because it has no configurable options. In section 5.3, the best configurations of each algorithm are compared against each other.

5.1 Experiment Setup and Instances

The experiments were performed on the bwUniCluster, part of the bwHPC project of the German state of Baden-Württemberg. The partitions were run on compute nodes with Intel Xeon E5-2670 (Sandy Bridge) processors clocked at 2.6 GHz, with 64 GB main memory. Since the runtime is out of the scope of this work, we ran 16 instances of KaHyPar at a time, to increase throughput.

We used a small test set consisting of 25 hypegraphs for tuning, and a large one with about 150 hypergraphs for the final evaluation. The hypergraphs come from several application fields:

- VLSI — we used hypergraphs from the ISPD98 Circuit Benchmark Suite [20] and the ISPD-2011 routability-driven placement contest [21]. In general, these instances have a large number of vertices and a large number of predominantly small hyperedges (of 2-5 vertices);
- various sparse matrix instances, randomly chosen from [22]. Each column of the matrix is a vertex, each row is a hyperedge, and the vertex and hyperedge are incident if the corresponding matrix entry is non-zero. These hypergraphs are primarily from the scientific computing field and are used to optimise performance. The balance criterion for the hypergraph partitioning ensures equal load distribution between computing nodes, and the minimisation of the cut or connectivity corresponds to reducing communication between nodes. These instances tend to have higher vertex degrees and hyperedge sizes in comparison to the VLSI ones, making them more challenging to partition;
- SAT instances, randomly chosen from [23]. These come in three variants: *primal*, *literal* and *dual*. In the primal hypergraph of a formula, each variable is represented with a vertex and each clause with a hyperedge. In the literal hypergraphs, a variable is represented by two vertices — one for the variable with and one without negation.

In the dual hypergraphs, the variables are represented with hyperedges and the clauses with vertices. The dual hypergraphs are a major challenge for FM-based partitioners, because they have a large number of vertices and comparatively few, but very large hyperedges. Details on the role of hypergraph partitioning in SAT solving are given in [9].

Each hypergraph was partitioned in $k \in 2, 4, 8, 16, 32, 64, 128$ blocks with balance constraint of $\epsilon = 0.03$ ten times with different random seeds. The average results for all seeds are used for each (hypergraph, k) combination, which constitutes one *instance*.

In presenting the results, we use the performance plot format introduced in [11]. (See figure 5.1 for an example plot with explanation.) The plots are produced as follows:

1. for each instance, choose the best result for each algorithm out of the ten runs;
2. for each instance and algorithm, calculate 1-(best/result) (the y-value), where "best" is the best result on that instance by any algorithm, and "result" is the result of the current one. A value of 0 means that algorithm produced the best result; the greater the values, the worse the performance relative to the best one. In this way the dots that are plotted are obtained.
3. for each algorithm separately, the instances are sorted in descending order by their y values. This means that two dots on the plot that are directly above each other usually do not refer to the same instance; rather, they refer to the same percentile of the sorted relative results for each algorithm.

Both axes are presented on a cube root scale in order to emphasise the lower-left part of the plots, which most clearly shows the differences between the algorithms. While the format allows for an arbitrary number of algorithms to be compared at once, we prefer comparing two at a time for clarity; when the best results for different instances are distributed between more algorithms, it is more difficult to draw conclusions from the plots.

5.2 Tuning Experiments

5.2.1 Loose Hyperedge Removal

The Loose Hyperedge Removal method with the original formula from [10] fails to demonstrate any overall improvement in quality on our benchmark set, with the exception of the sparse matrix instances (see figure 5.3). They have relatively large hyperedges that are prone to the scenario with many zero gains and the algorithm successfully escapes the situation by making a random initial move and "persevering" by subsequently moving neighbouring vertices, thus moving entire hyperedges to one block and removing them from the cut.

The algorithm performs very poorly on the ISPD instances. The reliance on the maximal hyperedge size for them proves inappropriate, because in most instances the median hyperedge size is 2, the 75-th percentile is about 4, and the maximum is about 40. This means that the first term of the formula, $\frac{MAX\_SIZE}{|e|}$ is usually in the range 10-20. Since vertex degrees are usually less than 10 and gains even less, the loose hyperedge additions are much greater than the gain values and the local search starts moving vertices with little regard to the change of cut.

To rectify the situation we attempt removing the reference to the maximal hyperedge size, and also removing the size as an explicit factor altogether. Each of these measures "saves" the performance on the ISPD instances, with little changes in the other instance classes. For the dual SAT instances, performance is considerably better if the size factor is included. On primal and literal SAT hypergraphs there is no improvement over the standard FM in any of the Loose Hyperedge Removal configurations.

Overall, the best performance over our tuning set is achieved with the formula with explicit hyperedge size factor, but no reference to the maximal hyperedge size:

$$incr_2(e) = \frac{1}{|e|} \cdot \frac{\sum_{c \in L_e} deg(v)}{\sum_{c \in F_e} deg(v)}$$

It proves to be the most robust and achieving the best balance between loose hyperedge gain additions and “natural” gains.

5.2.2 Interval Soft Gain

A straightforward configuration of Interval Soft Gain is one that uses a similar gain contribution function to that of Soft Gain. It is split in discreet intervals, which means that gain updates have to be made only if $\Theta_e(v)$ crosses an interval border. In addition to potentially reducing runtime, this also produces noticeably better results than Soft Gain. This is presumably due to the fact that our coarse-grained intervals left the gain contribution at 0 for the middle 20% of the range of $\Theta_e(v)$, i.e. for hyperedges that are within 10% of being evenly distributed between blocks. For them it is unclear to which block they should be moved and assigning them non-zero gain contributions may be counterproductive. To test this hypotheses, we increased the neutral interval, assigning non-zero gains only to hyperedges that are at least 80% in one block. This dramatically improved the quality of partitions. Figure 5.5 shows the improvements of the linear variant (emulating Soft Gain) and the end-only one (with the large neutral interval in the middle).

In FM, the gain of vertices is symmetric, because it is bound to the cut — a hyperedge has a positive gain contribution if it can be removed from the cut set and a negative if it would be added to it. Soft Gain retains this symmetry by using gains in the interval $[-1, 1]$. A disadvantage of this approach can be illustrated with the following example: a large hyperedge has 5% of its pins in one block and 95% in the other. A symmetric soft gain function, such as that from the Soft Gain method and its emulation with Interval Soft Gain, would assign penalties to these 95% of pins. This is a large number of pins and some of them are likely to be suitable candidates for moving, because they could help bring other hyperedges out of the cut. A symmetric gain function would assign them considerable penalties that could prevent them from being considered. The situation worsens for vertices which are incident to several hyperedges in this situation. The high flexibility of Interval Soft Gain allows us to avoid this problem by defining an asymmetric gain function that reserves the negative gain contribution strictly for the situations where the cut would be increased (i.e. when the unmodified FM would also assign negative gain conributions) and only give positive “soft gains”. In this case, the 5% of pins of our example hyperedge will receive increased priority, but the remaining 95% will not be restricted from moving. Figure 5.6 shows that the asymmetric approach outperforms the symmetric one.

5.2.3 Threshold Soft Gain

There are two primary concerns regarding the parameters of Threshold Soft Gain. If the thresholds are too sparse, it will be ineffective for large hyperedges, because most possible moves would be unaffected by the thresholds. If they are too dense, moves in smaller hyperedges would cross multiple thresholds at once, resulting in excessive gain increases which give priority to otherwise impractical moves. If opting for dense thresholds, they should be placed towards the end of the interval — for hyperedges that are distributed relatively evenly between blocks, it is unclear which direction is more prospective.

To determine the optimal density of thresholds, we experimented with differences between them of 0.1, 0.5 and 0.25. The results show that densely spaced thresholds at the end of the interval perform better and outperform the baseline. Despite the improvement over FM, this method performed worse than the related Interval Soft Gain, so we did not include it in the final evaluation.

5.2.4 Lookahead FM

The Lookahead FM algorithm has a single tuning parameter — the size of the vector, which corresponds to how many future vertex moves are considered. The maximal useful depth is limited depending on the properties of the hypergraph. Krishnamurthy [7] suggests a formula for it, which for our instances evaluates to up to 8. The reason that increasing the size further is not expected to yield improvement is that the gain vectors are compared lexicographically, so if there are different values at one level, further levels are obsolete. At greater vector lengths it becomes increasingly unlikely that two vectors would be identical. An advantage of using shorter vectors would be lower runtime, but since in this work we are only investigating the quality of results, we use the highest value for comparison with other methods. Figure 5.8 shows the difference between depths 2, 4 and 8 over the entire tuning benchmark set. Figure 5.9 shows the difference between depths 2 and 8 for different instance classes. It is rather low for the densely connected SAT dual hypergraphs where the method fails to yield considerable improvements, and the ISPD instances, which tend to have small hyperedges for which even low lookahead depth is sufficient. The greatest difference is in the sparse matrix instances.

5.3 Evaluation Experiments

In this section we evaluate the best configuration of each method over the large instance set. The configurations we used are the following:

- Soft Gain — as-is; it has no configurable parameters;
- Loose Hyperedge Removal — the $incr_2$ function which explicitly favours small sizes, but independently of the maximal hyperedge size;
- Interval Soft Gain — the asymmetric end-heavy function;
- Lookahead FM — depth 8.

We did not include Threshold Soft Gain in this section, because it is similar to Interval Soft Gain, but performs slightly worse.

Our experiments show that the soft gain approach in general fails to improve on FM. Its results are very close to those of FM, except in the sparse matrix instances, where Soft Gain is considerably worse, and in the primal SAT hypergraphs, where it has an advantage. Remarkably, only Soft Gain succeeds in improving the primal instances — it outperforms FM a lot more than for the similar literal instances.

Overall, Soft Gain distributes the gains too evenly, instead of promoting only prospective candidates, which prevents it from bringing substantial improvements. Interval Soft Gain (see section 4.4 for description) uses a similar method, but with a customisable gain distribution, which can overcome the problems inherent to Soft Gain.

The results from comparing each of the other methods to the baseline are qualitatively identical to what we observed during tuning, so we will proceed with the comparison of them with one another. Figure 5.11 shows a close race, with Soft Gain producing the best results for the literal instances and either Lookahead or Interval Soft Gain leading each of the other instance classes. A direct comparison between the two (figure 5.12) shows an advantage of Interval Soft Gain in the SAT primal hypergraphs and even more so in the ISPD instances, but weaker performance on the sparse matrix and SAT literal instances. Overall, they are very close to each other.

Lookahead FM and Interval Soft Gain both attempt to predict cut changes that would happen in future moves, each in a different way. Lookahead FM is stricter in that it works with the actual cut changes and strictly prioritises lower-level gains, not allowing a very high second-level gain to override a low first-level gain. Interval Soft Gain, on the other hand, is fuzzier. A vertex may have a high priority purely because of a large number of small gain bonuses adding up, and it could preemmpt an actually prospective vertex. The disadvantage of Lookahead FM is that it works with a fixed number of levels, which is not helpful for extremely large hyperedges. Interval Soft Gain copes better with this situation.

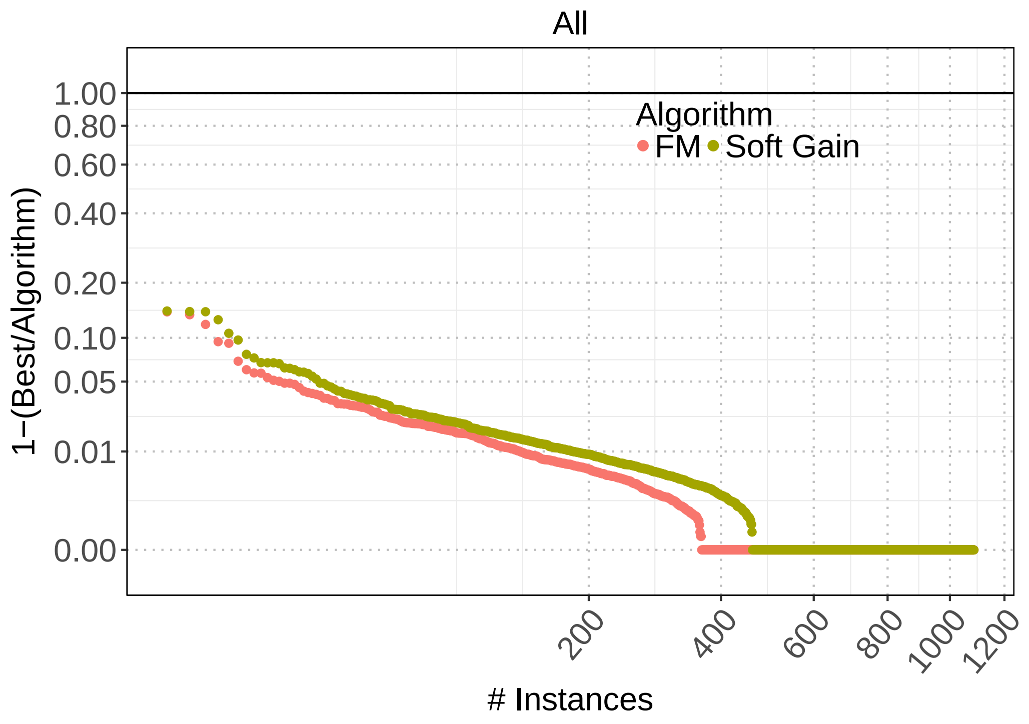


Figure 5.1: An example performance plot. There were a total of about 1100 instances, which is indicated by the x-value to which the graph extends. The right portion of the plot, where the graphs lie on the x-axis shows the instances for which each algorithm produced the best results. The "FM" algorithm did in about 720 cases (it is aligned to the axis approximately between 380 and 1100) and the "Soft Gain" algorithm on about 620 (which means that both algorithms produced identical results in about 240 cases). Note that the two graphs overlap in the lower right portion of the plot and one occludes the other. The left part of the plot shows how much worse each algorithm is, relative to the optimum for that instance. In this example, each algorithm is up to 15% worse than the other for a small number of instances and withtin 5% for most.

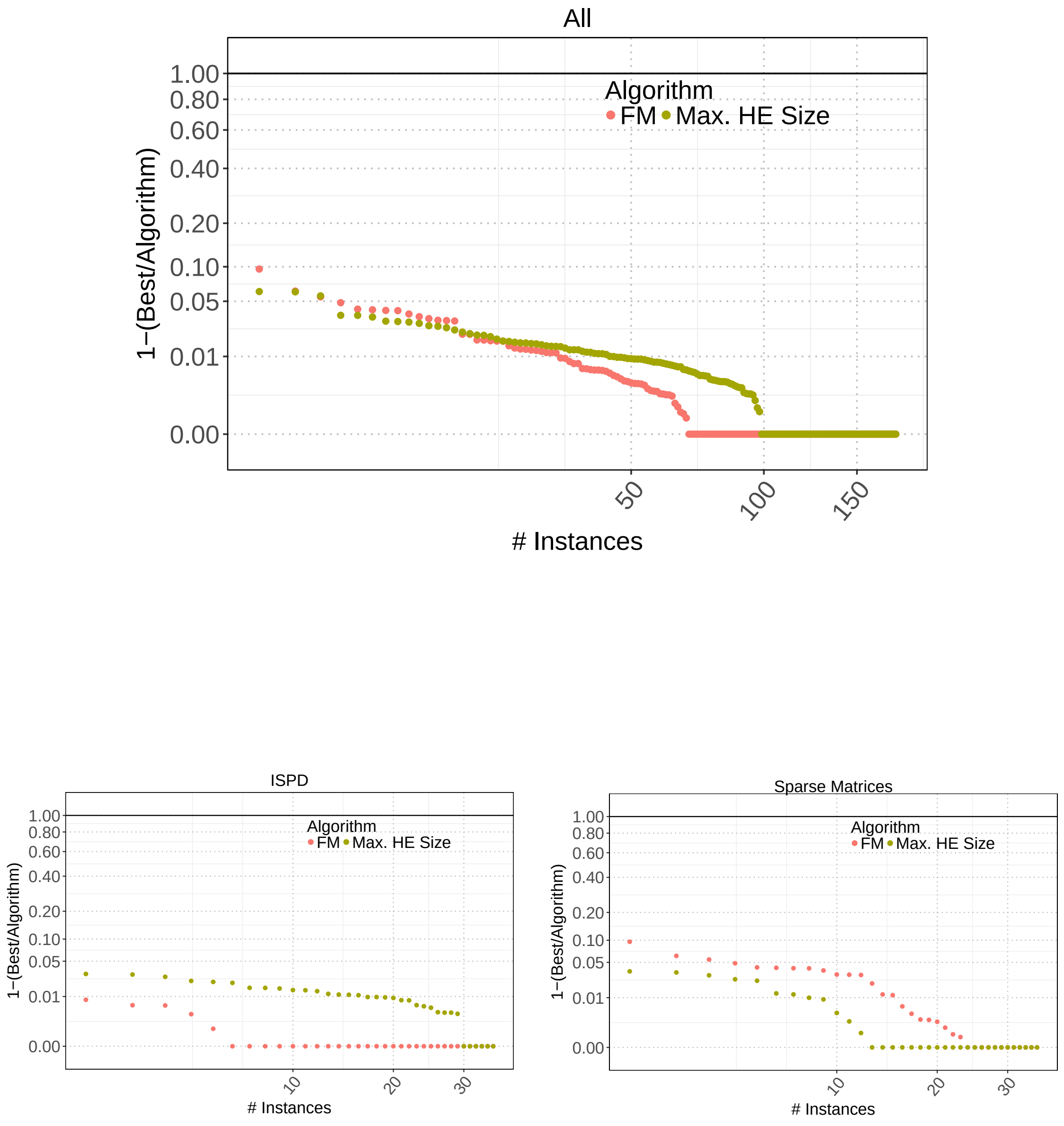


Figure 5.2: Comparison of FM with the Loose Hyperedge Removal algorithm as proposed in [10], over the entire tuning set, the ISPD instances and the sparse matrix instances.

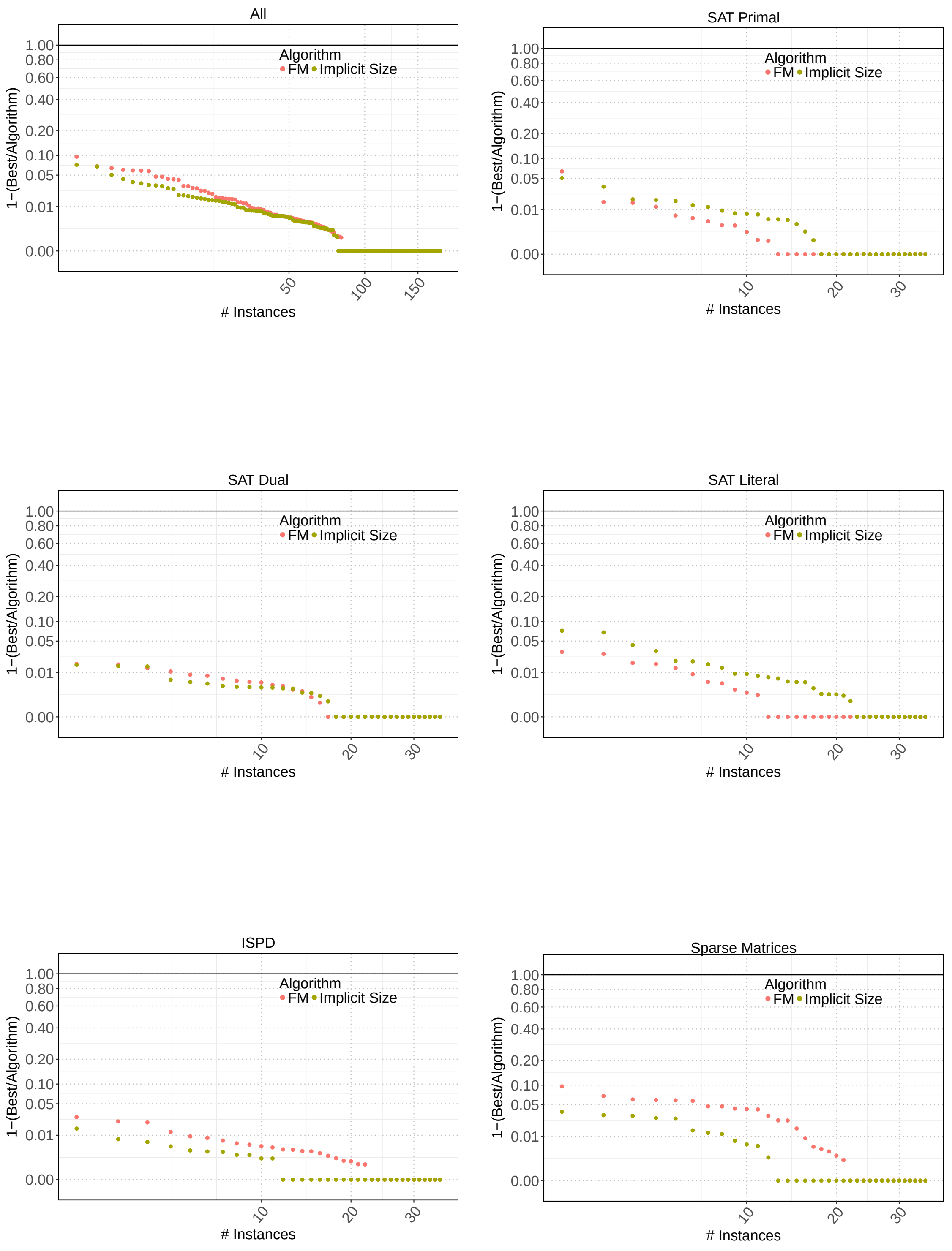


Figure 5.3: Performance plots comparing Loose Hyperedge Removal without size factor and FM

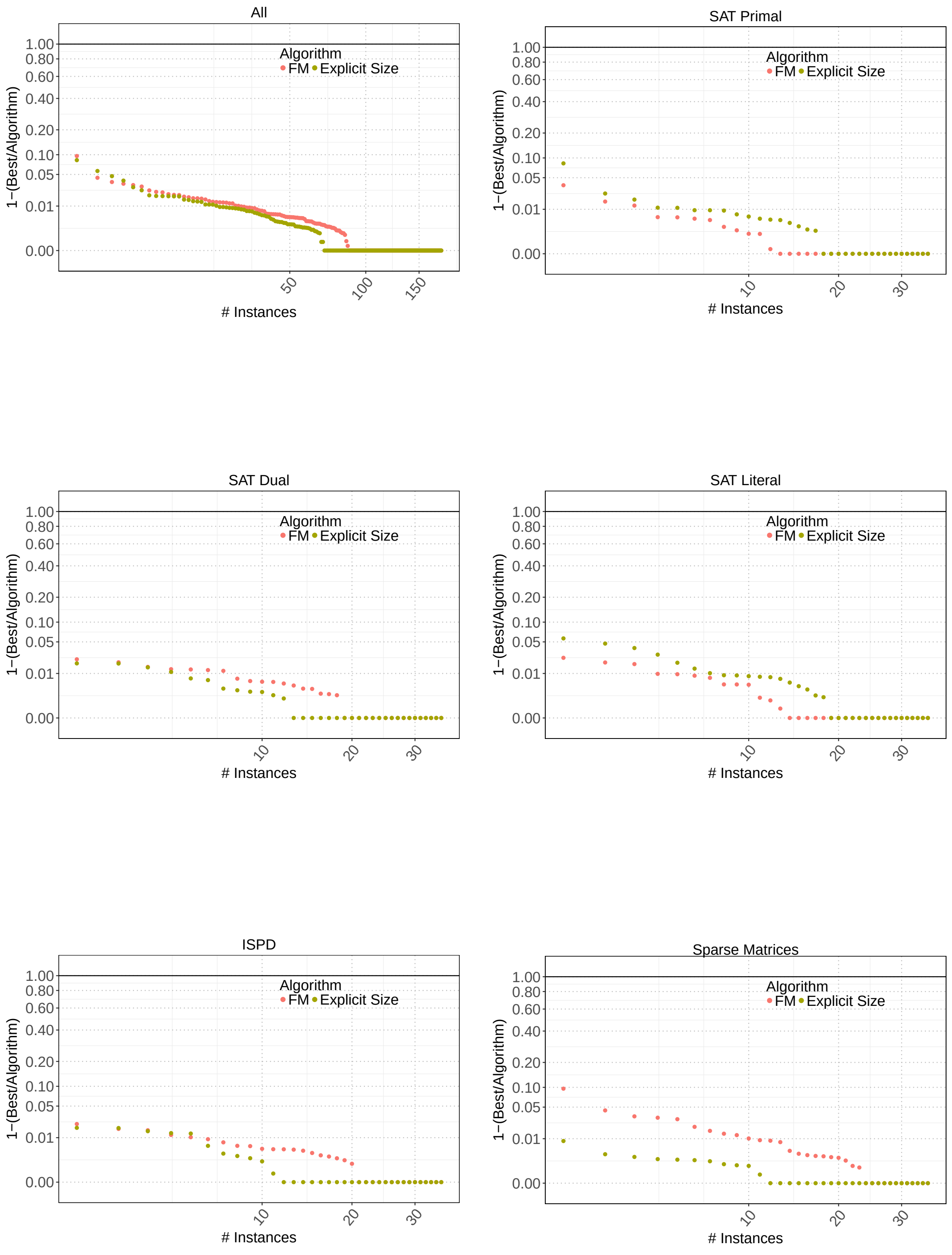
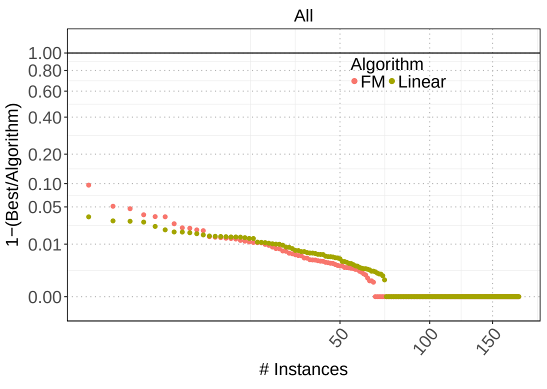
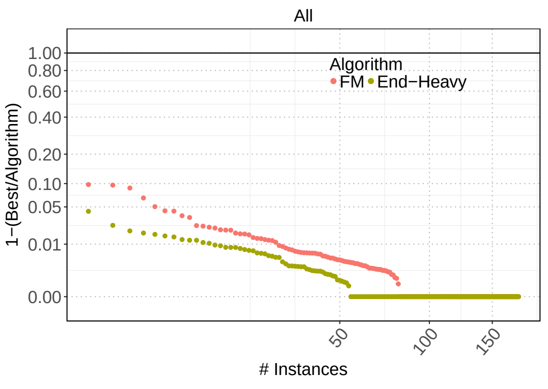


Figure 5.4: Performance plots comparing LHR without max edge size and FM





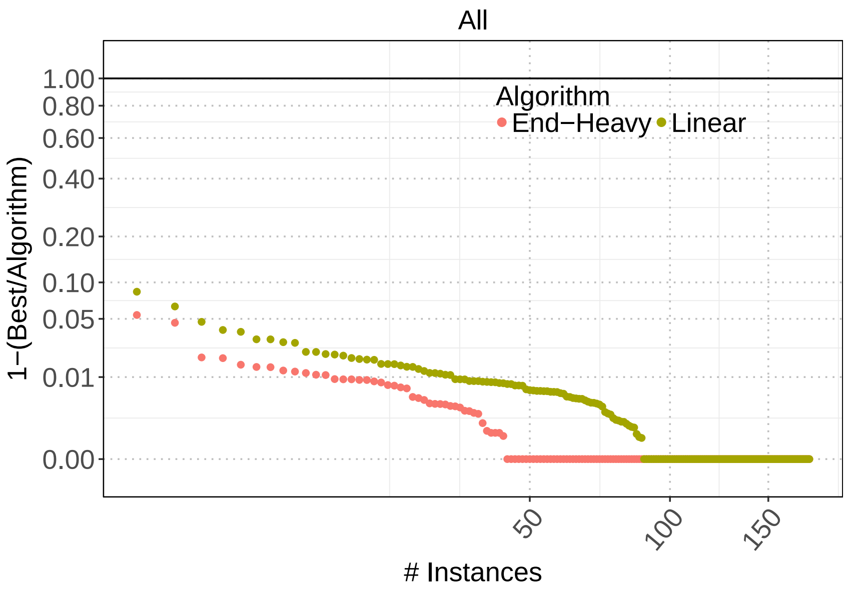
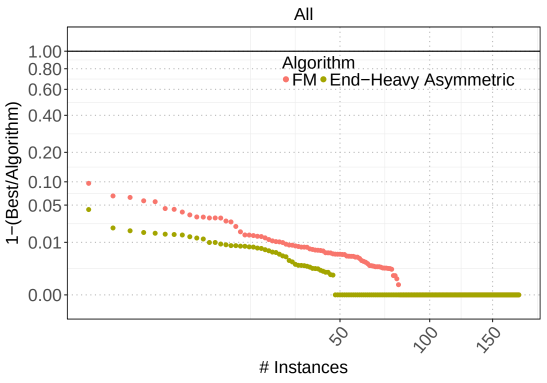


Figure 5.5: Comparison of the linear and end-heavy variants of Interval Soft Gain with the baseline FM (top row) and against each other (bottom).



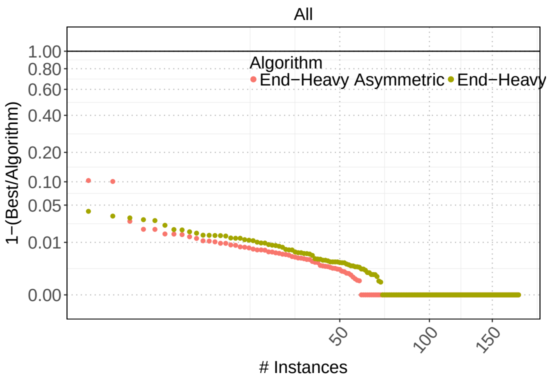


Figure 5.6: Comparison of the asymmetric variant of Interval Soft Gain with the baseline FM (left) and the symmetric one (right).

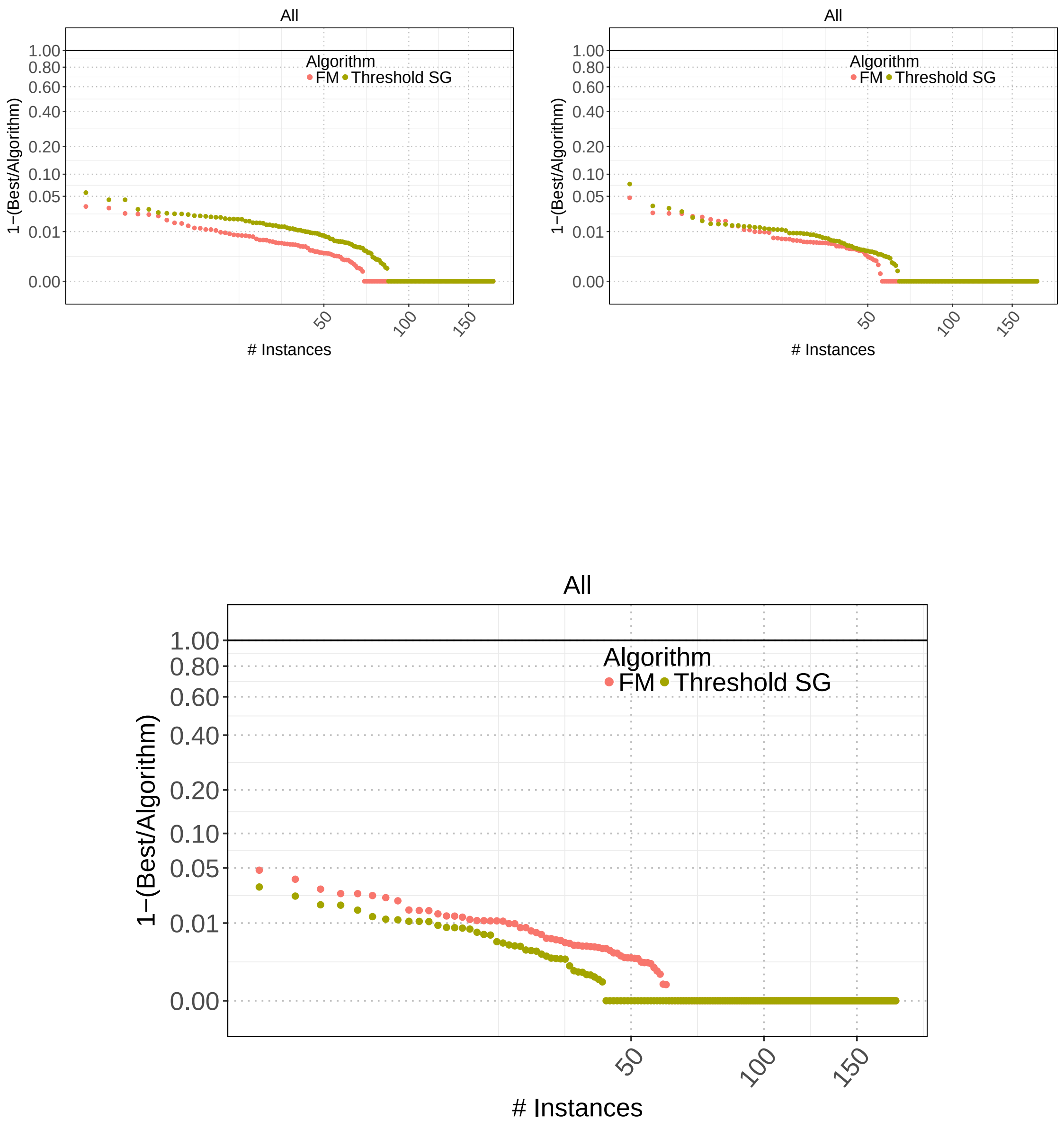


Figure 5.7: Different Threshold Soft Gain parametrisations, compared to the baseline. Top left: thresholds from 0.6 in 0.1 increments, top right: from 0.8 in 0.05 increments, bottom: from 0.9 in 0.025 increments.

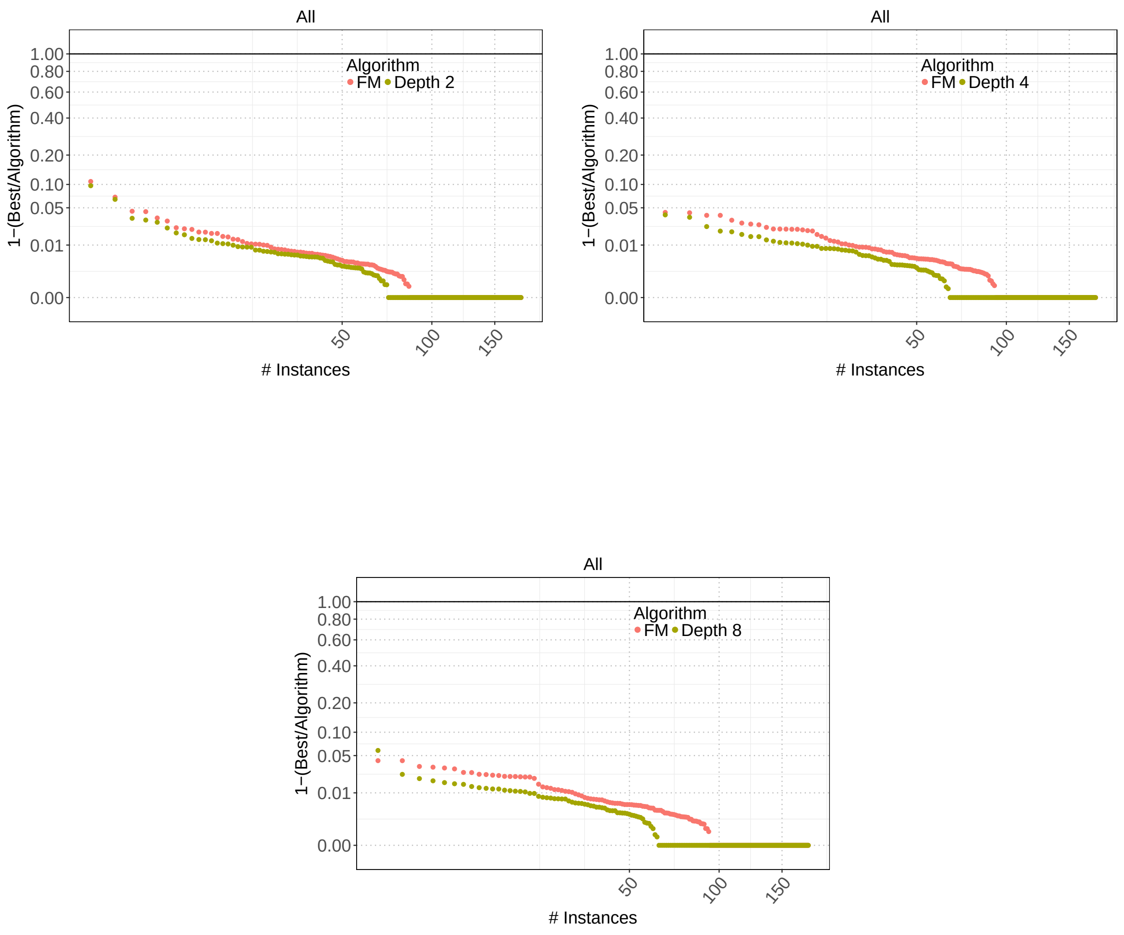
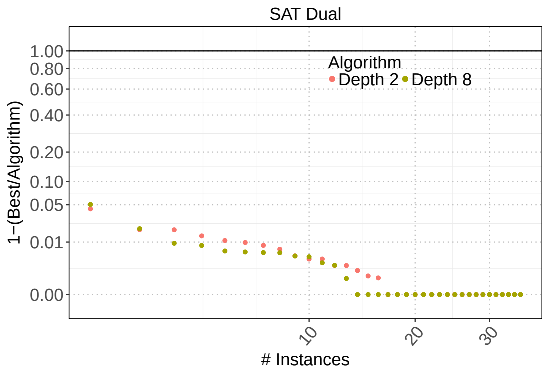
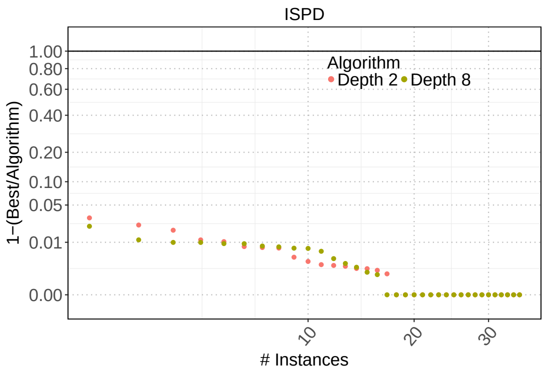


Figure 5.8: Comparison of Lookahead FM with depths 2, 4, and 8 with the baseline FM.





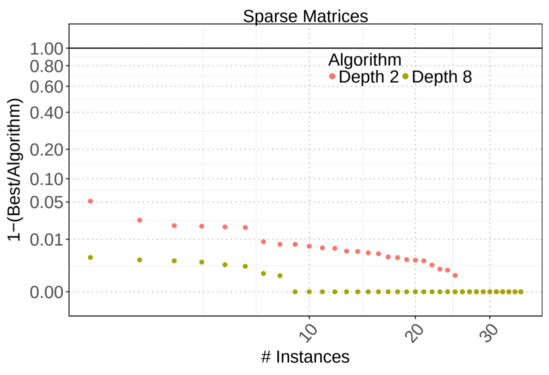
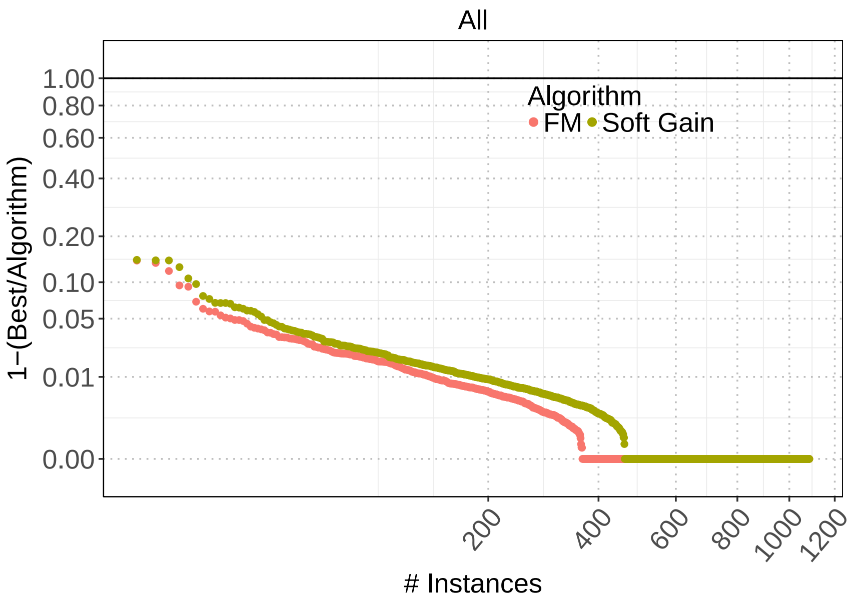
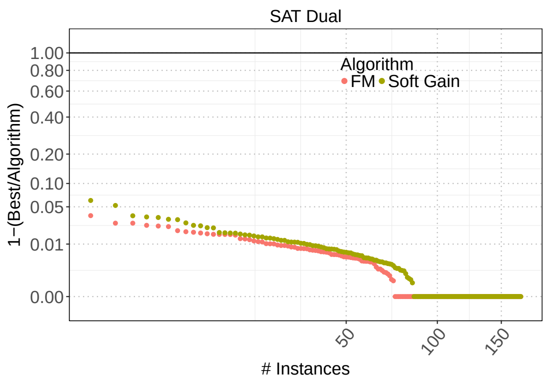


Figure 5.9: Comparison of Lookahead FM depths 2 and 8 over the SAT dual (top left), ISPD (top right) and sparse matrix (bottom) instances.





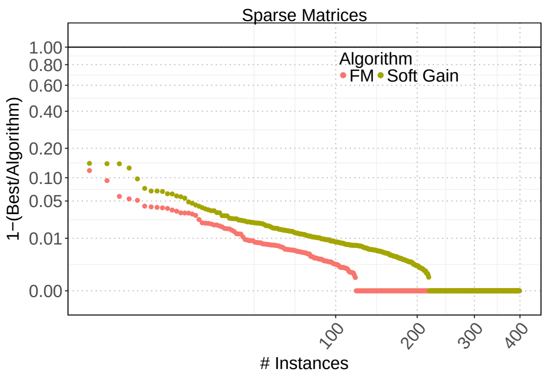


Figure 5.10: Performance plots comparing Soft Gain and FM fo all instances (top), the SAT Primal instances (bottom left) and sparse matrix (bottom right), which are the instance classes with the best and worst performance of Soft Gain, respectively.

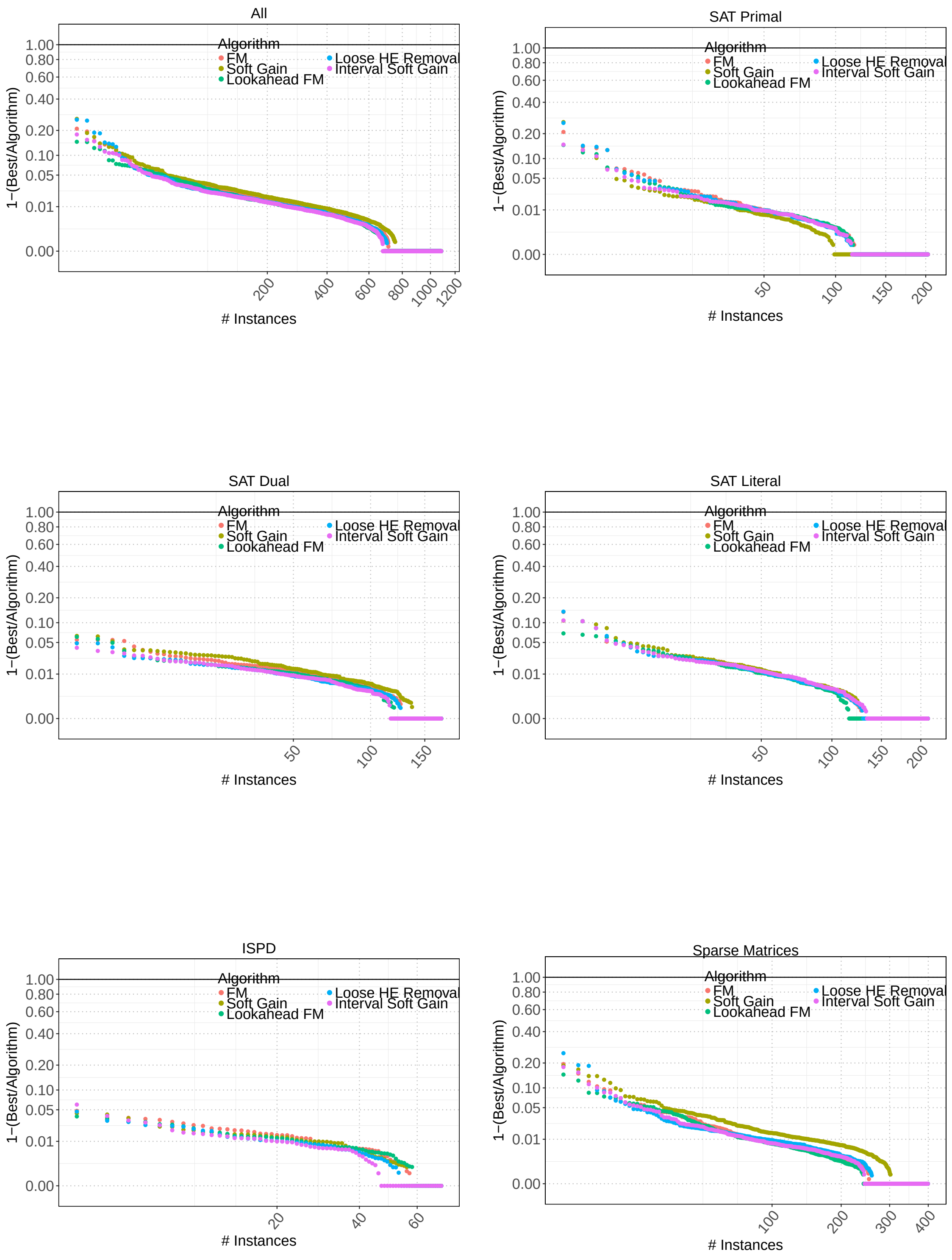


Figure 5.11: Comparison of FM, Soft Gain, Lookahead, Loose Hyperedge Removal and Interval Soft Gain.

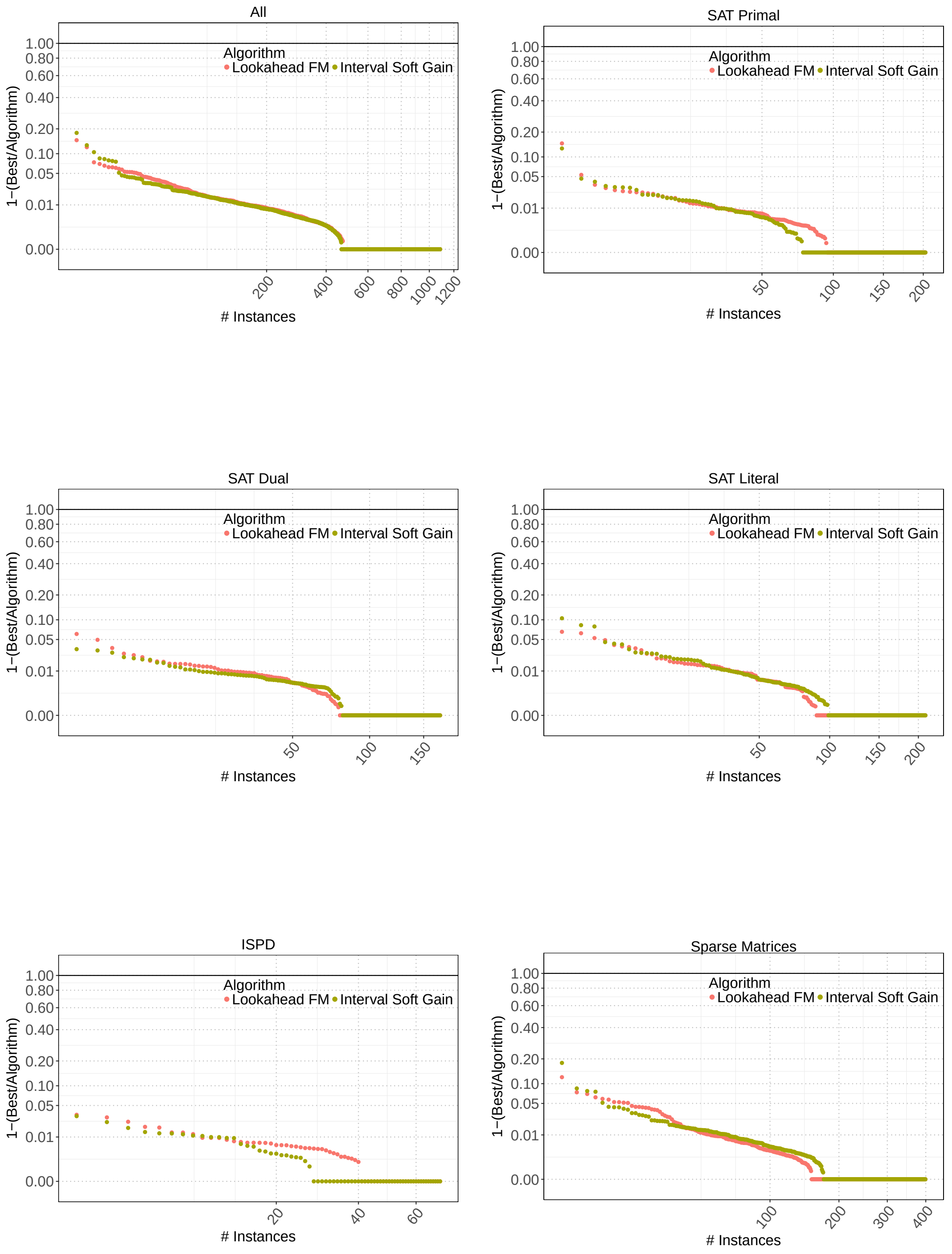


Figure 5.12: Comparison of Lookahead FM and Interval Soft Gain

6. Conclusion

In this chapter we summarise the most important contributions of this work and outline the future improvents that are beyond its scope.

6.1 Summary

We selected, implemented as part of KaHyPar and evaluated the quality of partitions of five strategies for improving FM. We demonstrated that it may be beneficial to augment or even replace the change of cut as a metric to choose the most prospective vertex on each move and use heuristics which take additional factors in consideration. Of the approaches we evaluated, Interval Soft Gain was developed in the course of this work. It proved to be one of the most successful methods, along with Lookahead FM [7]. We improved Loose Hyperedge Removal [10] and considerably improved the quality of its partitions, bringing the results close to the two leading algorithms.

6.2 Future Work

In the course of this work, the algorithms were developed in a flexible way that would allow testing different variants and configurations. Since the local search is one of the most time-consuming phases of the partitioning, it is important to develop optimised versions before integrating them into KaHyPar. In some cases it may be necessary to make compromises between speed and quality. For example, Loose Hyperedge Removal could be sped up by using integer gain additions instead of floating-point, but this will likely degrade the quality of the results by making the gain increases less accurate. Lookahead could benefit from compacting the vectors into integer types, using one byte per entry. This would restrict the gain range to about $[-128, 127]$ (depending on the exact implementation) and necessitate clipping values outside this range to its endpoints. This is unlikely to affect the results, because such gains are extremely rare, if at all present. While this would reduce copying and comparison overhead, it would complicate the gain update operations. Finally, since some parts of the local search code, e.g. the gain update routines, are executed extremely often, low-level tunings there could have a large impact on the runtime, so this is a necessary step in the integration of the new methods into KaHyPar.

Another obvious direction of future development is extending the best of the presented methods for use in the direct k-way partitioning. Especially Loose Hyperedge Removal could provide even greater improvements than in recursive bisection mode, because it would take into account a hyperedge's connectivity in all blocks and favour those where it has the strongest. Currently, Interval Soft Gain is defined over the Θ_e function, which in its current definiton relies on the existence of only two blocks. It must be extended in a meaningful way in order to work well for k-way.

Each of the algorithms we presented was evaluated using fixed parameters that were used for all hypergraphs. Due to its promising results and high flexibility, Interval Soft Gain can be extended to determine its configuration at runtime depending on the properties of the input hypergraph.

Lastly, there are other methods to improve FM which that could also be evaluated, e.g. the Lock Gains from [24] or the gain function from Social Hash Partitioner [25].

Bibliography

[1] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, March 1999.

[2] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, Jul 1999.

[3] Bruce Hendrickson and Tamara G Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519 – 1534, 2000. Graph Partitioning and Parallel Computing.

[4] Jin Huang, Ru Zhang, and Jeffrey Xu ; Yu. Scalable hypergraph learning and processing. In *Proceedings of the IEEE International Conference on Data Mining*, 2015.

[5] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley & Sons, Inc., New York, NY, USA, 1990.

[6] Daniel Delling, Andrew Goldberg, and Renato Werneck. Graph partitioning with natural cuts. Technical report, December 2010.

[7] B. Krishnamurthy. An improved min-cut algonthm for partitioning vlsi networks. *IEEE Trans. Comput.*, 33(5):438–446, May 1984.

[8] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.

[9] Zoltán Ádám Mann and Pál András Papp. Formula partitioning revisited. In Daniel Le Berre, editor, *POS-14. Fifth Pragmatics of SAT workshop*, volume 27 of *EPiC Series in Computing*, pages 41–56. EasyChair, 2014.

[10] Jason Cong, Honching Peter Li, Sung Kyu Lim, Toshiyuki Shibuya, and Dongmin Xu. Large scale circuit partitioning with loose/stable net removal and signal flow based clustering. In *Proceedings of the 1997 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '97, pages 441–446, Washington, DC, USA, 1997. IEEE Computer Society.

[11] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. *k-way Hypergraph Partitioning via n-Level Recursive Bisection*, pages 53–67. 2016.

[12] Ü. V. Çatalyürek. PaToH v3.2. `http://bmi.osu.edu/umit/software.html`. retrieved 27. Sep 2017.

[13] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 343–348, New York, NY, USA, 1999. ACM.

[14] Rob Bisseling. Mondriaan for sparse matrix partitioning. `http://www.staff.science.uu.nl/~bisse101/Mondriaan/`. retrieved 27. Sep 2017.

[15] Andrew E. Caldwell, Andrew B. Kahng, and Igor L. Markov. Improved algorithms for hypergraph bipartitioning. In *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, ASP-DAC '00, pages 661–666, New York, NY, USA, 2000. ACM.

[16] Erik G. Boman, Ümit V. Çatalyürek, Cédric Chevalier, and Karen D. Devine. The zoltan and isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering and coloring. *Sci. Program.*, 20(2):129–150, April 2012.

[17] Aleksandar Trifunovic and William J. Knottenbelt. *Parkway 2.0: A Parallel Multilevel Hypergraph Partitioning Tool*, page 789–800. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[18] Ümit V. Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora Uçar. Umpa: A multi-objective, multi-level partitioner for communication minimization.

[19] Tobias Heuer. Engineering Initial Partitioning Algorithms for direct k-way Hypergraph Partitioning. Bachelor thesis, Karlruhe Institute of Technology, 2015.

[20] Charles J. Alpert. The ispd98 circuit benchmark suite. In *Proceedings of the 1998 International Symposium on Physical Design*, ISPD '98, pages 80–85, New York, NY, USA, 1998. ACM.

[21] Natarajan Viswanathan, Charles J. Alpert, Cliff Sze, Zhuo Li, Gi-Joon Nam, and Jarrod A. Roy. The ispd-2011 routability-driven placement contest and benchmark suite. In *Proceedings of the 2011 International Symposium on Physical Design*, ISPD '11, pages 141–146, New York, NY, USA, 2011. ACM.

[22] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[23] M. Heule A. Belov, D. Diepold and M. Jarvisalo. The sat competition 2014. `http://www.satcompetition.org/2014/`, 2014.

[24] Jong-Pil Kim, Yong-Hyuk Kim, and Byung-Ro Moon. *A Hybrid Genetic Approach for Circuit Bipartitioning*, pages 1054–1064. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[25] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Alessandro Presta, and Yaroslav Akhremtsev. Social hash partitioner: A scalable distributed hypergraph partitioner. *CoRR*, abs/1707.06665, 2017.