Karlsruher Institut für Technologie

Bachelor Thesis

Combining Recursive Bisection and k-way Local Search for Hypergraph Partitioning

Charel Mercatoris

Date: 8. November 2018

Supervisors: Prof. Dr. rer. nat. Peter Sanders
Sebastian Schlag, M.Sc
Dr. rer. nat. Christian Schulz

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Abstract

We combine a multilevel heuristic recursive bisection algorithm for hypergraph partitioning with k-way local search. In particular we describe several algorithms that refine the ε-balanced k-way partition after recursive bisection. Furthermore we present some algorithms, which refine the partition during the recursive bisection process. Moreover we engineered an algorithm which improves the partition with active block scheduling and 2-way local searches. We test the different algorithms for optimising the *cut-net* and *connectivity* metric. Experimental results indicate that partitions, resulting form recursive bisection, can be improved slightly with little extra running time. By investing more running time, it is possible to improve the objective significantly.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 8.11.2018

Contents

1 Introduction

1.1 Motivation

Hypergraphs are a generalisation of graphs. The difference between graphs and hypergraphs is that in a hypergraph an edge or net can link more than two vertices. The basic problem examined in this thesis is the hypergraph partitioning problem. This problem consists of partitioning the vertex set of a hypergraph in k blocks. Graph and hypergraph partitioning is NP-Hard for certain metrics and it is even NP-Hard to find a constant factor approximation [18, 19]. To find a good approximation, multilevel heuristics are used in practice. A multilevel algorithm consists of tree phases: coarsening, initial partitioning and uncoarsening. First the size of the graph or hypergraph is decreased by contracting vertices. After calculating an initial partition, the coarsening is undone and the partition is refined. If we can find a good heuristic algorithm for hypergraph partitioning, this would mean we have a good heuristic algorithm for graph partitioning. Graph partitioning is a special case of hypergraph partitioning where all hyperedges contain two vertices.
Two areas in which graph and hypergraph partitioning are frequently used are scientific computing and VLSI design. Pothen states that graph partitioning can be used in parallel computing [20]. The concurrency in a program can be detected and represented as a graph. Partitioning this graph into subgraphs results in a decomposition of the data and the tasks associated with a computational problem. The subgraphs can be mapped to different processors of a multiprocessor and processed in parallel. For instance sparse direct and iterative solvers use graph partitioning to minimize the communication and ensure load balance [7]. Hypergraph partitioning can be used in the physical design of digital circuits for very large-scale integration (VLSI) systems [7]. To reduce the VLSI design complexity the digital circuit is partitioned into smaller components. The typical optimisation objective is to minimize the weight of connections between the different components. The total length of wires is kept as short as possible. Partitioning is one of the bottle neck in the design process. Further application areas for graph and hypergraph partitioning are image processing, road networks and complex networks [7].
Two major algorithms using the multilevel heuristic are recursive bisection and direct k-way partitioning. Recursive bisection recursively bisects the hypergraph until we have a k-way partition. The direct k-way algorithm partitions the hypergraph during initial partitioning into k blocks. Both algorithms provided good results for the *cut* metric [22, 23]. While optimising the *cut* metric, the algorithm minimizes the number of edges or hyperedges between different blocks. The KaHyPar [22, 23] implementation of both algorithms

use the multilevel paradigm. While recursive bisection uses only 2-way local search the direct k-way algorithm uses k-way local search. Local search algorithms successively move vertices from one block into an other to optimize the metric. In this thesis we combine recursive bisection with k-way local search by refining the partition and/or subpartitions at different states of the recursive bisection.

1.2 Contribution

In this thesis we engineer different algorithms which combine recursive bisection and k-way local search. On the one hand, we propose algorithms that can slightly improve the partition of the recursive bisection with minimal extra running time. On the other hand, we engineer some algorithms which can improve the partition by more than 3% by investing more running time. The algorithms can be divided in two classes: local search refinement after the recursive bisection, which always find better or equal partitions, and local search during the recursive bisection, which may result in worse partitions.
The refinement algorithms especially improve the objective of ISPD, Primal and Literal hypergraphs. The improvements get larger if the number of blocks in a partition increases. The algorithm RB+LC significantly improves the objective for small partitions, but has a longer running time. On the other side, RB+LT has a much lower running time and provides better results for larger partitions. The algorithms RB+L and RB+OL slightly improve the objective and have an insignificantly larger running time.
We tested a direct k-way partitioning algorithm, where we refined the partitions during uncoarsening with 2-way local searches instead of k-way local searches. The 2-way local searches are scheduled between neighbour blocks. This algorithm has a much higher running time and provides worse results.

1.3 Structure of Thesis

After stating the hypergraph partitioning problem and giving general definitions in Chapter 2, we give an overview of related work in Chapter 3. This chapter introduces n-level hypergraph partitioning with coarsening, initial partitioning and uncoarsening in Section 3.1. Furthermore, we will describe the local search algorithm of Kerninghan and Lin, FM local search and k-way FM local search in Section 3.2, V-cycles in Section 3.3 and the recursive bisection algorithm in Section 3.4. Then we will describe algorithms which combine recursive bisection and k-way local search in Chapter 4. The algorithms in Section 4.3 perform a k-way local search on the nodes of the recursive bisection tree. In Section 4.4 we propose an alternative recursive bisection and apply the local search on unfinished partitions. Active block scheduling is used to refine the partitioned hypergraph with 2-way local search in Section 4.5. In Section 4.6 we describe an algorithm that performs 2-way local search with active block scheduling during uncoarsening instead of k-way local search and

Section 4.7 introduces an algorithm that performs V-cycles after recursive bisection. We evaluate the different algorithms in Chapter 5. The experimental environment, the tuning parameters and the test instances are stated in Section 5.1. The results of the experiments are explained in Section 5.3 and 5.3.2. In Section 5.3.3 we compare our algorithms to the state-of-the-art hypergraph partitioner. Chapter 6 contains a conclusion and the future work.

2 Fundamentals

2.1 General Definitions

Hypergraphs are a generalisation of graphs. An undirected hypergraph is a quadruple $H = (V, E, c, w)$, where V is a set of vertices and E a set of hyperedges (or nets) with $E \subseteq 2^V$. The vertices of a net $e \in E$ are also called pins. $|e|$ is the number of pins in a net e, it is called the size of e. If $|e| = 1$ a net e is called-single node net. The vertex weight function $c : V \to \mathbb{R}_{\geq 0}$ assigns each vertex a positive weight. The hyperedge weight function $w : E \to \mathbb{R}_{\geq 0}$ assigns each edge a positive weight. We define $n = |V|$ and $m = |E|$ where $|\cdot|$ is the cardinality of a set. The weight functions c and w are extended to sets. Let V_i be a set of vertices ($V_i \subseteq V$) and E_i be a set of nets ($E_i \subseteq E$), then we define $c(V_i) = \sum_{v \in V_i} c(v)$ and $w(E_i) = \sum_{e \in E_i} w(e)$.
Let e be a net and v be a pin, then v is incident to e if $v \in e$. The set of all nets incident to a vertex $v \in V$ is $I(v) = \{e \in E \mid v \in e\}$. The number of incident nets $d(v) = |I(v)|$ is called the degree of $v \in V$. Two vertices $v_1, v_2 \in V$ are called adjacent if $\exists e \in E$ with $\{v_1, v_2\} \subseteq e$. The set of all neighbours of $v \in V$ is $\Gamma(v) = \{u \mid \exists e \in E, u \in e \wedge v \in e\}$.
A hypergraph $H' = (V', E', c', w')$ is called a subhypergraph of $H = (V, E, c, w)$ if the vertices of H' are contained in H and each net of H' is in E'. The weight functions of H' are the weight functions restricted on V' and E'. In other words $H' = (V', E', c', w')$ is subhypergraph of $H = (V, E, c, w)$ if:

(i) $V' \subseteq V$

(ii) $E' \subseteq E \cap V' \times V'$

(iii) $c' : V' \to \mathbb{R}_{\geq 0}, c'(v) = c(v)$

(iv) $w' : E' \to \mathbb{R}_{\geq 0}, w'(v) = w(v)$

For a hypergraph $H = (V, E, c, w)$ a k-way partition is a partition of the vertex set V in k blocks $\Pi = \{V_1, ..., V_k\}$ with:

(i) $\bigcup_{i=1}^{k} V_i = V$

(ii) $\forall i \in \{1, ..., k\} : V_i \neq \emptyset$

(iii) $\forall i, j \in \{1, ..., k\} \wedge i \neq j : V_i \cap V_j = \emptyset$.

The block id $b[v]$ of a vertex $v \in V$ is used to refer the block which contains v. So v is contained in $V_{b[v]}$. Let $L_{max} = \lceil (1 + \varepsilon) \frac{c(V)}{k} \rceil$ with $\varepsilon > 0$ be the upper bound for the block weights $c(V_i)$. Then the k-way partition Π of a hypergraph is called ε-balanced if it satisfies

the balance constraint: $\forall i \in \{1, ..., k\} : c(V_i) \leq L_{max}$. A block V_i is called underloaded if $c(V_i) < L_{max}$, overloaded if $c(V_i) > L_{max}$ and perfectly balanced if $\lceil c(V_i) = \frac{c(V)}{k} \rceil$.
Let $\Pi = \{V_1, ..., V_k\}$ be a k-way partition of a hypergraph $H = (V, E, c, w)$. $\Phi(e, V_i) = |\{v \in V_i \mid v \in e\}|$ is the number of pins of a net $e \in E$ which are in block V_i. e is connected to V_i if $\Phi(e, V_i) > 0$. A block V_i is adjacent to a vertex $v \notin V_i$ if v is incident to a net $e \in E$ which is connected to V_i ($\exists e \in I(v) : \Phi(e, V_i) > 0$). We use $R(v) = \{U \in \Pi \mid \exists e \in I(v) : \Phi(e, U) > 0\}$ to denote the set of all blocks adjacent to v. The connectivity set of a net $e \in E$ is the set of blocks to which e is connected: $\Lambda(e) = \{U \in \Pi \mid \Phi(e, U) > 0\}$. The connectivity of a net $e \in E$ is defined as the number of blocks $\lambda(e) = |\Lambda(e)|$ to which e is connected. If a net $e \in E$ is only inside one block ($\lambda(e) = 1$) the net is called internal. If e is connected to at least two blocks ($\lambda(e) > 1$) the net is called cut net. We call a vertex $v \in V$ border vertex (or border node) if v is incident to a cut net $e \in E$.
Let $H = (V, E, c, w)$ be a hypergraph with partition $\Pi = \{V_1, ..., V_k\}$. Two blocks $V_i, V_j \in \Pi, i \neq j$ are adjacent if there exits a net $e \in E$ where e is connected to V_i and V_j. The quotient graph of H is a undirected graph $Q = (Qv, Qe)$ with $Qv = \{V_1, ...V_k\} = \Pi$ and $Qe = \{\{a, b\} \subseteq Qv \mid \exists e \in E : \{a, b\} \subseteq \Lambda(e)\}$. In other words the nodes of the quotient graph are the blocks of the partition Π. There is an edge between two blocks if they are adjacent.
The k-way hypergraph partitioning problem is to find an ε-balanced k-way partition Π for a hypergraph $H = (V, E, c, w)$ which minimizes a certain metric. The k-way partitioning problem is NP-Hard for certain metrics, it is even NP-Hard to find good approximate solutions for graphs [18, 19]. Two of this metrics are the cut net metric (*cut*) and the connectivity metric. The cut net metric minimizes $cut(\Pi) = w(E')$ where $E' = \{e \in E \mid \lambda(e) > 1\}$ is the set of all cut nets. The connectivity metric or λ^{-1} metric considers the fact that a net is connected to more than two partitions [22]. The connectivity metric is $\lambda^{-1}(\Pi) = \sum_{e \in E'} (\lambda(e) - 1)$.
Let $H = (V, E, c, w)$ be a hypergraph. A recursive bisection tree is a tree $T = (N, R)$ where N are the nodes and R the edges. The nodes N are the subhypergraphs which occur during the recursive bisection of H. The root of T is the input hypergraph H of the recursive bisection. The leafs are the subhypergraphs containing only the vertices of a block of the final partition. The edges $R = \{(h, s) \mid h, s \in N \wedge s$ *results from the bisection of* $h\}$ represent the bisections of the hypergraphs. The depth of a node $n \in N$ in a tree $T = (N, R)$ is the length of the path from the root w of T to n. By convention the depth of the root is zero. Figure 2.1 illustrates a recursive bisection tree for calculating a 4-way partition. H is the original hypergraph of which a 4-way partition should be calculated. The depth of H is zero. H is bisected into H_1 and H_2, which have depth one. There is an edge from H to H_1 and H_2. Finally H_1 and H_2 are bisected recursively. So $T = (N, R)$ with $V = \{H, H_1, H_2\}$ and $R = \{(H, H_1), (H, H_2)\}$ is the recursive bisection tree of H.

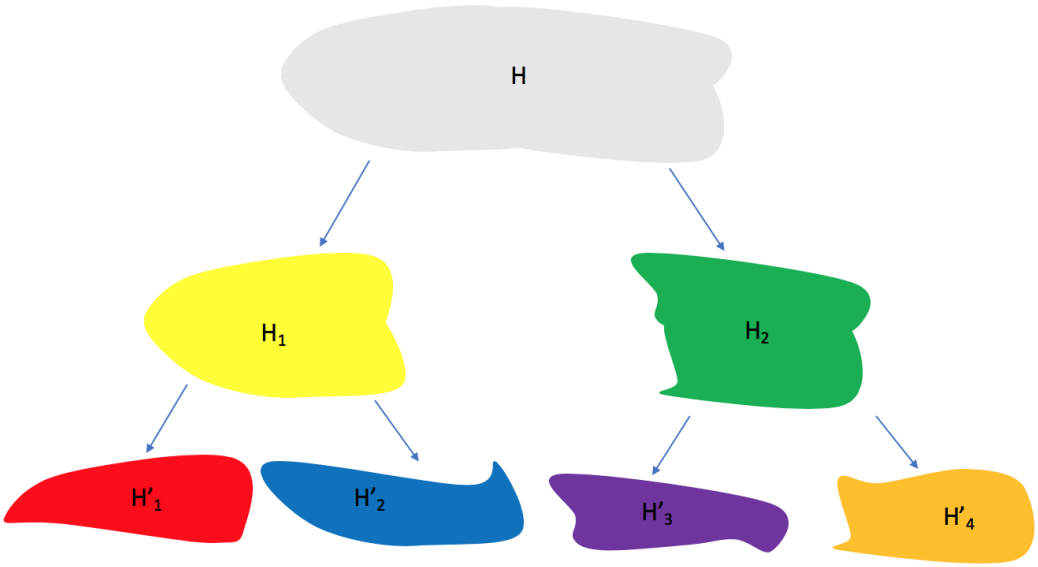


Figure 2.1: Illustration of the recursive bisection tree for a 4-way partitioning. The arrows represent a bisection. The nodes of the tree are the subhypergraphs resulting from bisection. The root H is the original hypergraph.

3 Related Work

3.1 n-Level Hypergraph Partitioning

To find a good ε-balanced k-way partition of a hypergraph, while optimizing a certain metric, heuristic multilevel algorithms are used. A multilevel algorithm consists of three phases [11, 22]: coarsening, initial partitioning and uncoarsening with local search refinement. The main idea of multilevel algorithms is to get a smaller hypergraph which has similar properties as the original hypergraph. This diminution of the hypergraph is called coarsening. After an initial partition has been calculated on the smallest hypergraph, it is uncoarsened and refined with a local search algorithm described in Section 3.2.

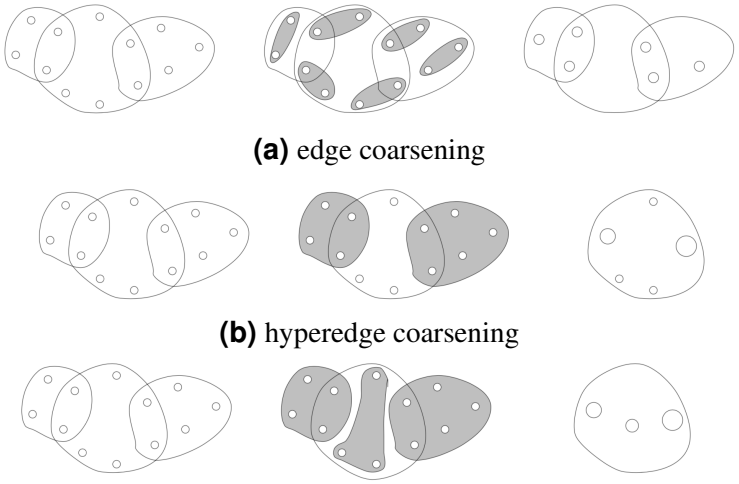
3.1.1 Coarsening

Coarsening is used to contract highly connected vertices with the objective of decreasing the number of nets as well as their size [14, 22]. This results in a smaller hypergraph with similar properties than the original hypergraph. Decreasing the number of nets leads to simpler instances for the initial partitioning. Let $H = (V, E, c, w)$ be a hypergraph. Contacting $(u, v) \in V \times V$ with $u \neq v$ means merging v into u, where u is the representative and v the contraction partner [23]. The weight of u is recalculated: $c(u) = c(u) + c(v)$. For each net $e \in I(v) \setminus I(u)$, v is replaced by u. For each net $e \in I(v) \cap I(u)$, v is removed. In other words, in each net $e \in E$ which contains v, v is replaced by u if e does not contain u, otherwise v is removed.
During coarsening a sequence of smaller hypergraphs is created [14]. Three different methods are [14]:

(i) *Edge Coarsening*: First a heavy-edge maximal matching of the vertices of the hypergraph is calculated, then they are contracted pairwise. This procedure is illustrated in 3.1a.

(ii) *Hyperedge Coarsening*: Let $H = (V, E, c, w)$ be a hypergraph. During *hyperedge coarsening* an independent set of nets $W \subseteq E$ $(\forall a, b \in W, a \neq b \implies a \cap b = \emptyset)$ is selected. Then for all nets $e \in W$ all vertices are contracted into one. Small nets with a large weight are preferred to be contracted. This procedure is illustrated in 3.1b.

(iii) *Modified Hyperedge Coarsening*: *Modified hyperedge coarsening* is illustrated in 3.1c. First a *hyperedge coarsening* is performed. Then the vertices in an uncontraced net, which are not part in a contracted net, are contracted.



(a) edge coarsening

(b) hyperedge coarsening

(c) modified hyperedge coarsening

Figure 3.1: Different methods to match the vertices during coarsening [14]

Algorithm 1: Coarsening of a Hypergraph

Input: $H = (V, E, c, w)$
Result: $H = (V, E, c, w)$
1 while H *is not small enough* do
2 $\quad (u, v) := argmax_{u \in V} score(u)$
3 $\quad H := \text{CONTRACT}(H, u, v)$
4 end

The Karlsruhe Hypergraph Partitioner (KaHyPar) [11, 22, 23] is a n-level hypergraph partitioner which uses a variation of *edge coarsening*. Let $H = (V, E, c, w)$ be a hypergraph. Between each level of the coarsening hierarchy, only two vertices are contracted. These vertices are determined by a rating function: $r(u, v) = \sum_{e \in I(v) \cap I(u)} \frac{w(e)}{|e|-1}$ with $u, v \in V, u \neq v$. It prefers vertex pairs (u, v) where $I(v) \cap I(u)$ contains a lot of heavy nets with a small size. This rating function is called the *heavy-edge* rating function. It is also used by hMetis [14] and Parkway [24]. To keep the vertex weight reasonably uniform KahyPar [23] uses a variant of the *heavy-edge* rating function: $r(u, v) = \frac{1}{c(v) \cdot c(u)} \sum_{e \in I(v) \cap I(u)} \frac{w(e)}{|e|-1}$. To find the next contraction partner we calculate for each $u \in V$ the rating $r(u, v)$ for each neighbour v of u ($v \in \Gamma(u)$). The next contraction partner of u will be the neighbour v with the highest rating $r(u, v)$.
Algorithm 1 performs a n-level coarsening on a hypergraph $H = (V, E, c, w)$. The coarsening steps are repeated until the hypergraph is small enough in Line 1. In Line 2 the next vertices $u, v \in V$, which shall be contracted, are determined according to the rating function r. Then (u, v) are contracted. After contraction, single node nets and parallel nets are

removed. A detailed explanation on the data structures and algorithms used to perform the n-level coarsening can be found in [11].
KaHyPar implements two different methods to coarsen a hypergraph [22, 23]. Let $H = (V, E, c, w)$ be a hypergraph which shall be partitioned into k blocks. The *heavy lazy* coarsening algorithm is used in the recursive bisection implementation of KaHyPar [23]. First all vertices are rated, which means for each vertex $u \in V$ the ratings with each neighbour $v \in \Gamma(v)$ is calculated. The neighbour v of u with the highest rating is chosen as contraction partner. To increase diversification ties are broken randomly. Each vertex with its contraction partner is inserted in an addressable priority queue with their rating as key. This data structure allows a fast access to the best rated contraction pair. In each iteration the *heavy lazy* algorithm removes the vertex pair (u, v) with the highest gain and contracts them. Since v is no longer contained, it is removed from the priority queue. The contraction can lead to parallel nets and single node nets. Single node nets are removed and only one of the parallel nets remains in the hypergraph. The ratings of the neighbours $\Gamma(u)$ of u are updated in the priority queue. Another coarsening algorithm is used by KaHyPar during direct k-way partitioning [22]. The algorithm works in passes. In each pass first a random permutation of the vertex set V is created. Then for each vertex $u \in V$ the contraction partner $v \in V$ is determined according to the *heavy-edge* rating function. To avoid imbalanced inputs for the initial partition phase, ties are broken in favour of unmatched vertices. A vertex $u \in V$ with $c(v) > \lceil \frac{c(V)}{k} \rceil$ is not contracted. As soon as a contraction pair (u, v) is found they are contacted and v is removed from then vertex set V. The vertex v is no longer considered as contraction partner in this or future passes. After removing v the next contraction pair is searched. A pass ends if each vertex has been considered as contraction partner or representative. Then a new pass starts. If the hypergraph is small enough, the coarsening is stopped.

3.1.2 Initial Partitioning

Coarsening is done until the number of vertices is small enough. For instance, if the number of vertices is $160k$ or no valid contraction can be found, then initial partitioning is executed. The goal of initial partitioning is to compute an ε-balanced k-way partition of the coarsened hypergraph. Furthermore, the metric should be optimized. A well balanced initial partition with a well optimized metric leads to good partition after uncoarsening [16]. This is due to the fact that the weights of the nets and vertices are updated during coarsening. KaHyPar [11, 22, 23] computes multiple initial partitions with different algorithms and random seeds. All of these algorithms use randomization and so different seeds can lead to different results. The partition with the best metric and least imbalance is chosen to continue. If there is no balanced partition, the partition with the smallest imbalance is chosen. Some initial partitioning algorithms are [16, 22, 23]:

(i) *Coarsening until k vertices are left:* The coarsening algorithm is performed until the hypergraph consists of only k vertices. Then the k blocks will be the k vertices. The

problem with this approach is that after performing the coarsening the vertices are likely to have different sizes which leads to highly imbalanced partitions.

(ii) *Recursive Bisection:* Compute the initial partition with recursive bisection, which is explained in Section 3.4.

(iii) *Random Partitioning:* During the random initial partitioning algorithm the vertices are assigned randomly to the k partitions. The disadvantage of this algorithm is that it does not optimize the metric. The advantage is that it is fast and runs in $O(|V|)$, so it can be repeated multiple times.

(iv) *Breadth-First-Search:* This initial partitioning algorithm uses Breadth-First-Search (BFS) to get a partition. First, k vertices are needed to initialize the algorithm. This algorithm chooses pseudo-peripheral vertices. To calculate them first one vertex is chosen, then a BFS is applied. The last vertex visited by the BFS is added to the pseudo-peripheral vertices. Then a BFS is applied initialized with the two vertices. This procedure is repeated until we have k vertices. Each block of the partition is initialized with one of the last vertices. Then BFS are scheduled in a round-robin fashion on the blocks and the vertices are added to the blocks in the order in which they are visited by the BFSs. The algorithm stops if all vertices have been assigned to a block.

(v) *Greedy Hypergraph Growing:* The greedy graph growing algorithm calculates a partition in a similar way as the BFS search algorithm in the previous paragraph. First k pseudo-peripheral vertices are calculated, then the blocks of the partition are initialized with these vertices. Then BFSs are started on the different blocks. The blocks for the vertices are now chosen by a gain function. Typical gain functions are the FM, Max-Net and Max-Pin gain functions. The FM-gain function is explained in Section 3.2. Let $H = (V, E, c, w)$ a hypergraph with partition $\Pi = \{V_1, ..., V_k\}$ and $v \in V$. The Max-Pin gain function is $g(v, V_i, V_j) = |\{u \mid u \in I(v) \wedge u \in V_j\}|$ and prefers moves of vertices based on the number of incident vertices in the growing block. The Max-Net gain function is $g(v, V_i, V_j) = |\{e \mid e \in \Gamma(v) \wedge V_j \in \Lambda(e)\}|$ and prefers moves of nets based on the number of incident nets that connects the growing block with the vertex. The vertex with the greatest gain is added to the block.

Further explanations on initial partitioning can be found in [22].

3.1.3 Uncoarsening

After initially partitioning, the hypergraph is successively uncoarsened. The vertices are uncontracted in reverse order of coarsening. Then a k-way local search algorithm is used to refine the hypergraph.
The input parameters of the uncoarsening Algorithm 2 are the coarsened hypergraph H, the initial partition Π and the imbalance ε. The algorithm returns the final partition Π and the original hypergraph H. While the hypergraph is not completely uncoarsened the vertices

Algorithm 2: Uncoarsening of a Hypergraph

```
Input: H = (V, E, c, w), Π = {V_1, ..., V_k}, ε
Result: H = (V, E, c, w), Π = {V_1, ..., V_k}
1 while H is not completely uncoarsened do
2 |   (H, Π, u, v) := UNCONTRACT(H, Π)
3 |   (H, Π) := LOCALSEARCH(H, Π, {u, v}, ε)
4 end
```

are uncontracted in reverse order of contraction in Line 2. Then local search is performed. The uncontracted vertices $\{u, v\}$ are used as input parameters in Line 3. The local search algorithm is explained in Section 3.2.

3.2 Local Search

The goal of a local search algorithm is to optimize the metric of an ε-balanced k-way partition partition $\Pi = \{V_1, ..., V_k\}$ of a hypergraph $H = (V, E, c, w)$. The basic idea is to move nodes from one block V_i to another block V_j in order to improve the partition.

3.2.1 Local Search of Kerninghan and Lin

One of the first local search algorithms was proposed in 1970 by Kerninghan and Lin [17]. They engineered a 2-way local search algorithm. Let $G = (V, E, w)$ be a graph, where V is the vertex set, $E \subseteq \{\{a, b\} \mid a, b \in V \wedge a \neq b\}$ is the edge set and $w : E \to \mathbb{R}_+ \setminus \{0\}$ is the edge weight function. The algorithm works with unit vertex weights. Let $\Pi = \{V_1, V_2\}$ be a perfectly balanced 2-way partition of V. A perfectly balanced partition is an ε-balanced partition where $\varepsilon = 0$. Let $E' = \{(x, y) \mid x \in V_1 \wedge y \in V_2\}$ be the set of cut edges. The partition is now refined by successively exchanging pairs of vertices (v_1, v_2) with $v_1 \in V_1$ and $v_2 \in V_2$ to optimize the cut $cut(\Pi) = \sum_{e \in E \cap E'} w(e)$. The gain of exchanging two vertices $(v_1, v_2) \in V_1 \times V_2$ is:

$$g(v_1, v_2) = \begin{cases} D(a) + D(b) - 2w(\{v_1, v_2\}), & \{v_1, v_2\} \in E \\ D(a) + D(b), & \{v_1, v_2\} \notin E \end{cases}$$

For $v \in V_i$ and $i \neq j$ $E(v) = \sum_{e=\{v,y\} \in E \cap E'} w(e)$ is the external cost of x. So the external cost of a vertex $v \in V_i$ is the sum of edge weights of all edges which contain v and go from V_1 to V_2. For $v \in V_i$ and $i \neq j$ $I(v) = \sum_{e=\{v,y\} \in E \cap \{\{x,y\} | x,y \in V_i\}} w(e)$ is the internet cost of v. The internal cost of a vertex $v \in V_i$ is the sum of all edge weights of all edges which contain v and are inside V_i. $D(v) = E(v) - I(v)$ is the gain of moving the vertex $v \in V$

Algorithm 3: Kerninghan and Lin Local Search

```
Input: G = (V, E, w), Π = {V_1, V_2}
Result: Π' = {V_1, V_2}
1  ∀v ∈ V : D(v) := E(v) − I(v)
2  ∀v ∈ V : active(v) := true
3  W := {Π}
4  while ∃v_1 ∈ V_1 : active(v_1) ∧ ∃v_2 ∈ V_2 : active(v_2) do
5  |  choose (v_1, v_2) ∈ V_1 × V_2 where g(v_1, v_2) is maximal
6  |  V_1 := (V_1 \ {v_1}) ∪ {v_2}
7  |  V_2 := (V_2 \ {v_2}) ∪ {v_1}
8  |  active(v_1) := false
9  |  active(v_2) := false
10 |  ∀x ∈ V_i \ {v_i} : D(x) := D(x) + 2w({x, v_i}) − 2w({x, v_j}), j ≠ i, w(e) = 0 if e ∉ E
11 |  W := W ∪ {{V_1, V_2}}
12 end
13 Π' := {V_1, V_2} ∈ W where cut({V_1, V_2}) is minimal
```

from block V_i to V_j, with $i \neq j$.
Algorithm 3 of Kerninghan and Lin calculates D for each vertex $v \in V$. All vertices are set active. Then select $(v_1, v_2) \in V_1 \times V_2$ where $g(v_1, v_2)$ is maximal. v_1 and v_2 correspond to the largest possible gain from a single exchange. $g(v_1, v_2)$ can be negative. Exchange them and update D in Line 10. The swapped nodes are set $inactive$ or $locked$ and cannot be swapped again. Continue until no swap operation is left. In W, we keep track of the different partitions which are calculated by the algorithm. Finally, choose the partition where the cut is minimal. If no improvement can be found, the algorithm has found a local optimum. Note that after each vertex has been swapped V_1 and V_2 have exchanged all their vertices. The running time of this refinement algorithm is $O(|V| \cdot |E| log(|E|))$. Further explications and proofs can be found in [17].

3.2.2 FM Local Search of Fiduccia and Matheyses

Fiduccia and Matheyses [10] base their refinement algorithm on Kerninghan and Lin's local search algorithm. Let $H = (V, E, c, w)$ be a hypergraph and $\Pi = \{V_1, V_2\}$ be an ε-balanced 2-way partition. The basic idea of the FM local search is to move vertices $v \in V$ one by one from one block to the other in order to minimize the cut $cut(\Pi)$. The next vertex which should be moved is chosen by the gain $g : V \to \mathbb{Z}$. The gain g indicates how much the cut is improved when moving the vertex. Note that $g(v)$ can be negative. The next vertex to be moved is chosen randomly if there are more vertices with maximum gain. Furthermore, a vertex can only be moved if the balance constraint is not violated.

Algorithm 4: FM Local Search

Input: $H = (V, E, c, w), \Pi = \{V_1, V_2\}, \varepsilon$
Result: $\Pi' = \{V_1, V_2\}$
1 $\forall v \in V : active(v) := true$
2 $W := \{\Pi\}$
3 `while` *moves can be done* `do`
4 choose $v \in V_i$ where $g(v)$ is maximal,$c(v) + c(V_i) \leq \lceil (1+\varepsilon)\frac{c(V)}{k} \rceil$ and $active(v)$
5 $V_i := V_i \setminus \{v\}$
6 $V_j := V_j \cup \{v\}$ where $j \neq i$
7 $active(v) := false$
8 $W := W \cup \{\{V_1, V_2\}\}$
9 `end`
10 $\Pi' := \{V_1, V_2\} \in W$ where $cut(\{V_1, V_2\})$ is minimal

Moves are even done if the cut gets worse. At the end the partition with the best cut is chosen like in the algorithm of Kerninghan and Lin. Each vertex can be moved only once. Algorithm 4 contains the basic idea of the FM local search. The input parameters are a hypergraph $H = (V, E, c, w)$, an ε-balanced 2-way partition and the imbalance ε. The algorithm returns a partition with the same or smaller cut. First all vertices are set active. Then they are moved like described above. The algorithm terminates if no moves can be done in Line 3. That means no vertices are active or no move can be done which does not violate the balance constraint. In W, we keep track of the different partitions which are calculated by the algorithm. Finally, the partition with the best cut is returned. Fiduccia and Matheyses use bucket arrays to efficiently handle the calculation of $g(v)$ [10].

3.2.3 Localized adaptive k-way FM Local Search

KaHyPar [22] implements a version of FM local search which can be used to refine a k-way partition efficiently. The local search algorithm moves the vertices with the best improvement in the objective even if they are negative to another block. Let $H = (V, E, c, w)$ be a hypergraph and $\Pi = \{V_1, ..., V_k\}$ be an ε-balanced k-way partition. The algorithm keeps track of the best solution and returns it. KaHyPar uses k priority queues to get the blocks with the best gain fast. Moreover, the algorithm moves a vertex $v \in V$ only to adjacent blocks $R(v) \setminus \{b[v]\}$. So the gain has to be calculated only for adjacent blocks and not for all k blocks. The local search algorithm is highly localized and is initialized with a subset of the border vertices. First only these vertices can be moved, after moving a vertex $v \in V$ it is set inactive and cannot be moved furthermore. The neighbours $N(v)$ are activated and can be moved too. The gain $g_i : V \to \mathbb{Z}$ defines the improvement of the objective by moving vertex $v \in V$ to block V_i.

$$g_i(v) = \sum_{e \in I(v) \cap \{e \in R | \Phi(e, b[v]) = 1\}} w(e) - \sum_{e \in I(v) \cap \{e \in R | \Phi(e, V_i) = 0\}} w(e)$$

Algorithm 5 performs a k-way local search on a hypergraph $H = (V, E, c, w)$ and an ε-balanced k-way partition $\Pi = \{V_1, ..., V_k\}$. It returns a refined partition Π'. The algorithm maintains k priority queues P_i which contain tuples $(v, g_i(v))$ where v is a vertex and $g_i(v)$ is the gain of moving v to block V_i. The key of the priority queues is the gain g_i. First all vertices are labelled $inactive$ and $unmarked$, the priority queues are enabled. The local search is initialized with a subset B of the border vertices. Each vertex $v \in B$ is activated and added to the priority queues in Line 4 to 11. The non-empty priority queues which correspond to underloaded blocks are enabled. Moves are performed while there are non-empty enabled priority queues or a stop condition is triggered. To find out more about these stop conditions we refer to [22]. Choose a feasible move with the biggest gain g_i and move the vertex to block V_i. A move is feasible if it does not violate the balance constraint. The moved vertex is labelled inactive and marked. It cannot be moved again. The priority queues are updated. The moved vertex v is deleted from the queues in Line 23 and the gain values of the active vertices are updated in Line 35. The gain update procedure is explained in [22]. The unmarked and disabled neighbours $\Gamma(v)$ of v are activated and added to the priority queues. Neighbour vertices which become internal are removed from the queues in Line 32. Since it is possible that gain g_i is negative we keep track of the best objective. After the algorithm has finished, the partition with the best metric is returned. Then all vertices become unmarked and inactive and Algorithm 5 is repeated until no further improvement can be found. For further information on the data structures and algorithms used to update the gains we refer to [22].

3.3 V-Cycle

The idea of V-cycles or iterated multilevel partitioning is explained by Walshaw [25]. The basic idea is to refine an ε-balanced k-way partition $\Pi = \{V_1, ..., V_k\}$ of a hypergraph $H = (V, E, c, w)$ by performing a partition$n$-sensitive coarsening and apply local search during uncoarsening.
Algorithm 6 performs a V-cycle. The input parameters are a hypergraph $H = (V, E, c, w)$, an ε-balanced k-way partition $\Pi = \{V_1, ..., V_k\}$, the objective to optimize $objective$ and an imbalance ε. The algorithm returns a refined partition Π. The hypergraph is coarsened from Line 1 to Line 4. Contractions are performed until the hypergraph is small enough or no valid contraction can be found. The only difference to the coarsening described in Section 3.1.1 is that only vertices which are in the same block are contracted. After coarsening the hypergraph is uncoarsened and refined with a local search algorithm. The refinement algorithm is initialized with the uncontracted vertices. V-cycles can be repeated to find better solutions.

Algorithm 5: k-way FM Local Search

```
Input: H = (V, E, c, w), Π = {V_1, ..., V_k}, B, ε
Result: Π' = {V_1, V_2}
1  ∀i ∈ {1, ..., k} : P_i ← priority queue of (v, g(v)) where v ∈ V with key g(v)
2  ∀i ∈ {1, ..., k} : enabled(P_i) := false
3  ∀v ∈ V : active(v) := false, marked(v) := false
4  for v ∈ B do
5  |  if λ(v) > 1 then
6  |  |  active(v) := true
7  |  |  for V_i ∈ R(v) \ {b[v]} do
8  |  |  |  P_i.add(v, g_i(v))
9  |  |  end
10 |  end
11 end
12 for i ∈ {1, ..., k} do
13 |  if V_i ≤ L_max ∧ ¬P_i.empty then
14 |  |  enabled(P_i) := true
15 |  end
16 end
17 while ∃i ∈ {1, ..., k} : enabled(P_i) := true ∧ ¬P_i.empty do
18 |  choose biggest g(v) with
   |  (v, g(v)) ∈ P_i, ∀i ∈ {1, ..., k} ∧ enabled(P_i) = true ∧ c(v) + c(V_i) ≤ L_max
19 |  active(v) := false
20 |  marked(v) := true
21 |  V_b[v] := V_b[v] \ {v}
22 |  V_i := V_i ∪ {v}
23 |  ∀i ∈ {1, ..., k} : P_i.remove(v, −)
24 |  for n ∈ Γ(v) do
25 |  |  if ¬marked(v) ∧ ¬enabled(v) then
26 |  |  |  active(n) := true
27 |  |  |  for V_i ∈ R(n) \ {b[n]} do
28 |  |  |  |  P_i.add(n, g_i(n))
29 |  |  |  end
30 |  |  end
31 |  |  if λ(n) = 1 then
32 |  |  |  ∀i ∈ {1, ..., k} : P_i.remove(n, −)
33 |  |  end
34 |  end
35 |  update the gain of all active neighbours n ∈ Γ(v)
36 end
```

Algorithm 6: Perform a V-Cycle

```
Input: H = (V, E, c, w), Π = {V_1, ..., V_k}, objective ∈ {λ^-1, cut}, ε
Result: Π = {V_1, ..., V_k}
1 while H is not small enough ∧ contraction can be performed do
2 |   (u, v) := argmax_{u∈V} score(u) ∧ u, v ∈ V_i
3 |   H := CONTRACT(H, u, v)
4 end
5 while H is not completely uncoarsened do
6 |   (H, Π, u, v) := UNCONTRACT(H, Π)
7 |   (H, Π) := LOCALSEARCH(H, Π, {u, v}, ε)
8 end
```

3.4 Recursive Bisection

A method to get an ε-balanced k-way partition of a hypergraph is to use recursive bisection [23]. Instead of performing coarsening, k-way initial partitioning and k-way local search during uncoarsening as described in Section 3.1, the hypergraph is partitioned in two blocks recursively. If a hypergraph should be divided in $k = 2^x, x \in \mathbb{N}$ blocks, a 2-way partition is calculated. Each block is bisected recursively until we have a k-way partition. This is only possible if k is a power of 2. If k is not a power of 2 the hypergraph is bisected into two blocks, where one block has a maximum weight of $(1+\varepsilon')\lceil\lfloor\frac{k}{2}\rfloor c(V)\rceil$ and the other a maximum weight of $(1+\varepsilon')\lceil\lceil\frac{k}{2}\rceil c(V)\rceil$. ε' is an adaptive imbalance, which ensures that the final partition is ε-balanced. $\varepsilon' = (1+\varepsilon)^{\frac{1}{\lceil\log_2(k)\rceil}} - 1$ is derived in [23].
Figure 3.2 illustrates the calculation of an ε-balanced 5-way partition of a hypergraph. H is the original hypergraph. H'_1, H'_2, H'_3, H''_1 and H''_2 are the subhypergraphs of H, which correspond to the final block. In Step 1 H is bisected into H_1 and H_2. Note that H_1 is divided into two subhypergraphs and block H_2 is divided into tree subhypergraphs. In Step 2 the H_1 is extracted and bisected. In Step 3 and Step 4 the final blocks H'_1 and H'_2 are extracted. Then H_2 is extracted and bisected. In Step 6 the final blocks H'_3 is extracted. Finally, H'_4 is bisected like in Step 2.
Algorithm 7 is the pseudo-code of the recursive bisection algorithm. The input parameters are a hypergraph $H = (V, E, c, w)$, an imbalance ε ,k_l the low index and k_h the high index of the blocks in which H should be partitioned. The hypergraph is partitioned in $k = k_h - k_l + 1$ blocks. The algorithm is initialized with the original hypergraph H, $k_l = 1$, $k_h = k$ and imbalance ε. RECURSIVEBISECTION$(H = (V, E, c, w), 1, k, \varepsilon)$ returns an ε-balanced k-way partition $\Pi = \{V_1, ..., V_k\}$. If $k_l = k_r$, H the recursion is stopped and V is a block of the final partition. In Line 6 the adaptive imbalance ε' is calculated according to [23]. To bisect the hypergraph the multilevel paradigm is applied. Furthermore, the subhypergraphs containing the block V_1 and V_2 are extracted with algorithm 8 described in Section 4.1. Note that the extraction of the subhypergraphs differs

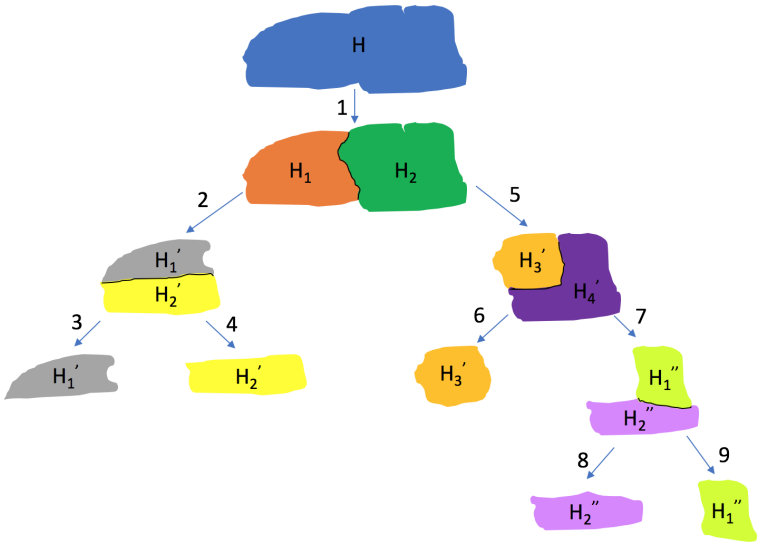


Figure 3.2: Example of calculating a k-way partition with recursive bisection

Algorithm 7: Recursive Bisection

Input: $H = (V, E, c, w), k_l, k_h, \varepsilon$
Result: $\Pi = \{V_{k_l}, ..., V_{k_h}\}$

$k := k_h - k_l + 1$
if $k = 1$ then
  return $\{V\}$
end
$\Pi_k := \{\}$
$\varepsilon' := (1 + \varepsilon)^{\frac{1}{\lceil \log_2(k) \rceil}} - 1$
while H *is not small enough* do
  $(u, v) := argmax_{u \in V} score(u)$
  $H := \text{CONTRACT}(H, u, v)$
end
$\Pi_2 = \{V_1, V_2\} := \text{RECURSIVEBISECTION}(H, \varepsilon')$
while H *is not completely uncoarsened* do
  $(H, \Pi_2, u, v) := \text{UNCONTRACT}(H, \Pi_2)$
  $(H, \Pi_2) := \text{LOCALSEARCH}(H, \Pi_2, \{u, v\}, \varepsilon')$
end
$m := \lceil \frac{k}{2} \rceil$
$\Pi_k := \Pi_k \cup \text{RECURSIVEBISECTION}(\text{EXTRACT}(H, \Pi_2, \{1\}, objective), k_l, k_l + m - 1, \varepsilon)$
$\Pi_k := \Pi_k \cup \text{RECURSIVEBISECTION}(\text{EXTRACT}(H, \Pi_2, \{2\}, objective), k_l + m, k_r, \varepsilon)$
return Π_2

with the current objective (see Section 4.1). Finally, these subhypergraphs are partitioned recursively.

4 Better Recursive Bisection Algorithm

In the next sections, we present several recursive bisection algorithms with k-way local search refinements. The idea of the following algorithms is to combine the advantages of k-way local search algorithms and recursive bisection presented in Section 3.2 and 3.4. For this purpose *local search* is applied to subhypergraphs during and/or after recursive bisections. Section 4.1 contains general corollaries and algorithms used to engineer the different refinement algorithms. We study six different approaches to refine the partitions with local search. The first approach is to apply a single local search on the partition after bisecting the hypergraph in k blocks. This algorithm is discussed in Section 4.2.
The second approach is to apply local search refinements on the nodes of the recursive bisection tree. We call this approach the *bottom-up* approach. First the hypergraph is partitioned with recursive bisection. Then local search is applied on the subhypergraphs created during the partitioning. The local search is first applied on the small subhypergraphs and finally on the original hypergraph. Figure 4.1 shows a recursive bisection tree and illustrates this approach. The blue arrows represent the bisection of the hypergraphs contained in the green boxes. The different colours of a hypergraph represent the different blocks of the partition. The local search can be applied to the hypergraphs and partitions contained in the green boxes. The refinement is done from the bottom up. This means hypergraph 3 is refined after the two subhypergraphs 1 and 2 have been refined. The algorithms following this scheme are explained in Section 4.3.
The third approach is to apply local search refinements directly after the bisection to unfinished partitions. The recursive bisection algorithm bisects the hypergraphs of the same depth in the recursive bisection tree before proceeding to the next depth. This means each hypergraph in the recursive bisection tree with depth $n \in \mathbb{N}$ is bisected before those of depth $n+1$. The bisections are calculated breadth first in the recursive bisection tree. After each bisection a k-way local search can be applied on the unfinished partition. The blocks of the unfinished partition are the hypergraphs in the recursive bisection tree with depth n or $n+1$, which still have to be partitioned or are leaves. This method is called the *top-down* approach.
Figure 4.2 represents a recursive bisection tree and illustrates the *top-down* approach. The nodes of the recursive bisection tree are hypergraphs and the blue arrows represent a bisection. The green and orange areas contain the blocks of a partition to which a local search can be applied. These partitions are numbered from 1 to 7 which represent the order of the

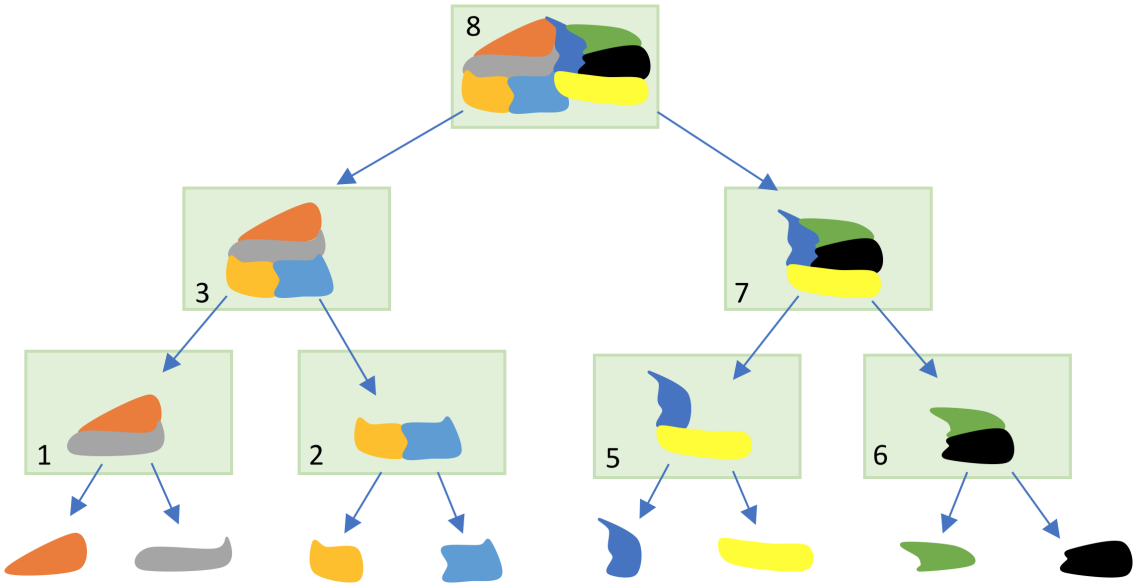


Figure 4.1: Illustration of the *bottom-up* approach for local search refinement.

local searches. Note that the blocks of a partition consist of hypergraphs same depth or a depth which differs by one. The refinements are done from the top down. The algorithms using the *top-down* approach are explained in Section 4.4.
The forth approach presented in Section 4.5 is to refine the k-way partition with 2-way local search. The partitions which should be refined are chosen by active block scheduling [21]. The fifth approach is to use 2-way local search with active block scheduling during uncoarsening instead of k-way local search to refine the partition after uncontraction. This algorithm is explained in Section 4.6. The final idea is to refine the partition with V-Cycles after the recursive bisection in Section 4.7.

4.1 Generals Concepts

To refine a subhypergraph with *local search*, it is extracted from the original hypergraph. Then the *local search* algorithm is applied and finally the original hypergraph is updated. The imbalance differs in the subhypergraph from the original hypergraph so ε has to be recalculated.

Corollary 4.1.0.1. *Let $H = (V, E, c, w)$ be a hypergraph with an ε-balanced partition $\Pi = \{V_1, ..., V_k\}$ and $H' = (V', E', c', w')$ be a subhypergraph of H with a partition $\Pi' = \{V'_1, ..., V'_{k'}\}$, $k' \leq k$. Then Π' has to be an ε'-balanced partition so that Π is an ε-balanced partition, with $\varepsilon' = (1+\varepsilon)\frac{\lceil \frac{c(V)}{k} \rceil}{\lceil \frac{c(V')}{k'} \rceil} - 1$.*

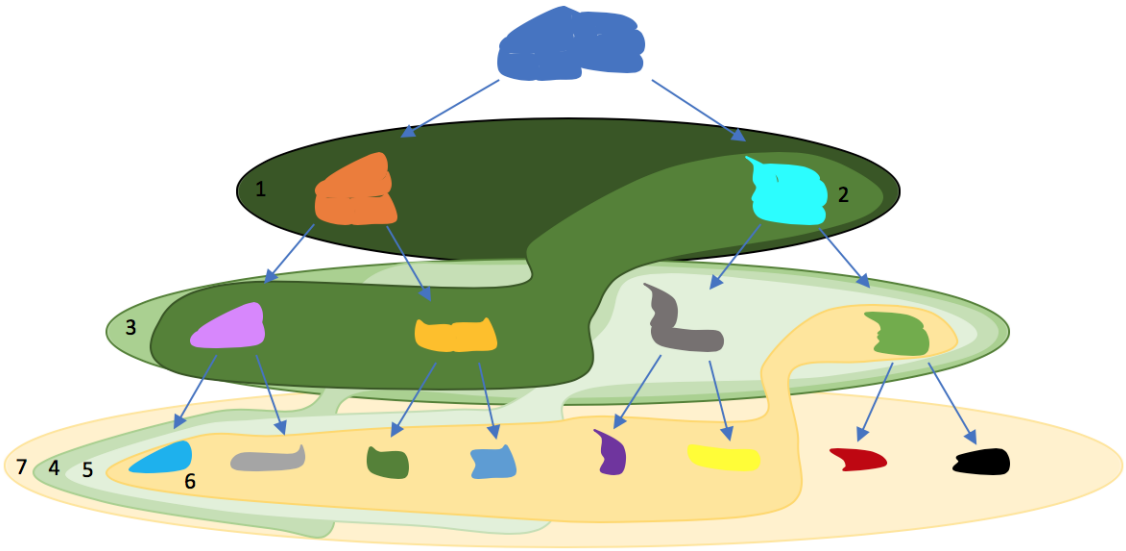


Figure 4.2: Illustration of the *top-down* approach for local search refinement.

Proof. We have to prove that Π is an ε-balanced partition if Π' is an ε'-balanced partition. $\Pi = \{V_1, ..., V_k\}$ with $\forall i \in \{1, ..., k\} : c(V_i) \leq (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$. $\Pi' = \{V'_1, ..., V'_{k'}\}$ is a subpartition of Π, so $\forall i \in \{1, ...k'\} : \exists j \in \{1, ..., k\}$ with $V'_i = V_j$.

$$\begin{aligned}
\forall i \in 1, ..., k' : c(V_j) = c(V'_i) &\leq (1+\varepsilon')\lceil \tfrac{c(V')}{k'} \rceil \\
&= (1 + (1+\varepsilon)\frac{\lceil \frac{c(V)}{k} \rceil}{\lceil \frac{c(V')}{k'} \rceil} - 1)\lceil \tfrac{c(V')}{k'} \rceil \\
&= (1+\varepsilon)\frac{\lceil \frac{c(V)}{k} \rceil}{\lceil \frac{c(V')}{k'} \rceil}\lceil \tfrac{c(V')}{k'} \rceil \\
&= (1+\varepsilon)\lceil \tfrac{c(V)}{k} \rceil
\end{aligned}$$

□

After extracting a subhypergraph *local search* is applied with ε'. Note that ε' can be larger than ε, but the overall balance remains.
Extracting a subhypergraph while optimizing the *cut-metric* differs from optimizing the λ^{-1}*-metric*. Let $H = (V, E, c, w)$ be a hypergraph and $H' = (V', E', c', w')$ be a subhybergraph, then the set of hyperedges E' depends on the metric. To optimize the *cut-metric* only hyperedges are taken into where all pins are completely inside V', $E' = \{e \in E \mid \forall v \in e : v \in V'\}$. To optimize the λ^{-1}*-metric*, *cut-net splitting* is used. A hyperedge e is contained in E' if all pins are completely inside V'. If not, e is split and a hyperedge e' is inserted in E' which contains all pins in e which are inside V'. Single pin hyperedges are deleted, so $E' = \{e' \mid \exists e \in E : e' = e \cap V' \wedge |e'| > 1\}$.
The following help function is used in different refinement algorithms.
Algorithm 8 is used to extract a subhypergraph $H' = (V', E', c', w')$ from a hypergraph

Algorithm 8: Extract a Subhypergraph

Input: $H = (V, E, c, w), \Pi = \{V_1, ..., V_k\}, p \subseteq \{1, ..., k\}, objective \in \{\lambda^{-1}, cut\}$
Result: $H' = (V', E', c', w'), \Pi' = \{V'_1, ..., V'_{k'}\}$

```
1  V' := {v ∈ V | v ∈ V_i, i ∈ p}
2  if objective = λ^-1 then
3  |  E' := {e' | ∃e ∈ E : e' = e ∩ V' ∧ |e'| > 1}
4  else
5  |  E' := {e ∈ E | ∀v ∈ e : v ∈ V'}
6  end
7  c' := c_|V'
8  w' := w_|E'
9  for i := 1, i <= k, i := i + 1 do
10 |  V_i := V_p[i]
11 end
```

Algorithm 9: Recursive Bisection with one k-way Local Search Refinement after Partitioning

Input: $H = (V, E, c, w), k \in \mathbb{N}, \varepsilon, objective \in \{\lambda^{-1}, cut\}$
Result: $\Pi = \{V_1, ..., V_k\}$

```
1  Π' = {V'_1, ..., V'_k} := RECURSIVEBISECTION(H, 1, k, ε, objective)
2  B := {e ∈ E | λ(e) > 1}
3  Π := LOCALSEARCH(H, Π', B, ε, objective)
```

$H = (V, E, c, w)$. $\Pi = \{V_1, ..., V_k\}$ is the current partition. $p \subseteq \{1, ..., k\}$ is a set of block identifiers which shall be extracted as subhypergraph. The metric which should be optimized is $objective$. The subhypergraph is computed by the rules described in 4.1. $\Pi' = \{V'_1, ..., V'_{k'}\}$ are the partitions of the extracted hypergraph with $k' = |p|$.

4.2 Single Local Search after Recursive Bisection

A simple approach to combine k-way local search and recursive bisection is to apply a k-way local search after the partitioning. First the hypergraph is partitioned with the recursive bisection algorithm explained in Section 3.4. Then a k-way local search is applied to refine the partition. Algorithm 9 is the pseudo-code of this algorithm. First the partition is calculated. Then it is refined with a k-way local search. The local search algorithm is initialized with the border nodes B.

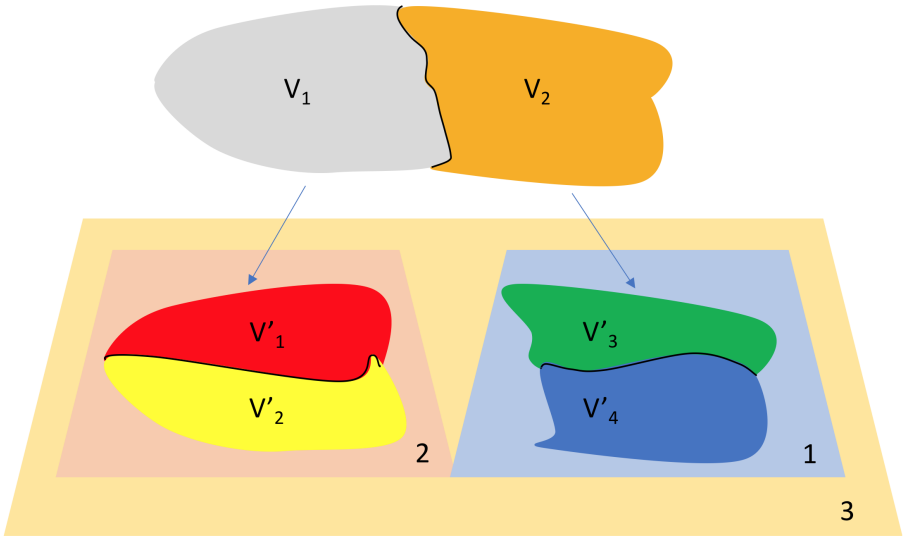


Figure 4.3: Example of local search at the nodes of recursive bisection tree. The arrows represent bisections. V_1', V_2', V_3' and V_4' are the blocks of a 4-way partition. The blocks in Box 1,2,3 can be refined with k-way local search.

4.3 Local Search at the Nodes of the Recursive Bisection Tree

The following algorithms use the *bottom-up* approach. First the partition is calculated with the recursive bisection algorithm. The k-way local search refinement is applied at the nodes of the recursive bisection tree from the bottom up (see Figure 4.1). The following algorithm applies a k-way local search after the hypergraph has been partitioned recursively by the recursive bisection. After each recursive call of the recursive bisection function, the partition is refined. This can lead to the exchange of vertices between blocks, which have not been considered during partitioning.
Algorithm 10 is a modification of the recursive bisection algorithm. The instructions in Line 1 can be found in Section 3.4. After the recursive call RECURSIVEBISECTION in Line 3 and 4 a k-way local search is applied to the hypergraph H with partition Π_k. Note that Π_k contains blocks which are not bisected further. An example for calculating a 4-way partition of a hypergraph with local search refinement at the nodes of the recursive bisection tree can be seen in Figure 4.3. First the hypergraph is divided into two blocks V_1 and V_2. Secondly, V_2 is recursively partitioned into V_3' and V_4'. Then a local search is applied on the partition containing V_3' and V_4' (Box 1 in Figure 4.3). Then the same procedure is repeated with V_2. Finally, after the 2-way local search in Box 1 and Box 2 a 4-way local search is applied at the partition containing V_1', V_1',V_3' and V_4'.
The general idea of the next algorithms is to apply the k-way local search at the same subpartitions as Algorithm 10. Since the local search refinements are only performed par-

Algorithm 10: Local Search Refinement during Recursive Bisection

Input: $H = (V, E, c, w), k_l, k_h, \varepsilon$
Result: $\Pi = \{V_{k_l}, ..., V_{k_h}\}$

1 ...
2 $m := \lceil \frac{k}{2} \rceil$
3 $\Pi_k := \Pi_k \cup$ RECURSIVEBISECTION(EXTRACT$(H, \Pi_2, \{1\}, objective), k_l, k_l + m - 1, \varepsilon)$
4 $\Pi_k := \Pi_k \cup$ RECURSIVEBISECTION(EXTRACT$(H, \Pi_2, \{2\}, objective), k_l + m, k_r, \varepsilon)$
5 $(H' := (V', E', c', w'), \Pi' = \{V_{k_l}, ..., V_{k_h}\}) =$ EXTRACT$(H, \Pi, \{k_l, ..., k_h\}, objective)$
6 $B := \{e \in E' \mid \lambda(e) > 1\}$
7 $\varepsilon' := (1 + \varepsilon) \frac{\lceil \frac{c(V)}{k} \rceil}{\lceil \frac{c(V')}{k'} \rceil} - 1$
8 $\Pi_k :=$ LOCALSEARCH$(H', \Pi_k, B, \varepsilon', objective)$
9 return Π_k

titions containing blocks which are not bisected further, we can refine the partition after the recursive bisection. By applying the k-way local search after the partitioning, we can change the order of the refinements. Furthermore, the refinement of subpartitions can be repeated. In Section 4.3.1 we introduce an algorithm which performs the same refinements as Algorithm 10 after the recursive bisection. The algorithms described in Section 4.3.2 and Section 4.3.3 repeat the local search at the nodes of the recursive bisection tree. The basic idea is to exploit the fact that a k-way local search refinement at subpartitions can lead to further improvements at larger partitions.

4.3.1 Single Local Search at each Node

The following algorithm applies k-way local search refinements at the nodes of the recursive bisection tree. The refinements are performed from the bottom up and in the same order as described in the previous section. For this reason Algorithm 11 descends in the recursive bisection tree and applies the k-way local search refinement while climbing up. In other words, the algorithm starts with an ε-balanced k-way partition Π computed by a recursive bisection algorithm. This partition divided into two partitions Π'_1 and Π'_2. These subpartitions correspond to the subhypergraphs computed by the first bisection of the recursive bisection algorithm. This procedure is repeated until the subpartitions contain only one block. During the recursive rise, the subpartitions are refined with k-way local search until Π'_1 and Π'_2 are refined. Finally, Π is refined and the algorithm ends.
Algorithm 11 is the pseudo-code of this algorithm. The recursive descend ends if the partition contains only one block because no local search can be applied. The partition $\Pi = \{V_{k_l}, ..., V_{k_r}\}$ is divided into two subpartitions $\Pi'_1 = \{V_{k_l+m}, ..., V_{k_r}\}$ and $\Pi'_2 = \{V_{k_l}, ..., V_{k_l+m-1}\}$ where $m = \lceil \frac{k}{2} \rceil$. Then Π'_1 and Π'_2 are refined recursively in Line 4 and 5. After the recursive refinement Π is refined in Line 9. Algorithm 12 combines the

Algorithm 11: Refinement at Nodes of the Recursive Bisection Tree

Input: $H = (V, E, c, w), \Pi = \{V_1, ..., V_k\}, \varepsilon, k_l \in \mathbb{N}, k_r \in \mathbb{N}, objective \in \{\lambda^{-1}, cut\}$
Result: $\Pi = \{V_1, ..., V_k\}$

RECURSIVEREFINEMENT$(H, \Pi, \varepsilon, k_l, k_r, objective)$
1 $k := k_r - k_l + 1$
2 if $k >= 2$ then
3 $\quad m := \lceil \frac{k}{2} \rceil$
4 $\quad$ RECURSIVEREFINEMENT$(H, \Pi, \varepsilon, k_l + m, k_r, objective)$
5 $\quad$ RECURSIVEREFINEMENT$(H, \Pi, \varepsilon, k_l, k_l + m - 1, objective)$
6 $\quad (H' = (V', E', c', w'), \Pi' = \{V_{k_l}, ..., V_{k_r}\}) :=$
$\quad$ EXTRACT$(H, \Pi, \{k_l, ..., k_r\}, objective)$
7 $\quad B := \{e \in E' \mid \lambda(e) > 1\}$
8 $\quad \varepsilon' := (1 + \varepsilon) \frac{\lceil \frac{c(V)}{k} \rceil}{\lceil \frac{c(V')}{k'} \rceil} - 1$
9 $\quad \Pi' :=$ LOCALSEARCH$(H', \Pi', B, \varepsilon', objective)$
10 $\quad \Pi := (\Pi \setminus \{V_{k_l}, ..., V_{k_r}\}) \cup \Pi'$
11 end

Algorithm 12: Recursive Bisection with Refinement at the Nodes of the Recursive Bisection Tree

Input: $H = (V, E, c, w), k \in \mathbb{N}, \varepsilon, objective \in \{\lambda^{-1}, cut\}$
Result: $\Pi = \{V_1, ..., V_k\}$
1 $\Pi' = \{V'_1, ..., V'_k\} :=$ RECURSIVEBISECTION$(H, 1, k, \varepsilon, objective)$
2 $\Pi = \{V_1, ..., V_k\} :=$ RECURSIVEREFINEMENT$(H, \Pi', \varepsilon, 1, k, objective)$

recursive bisection and recursive refinement. First the recursive bisection algorithm calculates an ε-balanced k-way partition Π' in Line 1. Then the partition is refined with the recursive refinement algorithm 11 in Line 2. The algorithm returns an ε-balanced k-way partition Π.
Note that Algorithm 12 calculates the same ε-balanced k-way partition and applies the k-way local search at the same subpartitions as the modified recursive bisection algorithm presented in the previous section. Since we separated the partitioning and the refinement, we can now execute the Algorithm 11 multiple times (see Section 4.3.2).

4.3.2 Repeated Local Search

The basic idea of the following algorithm is to apply k-way local search at the same subpartitions multiple times. First the ε-balanced k-way partition is calculated by the recursive bisection algorithm. Then this partition is refined at the nodes of the recursive bisection tree. For this purpose we use the recursive refinement algorithm of Section 4.3.1. We re-

Algorithm 13: Recursive Bisection with Repeated Refinement at the Nodes of the Recursive Bisection Tree

Input: $H = (V, E, c, w), k \in \mathbb{N}, \varepsilon, objective \in \{\lambda^{-1}, cut\}, max \in \mathbb{N}$
Result: $\Pi = \{V_1, ..., V_k\}$
1 $\Pi = \{V_1, ..., V_k\} :=$ RECURSIVEBISECTION$(H, 1, k, \varepsilon, objective)$
2 do
3 $\quad m :=$METRIC$(H, \Pi, objective)$
4 $\quad \Pi = \{V_1, ..., V_k\} :=$ RECURSIVEREFINEMENT$(H, \Pi, \varepsilon, 1, k, objective)$
5 $\quad improvement := m-$METRIC$(H, \Pi, objective)$
6 $\quad max := max - 1$
7 while $(improvement > 0) \wedge (max \neq 0)$

peat this refinement multiple times. It is possible that no k-way local search at the nodes of the recursive bisection tree with depth $d > 0$ could find an improvement. If the k-way local search at the root node with depth $d = 0$ improves the partition, then a second refinement at the nodes with depth $d > 0$ can lead to improvements. This algorithm exploits the fact that after k-way local search at a node with depth d can change the partition so that local searches at nodes with depth $d > d'$ can find better results.
Consider a refinement of the hypergraph in Figure 4.3. The 2-way local search of the subpartitions in Box 1 and 2 does not improve the current objective. However, the 4-way local search of the original partition in Box 3 can find a better result. The 4-way local search improved the partition, so vertices have been exchanged between blocks. So by repeating the local search of the subpartitions in Box 1 and 2 it is possible to find an improvement. On top of that if the 2-way local search in Box 1 and 2 improved the objective, the 4-way local search in Box 3 can find an improvement a second time.
First Algorithm 13 computes an ε-balanced k-way partition with the recursive bisection algorithm. Then the recursive refinement algorithm of Section 4.3.1 is applied until no improvement can be found or the algorithm has been applied max times. The metric (cut or $connectivity$) is calculated before and after the refinement to decide if the partition has improved.

4.3.3 Preferred Local Search on Small Subpartitions

Performing a k-way local search on a big partition is slower than performing a k'-way partition on a smaller partition, where $k > k'$. We adopt the same idea as in Section 4.3.2 to refine a partition at the nodes of the recursive bisection tree multiple times. However, the following algorithm prefers to refinements at nodes with maximum depth. So we prefer k-way local search on smaller partitions, which have a better running time. For this purpose we modify the recursive refinement algorithm described in Section 4.3.1. The algorithm starts with an ε-balanced k-way partition Π computed by a recursive bisection algorithm.

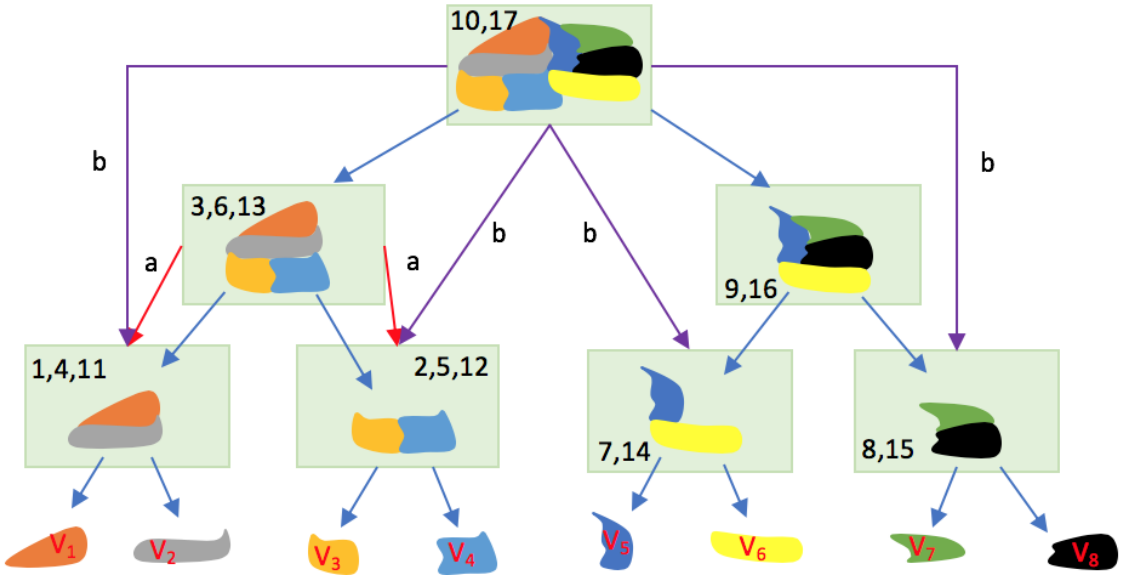


Figure 4.4: Illustration the recursive refinement of the nodes of the recursive bisection tree. Local searches at nodes with higher depth are preferred. Step 1 to 17 correspond to the order of the k-way local search refinements. The arrows a and b correspond to an additional recursive refinement after a local search has improved the objective. In Step 3 and 10 the k-way local search improves the partition.

First Π is divided into two subpartitions Π'_1 and Π'_2, which correspond to the subhypergraphs of a bisection in the recursive bisection algorithm. In other words, Π'_1 and Π'_2 are represented by two nodes in the recursive bisection tree. Secondly, Π'_1 and Π'_2 are refined recursively. Then a k-way local search is applied at Π. If this local search improved the objective then Π'_1 and Π'_2 are refined recursively again. The algorithm ends if the k-way local search at the node with depth $d = 0$ cannot improve the partition.
Figure 4.4 illustrates this procedure. The green boxes contain the hypergraphs and partitions of a recursive bisection tree on which the local search can be applied. In this example the local search is applied first in Step 1 and 2. Then the refinement in Step 3 finds an improvement and we recursively refine the partition in Step 1 and 2 again. Furthermore, the refinements in Step 4 to 9 do not find an improvement and so the original hypergraph with its partition is refined in Step 10. This local search finds an improvement and so the partition of the original hypergraph is recursively refined. Since the local searches from Step 11 to 17 do not find any improvements, the refinement is terminated.

The pseudo-code of this algorithm can be found in Algorithm 14. The partition $\Pi = \{V_{k_l}, ..., V_{k_r}\}$ is divided into two subpartitions $\Pi'_1 = \{V_{k_l+m}, ..., V_{k_r}\}$ and $\Pi'_2 = \{V_{k_l}, ..., V_{k_l+m-1}\}$. Then they are recursively refined in Line 4 and 5. A k-way local search is applied in Line 10 on Π. If this local search improved the partition, Π'_1 and Π'_2 are recursively improved again in Line 14. This leads to potentially more local search refinements at smaller hypergraphs and few large ones.

Algorithm 14: Refinement at the Nodes of the Recursive Bisection Tree, while Preferring k-way Local Search on small Subpartitions

```
Input: H = (V, E, c, w), Π = {V_1, ..., V_k}, ε, k_l ∈ ℕ, k_r ∈ ℕ, objective ∈ {λ^{-1}, cut}
Result: Π = {V_1, ..., V_k}
RECURSIVEREFINEMENTTREE (H, Π, ε, k_l, k_r, objective)
k := k_r − k_l + 1
if k >= 2 then
    m := ⌈k/2⌉
    RECURSIVEREFINEMENTTREE(H, ε, k_l + m, k_r, objective)
    RECURSIVEREFINEMENTTREE(H, ε, k_l, k_l + m − 1, objective)
    (H' = (V', E', c', w'), Π' = {V_{k_l}, ..., V_{k_r}}) :=
    EXTRACT(H, Π, {k_l, ..., k_r}, objective)
    B := {e ∈ E' | λ(e) > 1}
    ε' := (1 + ε) ⌈c(V)/k⌉ / ⌈c(V')/k'⌉ − 1
    old_metric :=METRIC(H', Π', objective)
    Π' := LOCALSEARCH(H', Π', B, ε', objective)
    improvement := old_metric−METRIC(H', Π', objective)
    Π := (Π \ {V_{k_l}, ..., V_{k_r}}) ∪ Π'
    if improvement > 0 then
        RECURSIVEREFINEMENTTREE(H, ε, k_l, k_r, objective)
    end
end
```

4.4 Local Search on Unfinished Partitions

In the following section, we present algorithms which use the *top-down* approach. In the previous algorithms, the recursive bisection tree was created depth first. This means after each bisection we bisect a hypergraph with maximum depth, which is no leaf of the recursive bisection tree. The recursive bisection tree is created in the same order as it would be traversed by a *depth first search*. However, during a *top-down* approach, the k-way local searches refinements are performed on unfinished partitions. The blocks of an unfinished partition correspond to the nodes of the recursive bisection tree, whose depths differ by one. In other words,the recursive bisection has to bisect the hypergraph in the same order as the *breadth first search* would traverse the recursive bisection tree.
This alternative recursive bisection algorithm maintains a set S of hypergraphs which still have to be bisected. S is initialized with the input hypergraph and the depth d is set to zero.

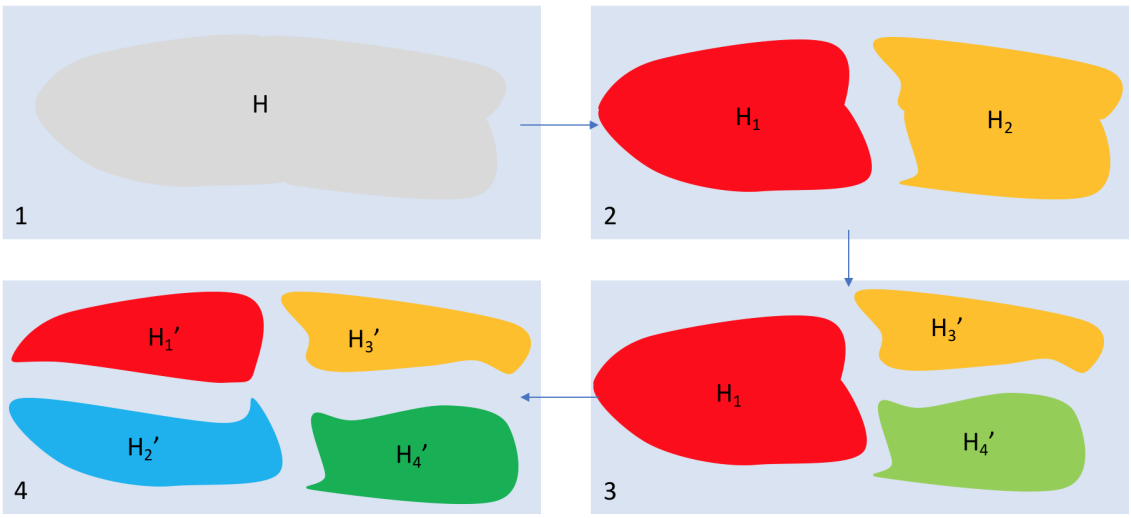


Figure 4.5: Example of the the alternative recursive bisection algorithm.

The algorithm successively removes a hypergraph $H \in S$ with depth d. H is bisected into H_1 and H_2. Note that the depth of H_1 and H_2 is $d + 1$. If they do not correspond to a leaf in the recursive bisection tree, they are added to S. If S does not contain a hypergraph with depth d, then d is increased by one. The algorithm stops if S is empty. Figure 4.5 illustrates the computation of an ε-balanced k-way partition with this algorithm. First the set $S = \{H\}$ contains the input graph which has depth $d = 0$ (see Step 1). Then H is removed and replaced by his subhypergraphs H_1 and H_2 which have depth $d' = 1$. So in Step 2 $S = \{H_1, H_2\}$. No hypergraph of depth $d = 0$ is contained in S and so the next hypergraph which should be bisected had depth $d' = 1$. In Step 3 H_2 is bisected into H'_3 and H'_4. Since the vertices of H'_3 and H'_4 are blocks of the final partition, they are not added to S. After Step 3 S only contains H_1. During the last Step 4 H_1 is bisected into H'_1 and H'_2 and the algorithm ends. Note that H'_1, H'_2, H'_3 and H'_4 have depth $d'' = 2$ and are leaves of the recursive bisection tree.
The following algorithms are going to refine the partitions which correspond to the set S. In Figure 4.5 the k-way local searches can be applied in Step 2,3 and 3, on the partitions corresponding to $\{H_1, H_2\}$, $\{H_1, H'_3, H'_4\}$ and $\{H'_1, H'_2, H'_3, H'_4\}$.

Local search refinement after each bisection. The following algorithm uses the *top-down* approach. The basic idea is to bisect the input hypergraph successively with the alternative recursive bisection algorithm described in Section 4.4. After each bisection a k-way local search is applied on the unfinished partition. First the set S is initialized with the triple $(V, 1, k)$, where V is the vertex set of the hypergraph and k is the number of blocks of the final partition. The following process is repeated until S contains a final partition. Note that S contains vertex sets corresponding to hypergraphs which have the same depth in the recursive bisection tree. For each $(V_i, k_l, k_r) \in S$ the number of blocks

Algorithm 15: Refinement after each Bisection

```
Input: H = (V, E, c, w), k ∈ ℕ, ε, objective ∈ {λ^-1, cut}
Result: Π = {V_1, ..., V_k}
1  S := {(V, 1, k)}                                   // S is initialized with the vertices of H
2  while |S| ≠ k do                                   // If S contains k vertex sets, then S is the final partition
3      S' := ∅                                        // We use S' to temporarily store vertex sets after bisection
4      for (V_i, k_l, k_r) ∈ S do                     // Iterate over S
5          Π := S ∪ S'                                // Π is the unfinished partition before bisection
6          S := S \ {(V_i, k_l, k_r)}                 // Remove the vertex set which is partitioned from S
7          k := k_r − k_l + 1                         // k is the numer of blocks in which V_i has to be partitioned
8          if k = 1 then                              // V_i is a block of the final partition
9              S' := S' ∪ {(V, E, k)}                 // V_i is not bisected
10         else
11             m := ⌈k/2⌉
12             H' := EXTRACT(H, Π, (, k_l, k_r))
13             {V_1, V_2} := COMPUTEBISECTION(H, , 2, ε, objective)   // Bisect V_i
14             S' := S' ∪ {(V_1, k_l, k_l + m − 1), (V_2, k_l + m, k_r)}
15             Π := S ∪ S'                            // Π is the unfinished partition
16             B := {e ∈ E | λ(e) > 1}                // B contains the border hyperedges
17             (H, Π) := LOCALSEARCH(H, Π, B, ε)      // Appy local search at Π
18             update S and S' according to the improved partition Π
19     S := S'
20 Π := S
```

$k = k_r - k_l + 1$ in which V_i shall be partitioned is calculated. If $k = 1$, then V_i is a block of the final partition and corresponds to a leaf in the recursive bisection tree. If $k > 1$, then V_i is bisected and added to the set S'. S' contains the vertex sets which correspond to the nodes of the recursive bisection tree, whose depth is bigger by one than that of S. Note that all vertices of V are contained in S or S'. Then a k-way local search is applied on the unfinished partition $\Pi = S \cup S'$. After iterating over all elements of S and removing them S' contains the vertex sets which are bisected next. Algorithm 15 is the pseudo-code of this algorithm.

Reducing the number of local searches. The previous algorithm performs a k-way local search algorithm after each bisection on a partition which contains all the vertices of the input hypergraph. This results in $O(k)$ local search over large partitions. We modify Algorithm 15 so that only $O(\log(k))$ k-way local searches are performed. The first approach is to apply the local search only on partitions which contain blocks that correspond

to nodes of the recursive bisection tree with same depth. So after bisecting each vertex set in S the k-way local search is performed on S'. In Algorithm 15 the local search will be applied in Line 19 instead of Line 17. Since the height of the recursive bisection tree lies in $O(\log(k))$, only $O(\log(k))$ local searches are performed. The second approach is to apply the k-way local searches at the end of the alternative recursive bisection. The algorithm computes the partition like Algorithm 15. Instead of performing a k-way local search after each bisection in Line 17, only the last $w \in O(\log(k))$ local searches are applied. In other words, the algorithm starts the local search refinement after $k - w$ bisections have been computed. The number w of k-way local searches is a tuning parameter.

4.5 2-Way Local Search with Active Block Scheduling

The following algorithm refines an ε-balanced k-way partition after recursive bisection similar to the algorithms in Section 4.3. Instead of applying a k-way local search algorithm on subhypergraphs, we apply a 2-way local search algorithm between two adjacent blocks. The main idea is to perform several fast 2-way local searches instead of few slow k-way local searches. For this reason we construct the quotient graph $Q = (Qv, Qe)$ of the input hypergraph $H = (V, E, c, w)$ and the partition $\Pi = \{V_1, ..., V_k\}$. The quotient graph is used to schedule the 2-way local search algorithms. We use active block scheduling [21, 12]. First all blocks are set active. The algorithm works in passes. In each pass we iterate over all adjacent blocks $\{V_1, V_2\}$, where at least one is active, and we apply a 2-way local search. Blocks which have not changed during a pass are considered inactive for the next pass, so blocks which do not change are avoided. The algorithm ends if all blocks are inactive. The pseudo-code can be found in Algorithm 16.

4.6 Active Block Scheduling During Uncoarsening

The following algorithm combines the 2-way local search algorithm with active block scheduling and the multilevel paradigm. For this purpose a hypergraph is coarsened, then an initial ε-balanced k-way partition is calculated. During uncoarsening no k-way local search algorithm is applied but the active block scheduling algorithm 16.
Algorithm 17 computes an ε-balanced k-way partition Π of a hypergraph H optimizing the metric $objective$. In Line 1, H is coarsened according to Section 3.1.1. Then an initial partition is calculated in Line 5 according to Section 3.1.2. Finally, the hypergraph is uncoarsened again according to Section 3.1.3 in Line 6. During the uncoarsening the partition Π is refined with the active block scheduling refinement algorithm 16 in Line 8. The 2-way local searches in algorithm 16 are initialized with the uncoarsened vertices $\{u, v\}$.

Algorithm 16: 2-way Local Search Refinement with Active Block Scheduling

```
Input: H = (V, E, c, w), Π = {V_1, ..., V_k}, ε, objective ∈ {λ^-1, cut}
Result: Π = {V_1, ..., V_k}
1  Qv := {V_1, ..., V_k} = Π
2  Qe := {{a, b} ⊆ Qv | ∃e ∈ E : {a, b} ⊆ Λ(e)}
3  Q := (Qv, Qe)                                   // Quotient graph of H and Π
4  ∀v ∈ Qv : active(v) := true                     // All blocks are set active
5  while ∃v ∈ Qv : active(v) = true do             // Stop the refinement if all blocks are inactive
6      ∃v ∈ Qv : active'(v) := false               // All blocks are set inactive for the next pass
7      foreach {a, b} ∈ Qe ∧ (active(a) ∨ active(b)) do   // Iterate over all adjacent
                                                   // blocks where at least one is active
           Π' := {a, b}
8          (H' = (V', E', c', w'), Π' = {V_i, V_j}) :=
           EXTRACT(H, Π, {id(a), id(b)}, objective)
9          B := {e ∈ E' | λ(e) > 1}
10         if |B| = 0 then                         // If there are no border vertices
                                                   // no refinement has to be done
               continue
11         ε' := (1 + ε) ⌈c(V)/k⌉ / ⌈c(V')/k'⌉ − 1
12         old_metric := METRIC(H', Π', objective)
13         Π' := LOCALSEARCH(H', Π', B, ε', objective)   // Apply the 2-way local search
14         improvement := old_metric − METRIC(H', Π', objective)
           Π := (Π \ {a, b}) ∪ Π'
15         if improvement > 0 then                 // If the objective has improved a and b
                                                   // are set active for the next pass
               active'(a) := true
16             active'(b) := true
17     active := active'
```

Algorithm 17: Direct k-way Partitioning with Active Block Scheduling During Uncoarsening

```
Input: H = (V, E, c, w), ε, k ∈ ℕ, objective ∈ {λ^-1, cut}
Result: Π = {V_{k_l}, ..., V_{k_h}}
while H is not small enough do
    (u, v) := argmax_{u∈V} score(u)
    H := CONTRACT(H, u, v)
end
Π' = {V_1, ..., V_k} := COMPUTEINITIALPARTITION(H, ε, k)
while H is not completely uncoarsened do
    (H, Π, u, v) := UNCONTRACT(H, Π)
    Π := PAIREWISEREFINEMENT(H, Π, ε, objective, {u, v})
end
```

Algorithm 18: Perform V-Cycles after Recursive Bisection

```
Input: H = (V, E, c, w), k ∈ ℕ, ε, objective ∈ {λ^-1, cut}, max ∈ ℕ
Result: Π = {V_1, ..., V_k}
1  Π = {V_1, ..., V_k} := RECURSIVEBISECTION(H, 1, k, ε, objective)
2  do
3      old_metric :=METRIC(H, Π, objective)
4      Π := PERFORMVCYCLE(H, Π, objective, ε)
5      improvement := old_metric−METRIC(H, Π, objective)
6      max := max − 1
7  while (improvement > 0) ∧ (max > 0)
```

4.7 Recursive Bisection with V-Cycle Refinement

Similar to the previous algorithms we want to optimize an ε-balanced k-way partition calculated by a recursive bisection algorithm. The following Algorithm 18 computes a partition and then refines it with V-Cycles. V-Cycles are explained in Section 3.3. The input parameters of Algorithm 18 of are a hypergraph H, the number of blocks k, the imbalance ε, the metric to optimize $objective$ and the maximum number of V-Cycles max. The result is an ε-balanced k-way partition Π. First an ε-balanced k-way partition is calculated by the recursive bisection algorithm in Line 1. Then Π is repetitively refined by V-Cycles 6 in Line 4. If no improvement could be found or the maximum number of V-Cycles is reached, the algorithm stops.

4.8 Algorithm Overview

Table 4.1 contains an overview of the algorithms presented in 4 with a small description and a reference to the detailed description.

Table 4.1: This table contains an overview of the different algorithms described in the previous sections.

Algorithm	Description	Reference
KaHyPar-R	recursive bisection	Algorithm 7
RB+OL	recursive bisection with one local search at the end	Algorithm 9
RB+LD	recursive bisection with local search at the nodes during partitioning	Section 4.3
RB+L	recursive bisection with local search at the nodes of the recursive bisection tree after partitioning	Algorithm 12
RB+LC	recursive bisection with repeated local search at the nodes of the recursive bisection tree after partitioning	Algorithm 13
RB+LT	recursive bisection with preferred local search at the bottom nodes of the recursive bisection tree	Section 4.3.3
RBA	alternative recursive bisection	Section 4.4
RBA+L	$O(k)$ local search on unfinished partitions	Section 15
RBA+LL	$O(\log(k))$ local search on unfinished partitions with the same depth in the recursive bisection tree	Section 4.4
RBA+LLE	$O(\log(k))$ local search at the end on unfinished partitions	Section 4.4
RB+AB	recursive bisection with active block scheduling refinement	Algorithm 16
KaHyPar-K+AB	direct k-way partitioning with active block scheduling during uncoarsening	Algorithm 17
RB+LVC	recursive bisection with V-Cycle refinement after partitioning	Algorithm 18

5 Experimental Evaluation

The algorithms introduced in Section 4 use k-way local search to refine an ε-balanced k-way partition computed with recursive bisection. To evaluate these algorithms we computed k-way partitions of a set of hypergraphs with the different algorithms. In the following sections we present the results of these experiments. Section 5.1 explains the experimental environment, the tuning parameters and the test instances. We present the different statistics we used to analyse the experimental results in Section 5.2. Finally, in Section 5.3 we discuss the experimental results. To get an overview of the different algorithms, we compute them on a hypergraph set containing 25 hypergraphs. This allows us to determine the algorithms, on which we perform further tests. Secondly, we evaluate the results of a larger set. Then we compare two of our best algorithms to the state-of-the-art partitioner. An overview of the different algorithms can be found in Table 4.1. The plots are created with R (version 3.5.0).

5.1 Experimental Setup

5.1.1 Environment

The experiments are performed on the BwUniCluster [1, 2]. For each hypergraph and algorithm we associate a single node with Intel Xeon E5-2670 (Sandy Bridge) processors, a processor frequency of 2.6 GHz, 2 sockets, 16 cores, 64 GB of main memory and a local disc with 2 TB [2]. Each socket has a tree level cache with 8x64 KB memory at level 1, 8x256 KB at level 2 and 20 MB at level 3. Each node has an adapter to connect to the InfiniBand 4X FDR interconnect. The different nodes are connected by an InfiniBand 4X FDR interconnect. The operating system of each node is Red Hat Enterprise Linux (RHEL) 7.4. The maximum running time for the different algorithms is one day.

5.1.2 Tuning Parameters

An experiment instance is a hypergraph, an algorithm and the different parameters of the algorithm. Each experiment instance is calculated on a single node of the BwUniCluster with a maximum main memory of 62 GB. The hypergraphs are partitioned with recursive bisection. The coarsening algorithm is the heavy lazy algorithm. The coarsening rating

function is the heavy edge rating function described in Section 3.1.1. Coarsening is performed until $160k$ vertices are left, where k is the number of blocks. During uncoarsening the partition is refined with 2-way local search. The local search during uncoarsening is stopped if there have been 350 fruitless moves in a row during refinement. A fruitless move is a move that could not improve the current best objective. The initial partition is calculated like described in Section 3.1.2.

5.1.3 Instances

The instances used to test the algorithms in Section 4 are a subset of a hypergraph benchmark set [22]. This benchmark set contains instances from the benchmark sets of the ISPD98 VLSI Circuit Benchmark Suite [5], the University of Florida SuiteSparse Matrix Collection [8] and the international SAT Competition 2014 [4]. The weight of the vertices and nets are one. The full benchmark set and subset can be found on the website of KaHyPar-CA [3]. Table A.1 lists 25 hypergraphs used to get an overview of the different algorithms, while Table A.2 lists 107 hypergraphs to perform further tests.
The hypergraph in the benchmark set can be divided into six types [12, 13]. The *DAC* and *ISPD* hypergraphs are VLSI instances and their vertices have a low average degree and their nets a low average size. On the other side, the *SPM* instances have a high average vertex degree and net size. *Primal*, *literal* and *dual* are three different SAT representations. *Primal* instances have a large vertex degree and small net size, while the opposite is the case for *dual* instances. *Literal* instances have a smaller average vertex degree than *primal* instances, but it is smaller than those of *dual* instances. The net size of *literal* instances is small.
Since the implementation of the algorithms includes randomization each hypergraph is partitioned ten times with the same algorithm with different initial seeds $s \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Each hypergraph is partitioned in $k \in \{4, 8, 16, 32, 64, 128, 256, 512, 1024\}$ blocks. The imbalance of the output partition is $\varepsilon = 0.03$.

5.2 Statistics

To evaluate the different algorithm we use the performance plots introduced in the recursive bisection paper of KaHyPar [23]. The plot relates the smallest average and minimum objective of all algorithms to the corresponding objective of each algorithm on a per-instance base. The metric can be *cut* or *connectivity*. The average metric is the average over the metrics of the different seeds and the minimum metric is the minimum over the metrics for the different seeds. For each algorithm we calculate for each instance $1 - best/algorithm$ where $best$ is the best algorithm for the instance. Then these ratios are sorted in decreasing order. To reduce right skewness the plots use a cube root scale for y axis. Note that the values at the y axis are between 0 and 1. If an algorithm has a value of 0 for an instance, it

has produced the best partition. If the value is close to 1 the algorithm has produced a bad partition compared to the others. An algorithm is considered to produce better partitions than another if its values are below those of the other.
Another method to compare two algorithms is to calculate the improvement from one algorithm to the baseline. We distinguish between average improvement and minimum improvement. The improvement is calculated as follows: $(1 - \frac{g(algorithm.metric)}{g(baseline.metric)}) * 100$ where g is the geometric mean. $algorithm.metric$ and $baseline.metric$ are the metrics (*cut* or *connectivity*) of the different instances of the algorithm and the baseline. The average improvement uses the average over the metrics of the different seeds while the minimum improvement uses the minimum. The improvement of an algorithm is expressed in percentages. If the algorithm has a better *cut* or *connectivity*, then the baseline, than the improvement is negative. If the baseline provides better results than the improvement is positive. The average time of an algorithm is the geometric mean over the different execution times of all experimental instances. The average running time is expressed in seconds.
To compare the different algorithms we use the Friedman test, a statistic significance test [9]. The null-hypothesis states that all algorithms are equivalent. The test returns a p-value for each algorithm pair. If the p-value between two algorithms is high, it is likely that their results are statistically equivalent. A small p-value below $\alpha = 0.01$ states that the results of two algorithms are significantly different.

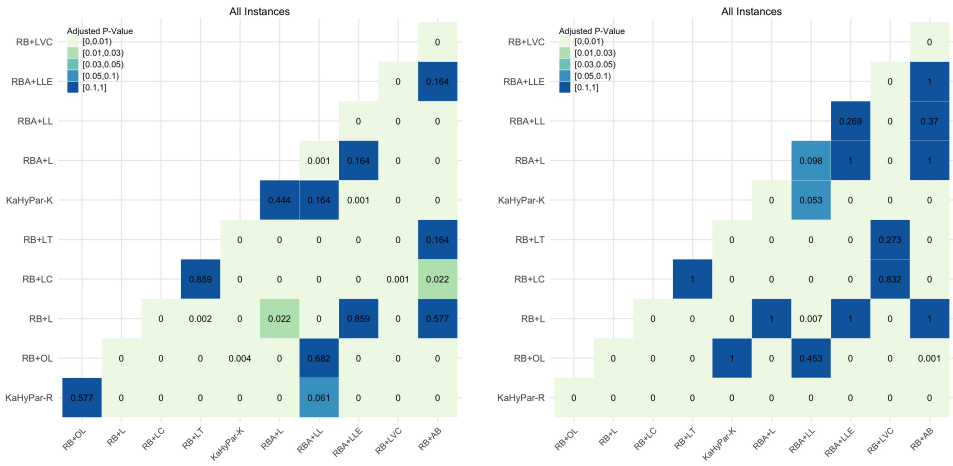


Figure 5.1: This figure contains the results of the Friedman test for most of the algorithms listed in Table 4.1. The test was computed on the results of the small hypergraph subset A.1 for *cut* (left) and *connectivity* (right). The two axes contain the different algorithms and the values in the matrix represent the p-value.

5.3 Experimental Results

5.3.1 Overview of The Different Algorithms

To get an overview of the performance of the algorithms we used the Friedman test with Rom correction. The tests are performed on the small hypergraph set listed in Table A.1. Figure 5.1 presents the results of the Friedman test in matrix form. The average improvement, minimum improvement and average running time of these experiments are summed up in Table 5.1 for *cut* and *connectivity*.
Note that for *cut* (Figure 5.1 left) recursive bisection with one k-way local search at the final partition (RB+OL) does not improve KaHyPar-R. According to Table 5.1 RB+OL only has an improvement of 0.18%. On the other hand, Figure 5.1 (right) shows that RB+OL and KaHyPar-R are statistically different for *connectivity*. RB+OL improves *connectivity* by 0.99%. The average running time for *cut* and *connectivity* almost has not increased, so the metric can be improved with little effort. The insignificant additional running time of RB+OL can be explained by the fact that only one k-way local search is applied, which is insignificant to the running time of the recursive bisection.
By investing more additional running time recursive bisection with local search refinement at the nodes of the recursive bisection (RB+L) improves *cut* by 1.42% and *connectivity* by 1.40%. According to the Friedman test, this improvement is significantly better than that of RB+OL. During the refinements at the nodes of the recursive bisection tree, the algorithm performs $O(k)$ k-way local searches. Note that most of the local searches are

Table 5.1: This table contains the average improvement (Avg.), minimum improvement (Min.) and average time (Time) of the different refinement algorithms for *cut* and *connectivity*. The baseline is the original recursive bisection (KaHyPar-K).

Algorithm	Avg. (%)	Min. (%)	Time (s)	Avg. (%)	Min. (%)	Time (s)
		cut			λ^{-1}	
KaHyPar-R	0.00	0.00	53.64	0.00	0.00	67.04
KaHyPar-K	0.90	0.66	50.46	1.42	1.15	30.21
RB+OL	0.16	0.18	54.62	1.07	0.99	68.13
RB+L	1.45	1.42	56.21	1.48	1.40	69.59
RB+LC	1.83	1.79	88.49	1.83	1.76	106.12
RB+LT	1.81	1.77	70.18	1.83	1.75	84.15
RBA+L	0.77	0.88	724.50	1.29	1.24	581.58
RBA+LL	0.01	0.14	57.52	1.13	1.01	70.90
RBA+LLE	1.19	1.26	67.14	1.28	1.18	80.74
RB+LVC	2.62	2.53	186.95	2.46	2.23	175.96
RB+AB	1.24	1.27	125.85	1.12	1.08	115.79

applied on small subpartitions. RB+L provides better results than RB+OL because more local searches are applied. Furthermore, it is possible that the refinements found by the k-way local searches at the subpartitions lead to a better refinement at the larger portions.
By repeating the local search at the nodes of the recursive bisection tree (RB+LC) the metrics could be improved significantly compared to RB+L. RB+LC improves *cut* by 1.79% and *connectivity* by 1.76%, but its average running time is more than 1.5 times longer. The recursive bisection algorithm with repeated local search at the nodes of the recursive bisection, which prefers local search at the lower nodes (RB+LT), produces results which are statistically equivalent to those of RB+LC. However, the average running time of RB+LT is smaller for *cut* and *connectivity*. RB+LC possibly has a better improvement than RB+L, because after refining a partition, it is possible that a k-way local search at subpartitions finds further improvements.
RB+LT applies the k-way local search refinements in the same order as RB+LC. Instead of continuing the local search refinements at the nodes of the recursive bisection tree after a local search has found an improvement, RB+LT restarts the refinement at the bottom nodes. This results in more k-way local search refinements at small partitions. The stop condition for both algorithms is the same: The algorithm cannot improve the objective at the nodes of the recursive bisection tree. So RB+LT computes the refinement at smaller partitions, this may lead to similar results and different running times.
The recursive bisection algorithm with local search after each bisection (RBA+L) has an average running time that is more than 9 times higher than that of KaHyPar-R. RBA+L improves *cut* by 0.88% and *connectivity* by 1.24%. RBA+L performs $O(k)$ local searches on the original hypergraph. The previous algorithms on the other side perform the lo-

cal searches on subpartitions. This could explain the longer running time. Despite the additional running time, it provides worse results than RB+L, RB+LC and RB+LT. One explanation may be that RBA+L improves the objective between subpartitions. Applying a local search on unfinished partitions corresponds to refining a partition, where each block contains more than one block of the final partition. This could make bisection harder for the recursive bisection algorithm.
The results of the recursive bisection algorithm with local search refinement at each depth (RBA+LL) are statistically equivalent to those of RB+OL. The average running time of RBA+LL is higher. Since RB+L provides better results and has a lower average running time than RBA+L and RBA+LL, we do not perform further experiments for those two algorithms.
Of those algorithms performing the local search on the unfinished partitions, RBA+LLE has the best minimum improvements 1.26% for *cut* and 1.18% for *connectivity*. Its average running time is approximately 1.2 times larger than that of KaHyPar-R. It is statistically equivalent to RB+L. RBA+LLE performs the different $O(\log(k))$ k-way local searches on the original hypergraph, not subhypergraphs. This can explain the longer running time. RBA+LLE performs less k-way local search refinements than RBA+L, but finds better results. RBA+LLE skips the first $k - 2 * \log(k)$ refinements of RBA+L. It avoids the k-local search on partitions containing large blocks, which have to be bisected a lot during the algorithm. This could explain the different improvements.
The results of recursive bisection with 2-way local search refinement with active block scheduling are statistically equivalent to those of RB+L and RBA+LLE for *cut* and RB+L, RBA+L, RBA+LLE for *connectivity*. The best improvement provides the recursive bisection algorithm with one V-cycle refinement after the partitioning (RB+LVC). It improves *cut* by 2.53% and *connectivity* by 2.23%. According to the Friedman test, the results of RB+LVC are significantly different to those of the other algorithms for *cut*. For *connectivity* the results of RB+LVC are not statistically different to those of RB+LC. Compared to KaHyPar-R, its average running time is 3.5 times longer for *cut* and 2.6 times for *connec-*

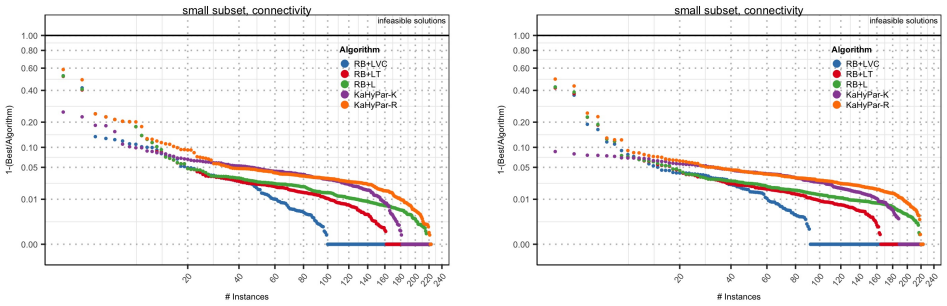


Figure 5.2: This Figure sums up the experimental results of RB+L, RB+LT and RB+LVC. The pot on the left side shows the improvement of *cut* and the plot on the right side the improvement of *connectivity* from the original recursive bisection.

tivity. If we refine the partition with multiple V-Cycles, it may be improved further but the running time will increase significantly. We have tried recursive bisection with a maximum of 30 V-cycles but we had to stop the experiments due to their long running time. So we compute RB+LVC with only one V-cycle on the big subset.
Figure 5.2 sums up the experimental results of RB+L, RB+LT and RB+LVC in a performance plot. RB+LVC provides the best results for more than half of the instances. The second-best algorithm is RB+LT. Even if RB+L does not calculate the best metric it is situated most of the time under direct k-way partitioning (KaHyPar-K). All in all RB+L, RB+LT and RB+LVC provide better results than direct k-way partitioning and improve the metric of the original recursive bisection.
Figure 5.3 shows the experimental results of the direct k-way partitioning algorithm with active block scheduling refinement during uncoarsening (KaHyPar-K+AB). The experiments are performed on the small hypergraph set listed in Table A.1. KaHyPar-K+AB provides worse partitions for *cut* (left) and *connectivity* (right) than the original direct k-way algorithm (KaHyPar-K). According to Table 5.2, *cut* becomes 3.16% worse and *connectivity* becomes 2.36% worse than direct k-way. Compared to KaHyPar-K the average running of KaHyPar-K+AB is more than twenty times slower. Due to the bad average running time and the poor improvements, we do not calculate further experiments for KaHyPar-K+AB.

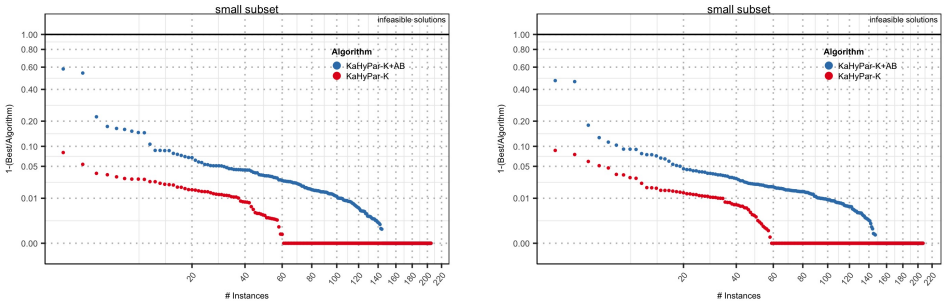


Figure 5.3: Comparing the resulting partitions of the direct k-way algorithm and the direct k-way algorithm with active block scheduling refinement. The pot on the left side shows the improvement of *cut* and the plot on the right side the improvement of *connectivity*.

Table 5.2: Comparing the direct k-way algorithm with direct k-way algorithm with active block refinement while optimizing *cut* and λ^{-1}. The table contains the average improvement (Avg.), minimum improvement (Min), and average running time (Time).

Algorithm	Avg. (%)	Min. (%)	Time (s)	Avg. (%)	Min. (%)	Time (s)
		cut			λ^{-1}	
KaHyPar-K	0.00	0.00	55.42	0.00	0.00	55.42
KaHyPar-K+AB	-3.16	2.79	1345.31	-2.36	1.91	1234.15

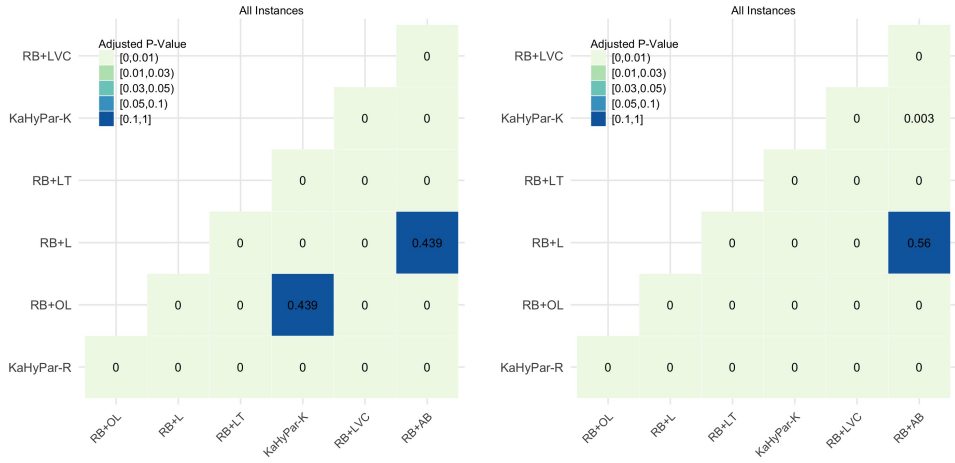


Figure 5.4: This figure contains the results of the Friedman test for most of the algorithms listed in Table 4.1. The test was computed on the results of the large hypergraph set A.2 for *cut* (left) and *connectivity* (right). The two axes contain the different algorithms and the values in the matrix represent the p-value.

5.3.2 Experiments on the Larger Hypergraph Set

The results of the large hypergraph subset, listed in Table A.2, are presented in the following section. Figure 5.4 contains the p-values of the Friedman test with Rom correction. Table 5.3 contains the average improvement, minimum improvement and average time of the experiments on the big benchmark subset. According to the Friedman test, the direct k-way algorithm (KaHyPar-K) provides results for *cut*, which are statistically equivalent to those of RB+OL. Furthermore, they have approximately the same minimum improvement. Compared to KaHyPar-K the average running time of RB+OL is approximately 1.4 times longer.
RB+AB and RB+L are statistically equivalent for *cut* and *connectivity*. They both have similar minimum improvements, but the average running time of RB+AB is significantly larger. We can assume that the other algorithms are not statistically equivalent because their p-value is below $\alpha = 0.01$. The large running time of RB+AB could be explained by the highly localized approach. We only apply 2-way local searches with active block scheduling. An explanation would be that the 2-way local searches find a lot of small improvements. So the active block scheduling algorithm would continue and the over all improvement could be small.
RB+L improves *cut* by 0.982% and *connectivity* by 1.119%. RB+OL improves *cut* by 0.712% and *connectivity* by 0.878%. They need no significant additional running time. This reinforces the interpretation made in the previous section. The running time of RB+L and RB+OL is not significantly different to that of the original recursive bisection.
The recursive bisection algorithm with one V-Cycle refinement (RB+LVC) provides the best improvement for both metrics. However, it is 5.9 times slower for *cut* and 4.46 times

Table 5.3: The table sums up the results for all experiment instances of the large hypergraph set A.2. It contains the average improvement (Avg.), minimum improvement (Min), and average running time (Time).

Algorithm	Avg. (%)	Min. (%)	Time (s)	Avg. (%)	Min. (%)	Time (s)
		cut			$\lambda-1$	
KaHyPar-R	0.000	0.000	166.82	0.000	0.000	234.65
RB+OL	0.732	0.712	168.08	0.897	0.878	236.52
RB+L	1.009	0.982	171.22	1.206	1.199	240.30
RB+LT	1.424	1.382	212.55	1.655	1.638	296.72
RB+AB	0.944	0.948	367.58	1.065	1.053	408.12
RB+LVC	2.243	2.178	984.71	2.624	2.512	1046.25
KaHyPar-K	0.700	0.730	230.43	1.512	1.504	195.49

slower for *connectivity* than KaHyPar-R. For *cut* all recursive bisection algorithms with k-way local search refinement except RB+OL have a better minimum improvement than KaHyPar-K. On the other hand, only RB+LT and RB+LVC provide better results than direct k-way partitioning for *connectivity*. An explanation for this fact may be that for *connectivity* a k-way local search can find better results than a 2-way local search. The k-way local search refinement has a much more global view. Only RB+LT and RB+LVC use enough k-way local searches to improve the results of the recursive bisection. So it is possible that for *connectivity* RB+LT finds the best improvements at the larger subpartitions.

Table 5.4 contains the minimum improvement and Table 5.5 the average running time of calculating an ε-balanced k-way partition, while optimizing *cut* and *connectivity*, for the different partition sizes. All in all the the improvements found by the k-way local search refinements and the average running time get larger with the number of blocks k. Especially the average running time of RB+AB gets larger. This could be explained by the fact that during active block scheduling 2-way local searches are performed on adjacent blocks. By increasing the number of blocks k, the number of 2-way local searches refinements increases. Additionally if the blocks are highly connected , the number of refinements might increase because we refine pairs of blocks, where at least one block is active.

The improvements of RB+LVC get smaller when k gets larger, while RB+LT gets an improvement over 2% for partitions with more than 512 blocks. However, the average running time for RB+LT is approximately two times larger than that of the original recursive bisection. The results for connectivity are similar. RB+LVC provides better results for small k, while RB+LT improves the objective by more than 2% for large partitions for *connectivity*. The size of the recursive bisection tree increases with the number of blocks k. A larger recursive bisection tree leads to possibly more local search refinements. This can explain the increase in running time and improvement with k for RB+LT. For large k a lot of test instances exceed the time limit of one day for RB+LVC.

Table 5.4: This table contains the minimum improvements (%) of the different refinement algorithms for the different partition sizes (k) and objectives (Obj.). The algorithms are compared to the baseline: KaHyPar-R.

Obj.	Algorithm	k=4	k=8	k=16	k=32	k=64	k=128	k=256	k=512	k=1024
	RB+OL	0.318	0.555	0.511	0.682	0.847	0.873	0.933	0.993	0.826
	RB+L	0.318	0.674	0.635	0.880	1.058	1.164	1.324	1.638	1.436
	RB+LT	0.398	0.809	0.859	1.193	1.460	1.612	1.840	2.429	2.326
cut	RB+AB	0.383	0.980	0.953	1.097	1.136	1.035	1.067	1.032	0.890
	RB+LVC	1.759	2.979	2.654	2.656	2.592	2.150	1.931	1.456	0.974
	KaHyPar-K	2.107	2.064	1.179	0.447	-0.111	0.179	-0.245	0.494	-0.021
	RB+OL	0.282	0.435	0.649	0.830	0.968	1.091	1.200	1.320	1.317
	RB+L	0.282	0.525	0.832	1.063	1.231	1.412	1.631	1.987	2.183
	RB+LT	0.333	0.664	1.132	1.438	1.666	1.889	2.170	2.724	3.252
λ^{-1}	RB+AB	0.340	0.756	0.946	1.114	1.207	1.267	1.348	1.342	1.280
	RB+LVC	1.763	2.848	3.020	3.163	3.033	2.734	2.517	1.804	1.397
	KaHyPar-K	2.009	2.454	2.402	2.166	1.587	0.794	0.384	0.712	0.640

Table 5.5: This table contains the average running time (s) of the different refinement algorithms for the different partition sizes (k) and objectives (Obj.).

Obj.	Algorithm	k=4	k=8	k=16	k=32	k=64	k=128	k=256	k=512	k=1024
	KaHyPar-R	116.9	155.6	171.9	163.5	193.5	181.7	168.6	174.5	182.9
	RB+OL	117.0	155.8	172.0	164.0	194.1	183.1	170.7	177.3	187.8
	RB+L	117.0	156.2	172.9	165.5	196.6	186.7	175.7	184.3	197.9
cut	RB+LT	117.4	157.2	175.6	170.5	211.0	223.5	243.9	304.5	368.0
	RB+AB	119.1	161.7	185.4	197.8	276.7	331.6	469.6	715.7	1091.4
	RB+LVC	334.3	599.6	738.5	1155.4	2030.4	1498.8	1021.2	844.3	609.7
	KaHyPar-K	176.7	275.7	108.2	198.9	424.6	311.7	208.4	174.6	179.1
	KaHyPar-R	135.5	193.3	179.8	216.4	267.3	299.7	351.6	224.2	252.6
	RB+OL	135.7	193.7	180.3	217.2	268.5	301.4	354.9	227.7	259.1
	RB+L	135.7	194.0	181.1	218.6	270.9	305.9	361.6	236.2	271.0
λ^{-1}	RB+LT	136.1	195.8	184.0	225.9	287.4	350.1	459.0	372.6	526.0
	RB+AB	137.4	204.5	192.5	248.2	337.4	447.4	709.5	611.9	935.5
	RB+LVC	333.3	478.9	617.2	767.9	1336.0	1486.4	2931.5	781.5	628.8
	KaHyPar-K	78.8	164.3	70.9	99.5	272.0	241.5	483.4	145.4	206.8

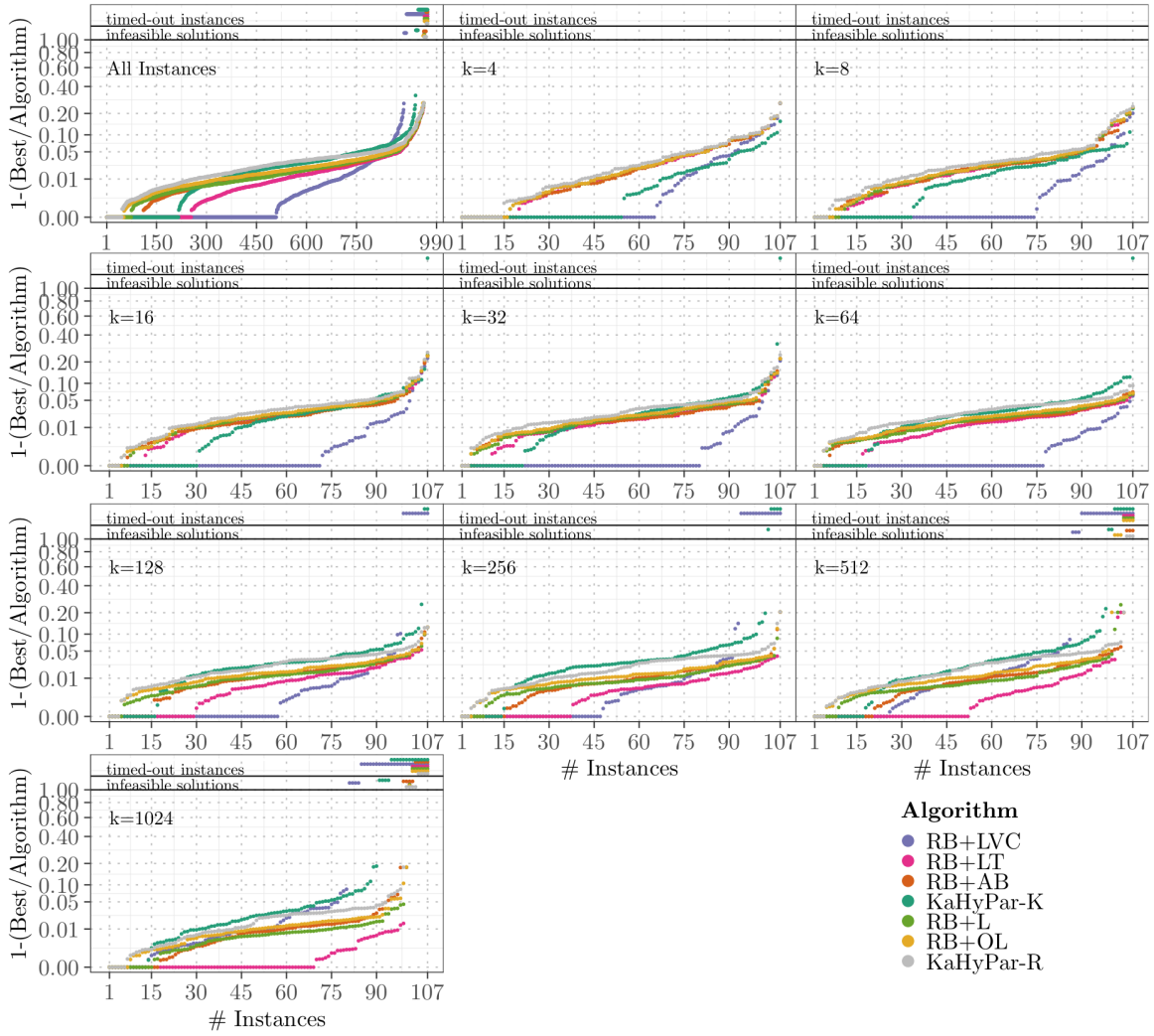


Figure 5.5: This figure contains the improvement plots for *cut*. The top left plot contains all instances, while the other show the results for the different partition sizes. The instances which exceeded the time limit of one day are represented at the top of each plot. Infeasible solutions correspond to imbalanced partitions.

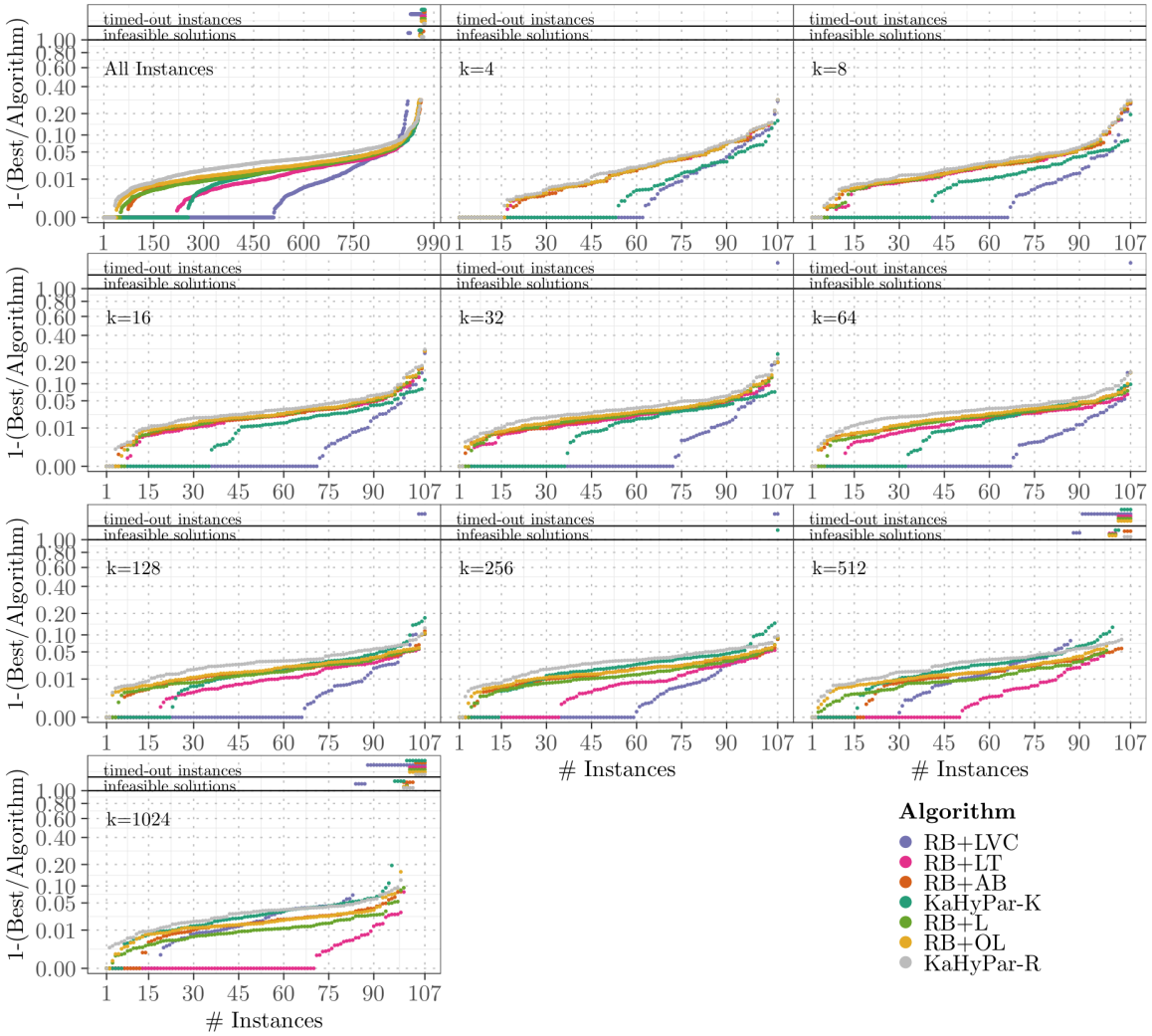


Figure 5.6: This figure contains the improvement plots for *connectivity*. The top left plot contains all instances, while the other show the results for the different partition sizes. The instances which exceeded the time limit of one day are represented at the top of each plot. Infeasible solutions correspond to imbalanced partitions.

Figures 5.5 and 5.6 show the improvement plots for the different partition sizes k. Over all instances (top left) RB+LVC provides the best results for both metrics. RB+LT has the second-best results for *cut* and KaHyPar-K for connectivity. The original recursive bisection computes the worst results, which means that each algorithm could improve the objective. For *cut* and small k, RB+LVC and direct k-way provide the best results. With increasing block number, the improvement of direct k-way gets smaller and of that RB+LT gets larger, until it surpasses that of RB+LVC. For *connectivity* the improvements over all instances are similar to that for *cut*. RB+LVC provides the best improvement for partitions smaller or equal than 256. The direct k-way algorithm remains longer the second-best algorithm, until RB+LT surpasses both for partitions larger than 512. For small k (≤ 32) RB+OL, RB+L, RB+LT provide similar results. This could be explained by the fact that for small k RB+L and RB+LT do not apply as many local search refinements as for large k. The number of k-way local searches applied increases with k and so does the improvement, especially for RB+LT. All in all the results of the improvement plots match to those of the minimum improvement in Table 5.4.
Figures 5.7 and 5.8 contain the improvement plots for the different hypergraph types. Table 5.6 and 5.7 contain the minimum improvement and average running time of the different hypergraph types. The improvements are similar for all hypergraph types, except for the *Dual* hypergraphs where only RB+LVC could improve the objective. *Dual* instances have a large net size. An explanation could be that the k-way local search refinements could not improve the objective. It is hard for k-way local search to remove large nets between different blocks. This can be explained by the fact that the local search we use is move based. This means only one vertex is moved at a time. This can be observed in the average running time of RB+LT, which is similar to that of RB+L. Note that RB+L and RB+LT apply the local search refinements at the same subpartitions. The difference between these algorithms is that RB+LT can repeat the refinements if it finds improvements. The similar running times indicate that the k-way local searches could not find a lot of improvements. RB+LVC uses a V-Cycle, which coarsens the hypergraph. By coarsening a hypergraph the net size becomes smaller, which can be the reason why RB+LVC can improve the objective for *Dual* instances.
RB+LT improved the objective by more than 2% for the *ISPD*, *Pimal* and *Literal* hypergraphs. All these hypergraphs have a small average net size, which might be the reason why the k-way local search refinements can improve the objective. For these hypergraphs RB+L provides significantly worse results than RB+LT. This could be caused by the repeated local search at the nodes of the recursive bisection tree of RB+LT.
For *cut* RB+LVC has an improvement of 3.144% for *ISPD* and 3.155% for *Dual* hypergraphs. Note that the average running time of RB+LVC is significantly higher for the *Dual* hypergraphs. While optimizing *connectivity* metric, RB+LVC has worse results for *ISPD* and better results for *Dual* hypergraphs. RB+LVC provides the best improvement for Dual hypergraphs, but has the longest average running time.

Table 5.6: This table contains the minimum improvements (%) of the different refinement algorithms for the different graph types and objectives (Obj.). The algorithms are compared to the baseline: KaHyPar-R.

Obj.	Algorithm	*	DAC	ISPD	Primal	Literal	Dual	SPM
	RB+OL	0.732	0.654	1.386	1.584	1.451	0.033	0.338
	RB+L	1.009	0.877	1.750	1.917	1.733	0.079	0.632
	RB+LT	1.424	1.141	2.326	2.424	2.334	0.115	1.029
cut	RB+AB	0.944	1.656	1.945	1.367	1.351	0.252	0.632
	RB+LVC	2.243	3.987	3.144	2.302	2.339	3.155	1.736
	KaHyPar-K	0.700	1.562	1.479	2.058	1.219	-1.216	0.389
	RB+OL	0.897	0.321	1.460	1.820	1.483	0.137	0.603
	RB+L	1.206	0.463	1.836	2.107	1.662	0.306	0.979
	RB+LT	1.655	0.743	2.299	2.604	2.254	0.530	1.439
λ^{-1}	RB+AB	1.065	0.983	1.740	1.463	1.370	0.740	0.828
	RB+LVC	2.624	3.488	2.700	2.856	2.764	3.543	2.244
	KaHyPar-K	1.512	2.528	2.256	3.452	2.613	-1.841	1.236

Table 5.7: This table contains the average running time (s) of the different refinement algorithms for the different graph types and objectives (Obj.).

Obj.	Algorithm	*	DAC	ISPD	Primal	Literal	Dual	SPM
	KaHyPar-R	166.8	519.9	98.0	158.4	221.7	320.1	117.9
	RB+OL	168.1	519.1	98.5	160.7	224.8	322.2	118.5
	RB+L	171.2	521.5	100.3	165.6	230.5	327.1	120.5
cut	RB+LT	212.6	532.4	131.6	263.3	364.1	331.0	135.0
	RB+AB	367.6	668.4	278.4	505.8	799.1	460.4	204.7
	RB+LVC	984.7	4844.7	356.2	464.1	756.4	2535.3	773.6
	KaHyPar-K	230.4	1294.2	74.9	102.1	200.2	313.2	227.8
	KaHyPar-R	234.6	894.9	115.7	170.2	229.5	510.1	168.8
	RB+OL	236.5	898.3	116.8	173.5	233.4	512.9	169.6
	RB+L	240.3	900.0	118.2	179.1	239.9	519.9	172.1
λ^{-1}	RB+LT	296.7	955.3	155.7	288.3	402.5	571.9	193.1
	RB+AB	408.1	1353.9	222.2	471.8	708.3	666.6	230.1
	RB+LVC	1046.3	7360.9	271.6	489.7	783.0	2199.8	713.7
	KaHyPar-K	195.5	876.8	48.9	94.9	155.5	241.8	203.1

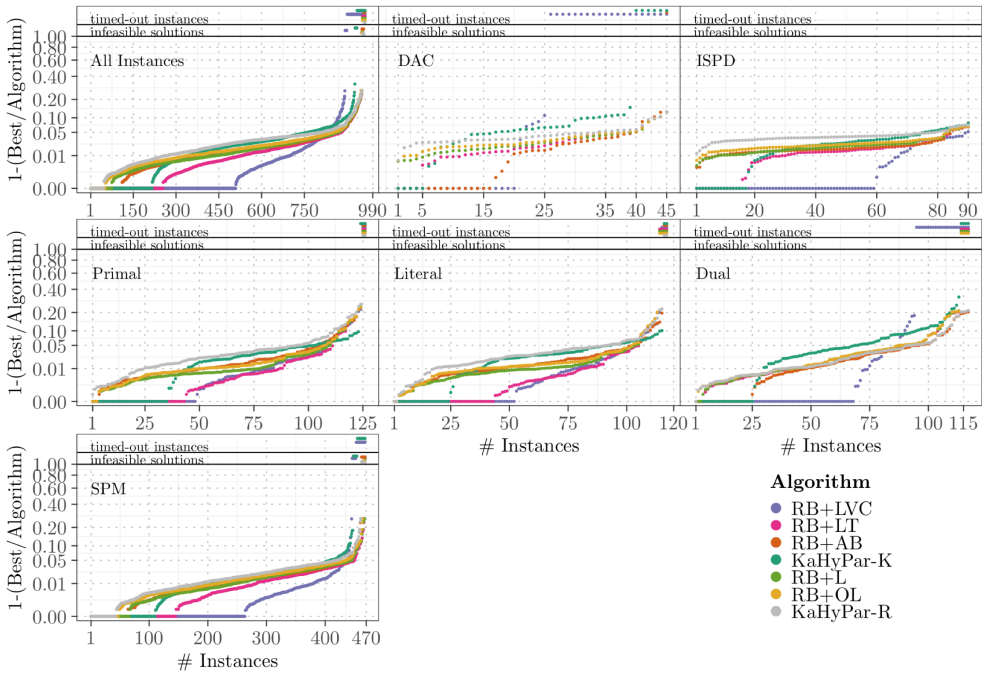


Figure 5.7: These improvement plots show the improvement for the different hypergraph types. The optimized metric is *cut*.

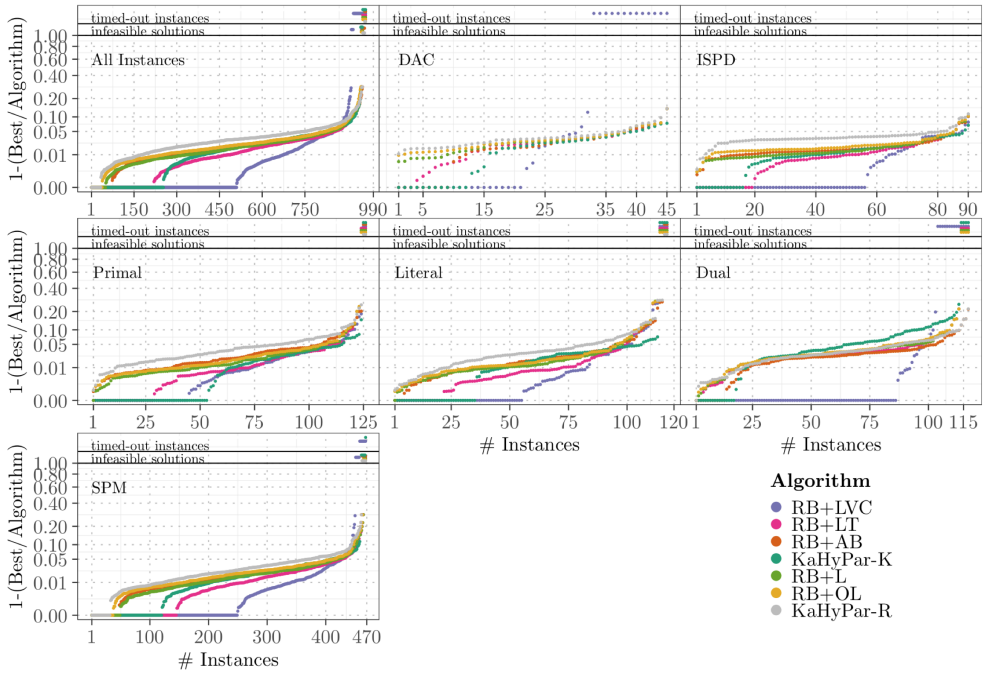


Figure 5.8: These improvement plots show the improvement for the different hypergraph types. The optimized metric is *connectivity*.

5.3.3 Comparison to State-Of-The-Art Partitioner

Finally, we compare our recursive bisection with V-Cycle refinement (RB+LVC) and recursive bisection with repeated k-way local search at the nodes of the recursive bisection tree (RB+LT) with state-of-the-art hypergraph partitioner. We compare with four KaHyPar [11, 22, 23] configurations (KaHyPar-R, KaHyPar-CA, KaHyPar-MF, KaHyPar-R-MF), two hMetis [14] configurations (hMetis-R, hMetis-K) and two PaToH [24] configurations (PaToH-Q, PaToH-D). For the tuning parameters and description of the different algorithms, we refer to the corresponding publications [6, 12, 13, 15, 16, 23].
Figure 5.9 contains the improvement plots which compare RB+LVC and RB+LT to the state-of-the-art partitioner for cut and connectivity. For cut (top) RB+LVC provides a better objective than the other algorithms except KaHyPar-MF, while RB+LT performs results which are similar to those of KaHyPar-R-MF. For *connectivity* (bottom) RB+LVC performs the second-best results after KaHyPar-MF. Note that the performance difference between RB+LVC and KaHyPar-MF is bigger for connectivity than for *cut*. RB+LT produces similar results than KaHyPar-CA.
Table 5.8 contains the average running time for the different algorithms. Note that KaHyPar-MF computes better partitions in a shorter time than RB+LVC for *cut* and *connectivity*. On the other side, RB+LT computes similar results than KaHyPar-R-MF and has a similar running time for *cut*. Finally, the results of KaHyPar-R-CA are similar to those of RB+LT, but it is approximately 1.5 times faster than RB+LT.

Table 5.8: The table compares the average running time (s) of the different algorithms.

metric/algo	KaHyPar-R	RB+LT	RB+LVC	KaHyPar-CA	KaHyPar-R-MF	KaHyPar-MF	hMetis-R	hMetis-K	PaToH-Q	PaToHs-D
cut	165.84	177.01	1138.45	-	180.64	436.71	319.34	178.60	15.75	3.88
λ^{-1}	221.44	235.41	844.36	164.98	-	221.60	339.11	171.68	17.41	4.22

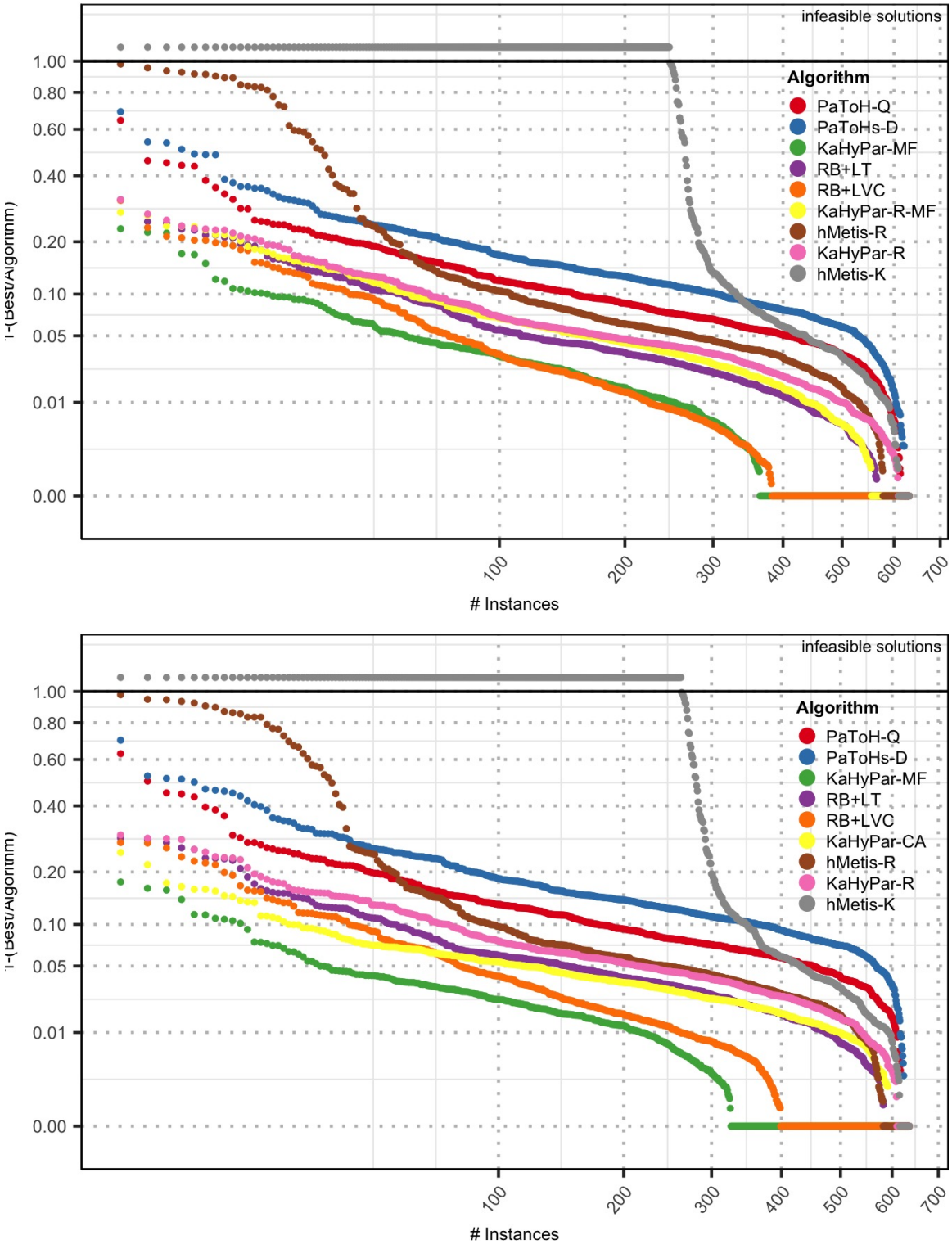


Figure 5.9: The plot on the top is the improvement plot for the *cut* metric, while the bottom plot is the improvement plot for *connectivity*. The tests were performed on the hypergraph set A.2 for partition sizes $k \in \{4, 8, 16, 32, 64, 128\}$.

6 Discussion

6.1 Conclusion

In this thesis we combine the recursive bisection algorithm with k-way local search. The main idea is to improve the objective during and after the recursive bisection by applying k-way local search refinements.
Our first algorithm applies one k-way local search after the recursive bisection. This algorithms improved slightly the objective with slightly no additional running time. Furthermore, we engineered three algorithms which applied the k-way local search at the nodes of the recursive bisection tree. The first one applied the refinements bottom up. The second one repeated this procedure until no improvement could be found. And the last one prefers local search at the bottom nodes of the recursive bisection tree. These algorithms significantly improve the partition provided by the recursive bisection algorithm. However, the repeated k-way local search refinements are accompanied by significantly lager running times.
Moreover, we designed three algorithms which perform the refinements at unfinished partitions during recursive bisection. One of these algorithms performs a k-way local search after each bisection. This approach results in a much larger running time than the original recursive bisection. Its improvement is worse than that of the previous algorithms. The second algorithm performs a k-way local search refinement at each level of the recursive bisection tree. This algorithm has a similar running time than recursive bisection but has nearly no improvement. The third algorithm applies $O(\log(k))$ local search refinements at the end of the recursive bisection. It provides significantly better results than the previous algorithms.
We describe an algorithm, which refines the partition after recursive bisection with multiple 2-way local searches. We use active block scheduling to refine blocks pair wise with 2-way local search. This approach has a similar improvement as recursive bisection with local search refinements at the nodes of the recursive bisection tree. However, it has a significantly higher running time.
One of our algorithms modifies a direct k-way algorithm. We refine the partition during uncoarsening with 2-way local searches with active block scheduling instead of k-way local search. This algorithm has a much higher running time and provides much worse partitions.
Finally, we suggest an algorithm, which refines the partition provided by recursive bisection with V-Cycles. The running time of applying multiple V-Cycles is much too high, so

we perform only one. This is still one of our slowest algorithms, but it provides the best results.
Our most promising algorithms are RB+L, RB+LT and RB+LVC. RB+L can refine the partition provided by the recursive bisection algorithm with little extra running time and scales well with k. RB+LT on the other side needs more extra running time but has better improvements. The improvement of both algorithms gets better with large k. Finally, RB+LVC needs more extra running time but provides the best results for small k.

6.2 Future Work

We performed the test on two hypergraph sets containing 25 and 107 hypergraphs. To get more reliable results and statements the tests should be performed on more hypergraphs. Furthermore, we want to test the algorithms for even bigger partitions, to see how they scale. Moreover, we should test the remaining algorithms (RBA+L, RBA+LL and RBA+LLA) on a larger hypergraph set. Additionally we should compare our refinement algorithms to the state-of-the-art algorithms for large k. We could also work on combining the different refinement approaches into one algorithm. Finally, we could combine KaHyPar-FM and the different refinement algorithms.

A Hypergraph Sets

Table A.1: Small hypergraph subset to quickly test algorithms

hypergraph name
ISPD98_ibm06.hgr
ISPD98_ibm07.hgr
ISPD98_ibm08.hgr
ISPD98_ibm09.hgr
ISPD98_ibm10.hgr
laminar_duct3D.mtx.hgr
mixtank_new.mtx.hgr
mult_dcop_01.mtx.hgr
RFdevice.mtx.hgr
sat14_6s9.cnf.dual.hgr
sat14_6s133.cnf.dual.hgr
sat14_6s133.cnf.hgr
sat14_6s153.cnf.dual.hgr
sat14_6s153.cnf.hgr
sat14_6s153.cnf.primal.hgr
sat14_aaai10-planning-ipc5-pathways-17-step21.cnf.hgr
sat14_aaai10-planning-ipc5-pathways-17-step21.cnf.primal.hgr
sat14_atco_enc2_opt1_05_21.cnf.hgr
sat14_atco_enc2_opt1_05_21.cnf.primal.hgr
sat14_dated-10-11-u.cnf.dual.hgr
sat14_dated-10-11-u.cnf.hgr
sat14_dated-10-11-u.cnf.primal.hgr
sat14_dated-10-17-u.cnf.dual.hgr
sat14_hwmcc10-timeframe-expansion-k45-pdtvisns3p02-tseitin.cnf.primal.hgr
vibrobox.mtx.hgr

Table A.2: Large hypergraph subset to test algorithms

hypergraph name
2cubes_sphere.mtx.hgr
2D_54019_highK.mtx.hgr
af_4_k101.mtx.hgr
af_shell1.mtx.hgr
Andrews.mtx.hgr
as-caida.mtx.hgr
av41092.mtx.hgr
BenElechi1.mtx.hgr
c-61.mtx.hgr
case39.mtx.hgr
cfd1.mtx.hgr
ckt11752_dc_1.mtx.hgr
cnr-2000.mtx.hgr
coupled.mtx.hgr
dac2012_superblue14.hgr
dac2012_superblue16.hgr
dac2012_superblue19.hgr
dac2012_superblue3.hgr
dac2012_superblue9.hgr
denormal.mtx.hgr
dielFilterV2clx.mtx.hgr
EternityII_A.mtx.hgr
ex19.mtx.hgr
gearbox.mtx.hgr
hvdc1.mtx.hgr
Ill_Stokes.mtx.hgr
ISPD98_ibm09.hgr
ISPD98_ibm10.hgr
ISPD98_ibm11.hgr
ISPD98_ibm12.hgr
ISPD98_ibm13.hgr
ISPD98_ibm14.hgr
ISPD98_ibm15.hgr
ISPD98_ibm16.hgr
ISPD98_ibm17.hgr
ISPD98_ibm18.hgr
laminar_duct3D.mtx.hgr
lhr14.mtx.hgr
light_in_tissue.mtx.hgr

Lin.mtx.hgr
lp_pds_20.mtx.hgr
m14b.mtx.hgr
mc2depi.mtx.hgr
mixtank_new.mtx.hgr
mult_dcop_01.mtx.hgr
NotreDame_actors.mtx.hgr
opt1.mtx.hgr
para-4.mtx.hgr
pdb1HYS.mtx.hgr
Pd_rhs.mtx.hgr
pkustk11.mtx.hgr
poisson3Db.mtx.hgr
powersim.mtx.hgr
Pres_Poisson.mtx.hgr
psse2.mtx.hgr
sat14_dated-10-11-u.cnf.primal.hgr
sat14_dated-10-17-u.cnf.dual.hgr
sat14_dated-10-17-u.cnf.hgr
sat14_dated-10-17-u.cnf.primal.hgr
sat14_hwmcc10-timeframe-expansion-k45-pdtvisns3p02-tseitin.cnf.dual.hgr
sat14_hwmcc10-timeframe-expansion-k45-pdtvisns3p02-tseitin.cnf.hgr
sat14_hwmcc10-timeframe-expansion-k45-pdtvisns3p02-tseitin.cnf.primal.hgr
sat14_itox_vc1130.cnf.dual.hgr
sat14_itox_vc1130.cnf.hgr
sat14_itox_vc1130.cnf.primal.hgr
sat14_manol-pipe-c8nidw.cnf.dual.hgr
sat14_manol-pipe-c8nidw.cnf.hgr
sat14_manol-pipe-c8nidw.cnf.primal.hgr
sat14_manol-pipe-g10bid_i.cnf.dual.hgr
sat14_manol-pipe-g10bid_i.cnf.hgr
sat14_manol-pipe-g10bid_i.cnf.primal.hgr
sat14_MD5-28-4.cnf.dual.hgr
sat14_MD5-28-4.cnf.hgr
sat14_MD5-28-4.cnf.primal.hgr
sat14_openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.085-SAT.cnf.dual.hgr
sat14_openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.085-SAT.cnf.hgr
sat14_openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.085-SAT.cnf.primal.hgr
sat14_SAT_dat.k85-24_1_rule_3.cnf.dual.hgr

sat14_SAT_dat.k85-24_1_rule_3.cnf.hgr
sat14_SAT_dat.k85-24_1_rule_3.cnf.primal.hgr
sat14_SAT_dat.k90.debugged.cnf.dual.hgr
sat14_SAT_dat.k90.debugged.cnf.hgr
sat14_SAT_dat.k90.debugged.cnf.primal.hgr
sat14_slp-synthesis-aes-top29.cnf.dual.hgr
sat14_slp-synthesis-aes-top29.cnf.hgr
sat14_slp-synthesis-aes-top29.cnf.primal.hgr
sat14_UCG-15-10p1.cnf.dual.hgr
sat14_UCG-15-10p1.cnf.hgr
sat14_UCG-15-10p1.cnf.primal.hgr
sat14_UR-15-10p1.cnf.dual.hgr
sat14_UR-15-10p1.cnf.hgr
sat14_UR-15-10p1.cnf.primal.hgr
sat14_UR-20-5p0.cnf.dual.hgr
sat14_UR-20-5p0.cnf.hgr
sat14_UR-20-5p0.cnf.primal.hgr
shock-9.mtx.hgr
shyy161.mtx.hgr
skirt.mtx.hgr
sme3Db.mtx.hgr
spmsrtls.mtx.hgr
Stanford.mtx.hgr
stokes128.mtx.hgr
TF16.mtx.hgr
thermomech_TC.mtx.hgr
torso3.mtx.hgr
vibrobox.mtx.hgr
water_tank.mtx.hgr
waveguide3D.mtx.hgr

Bibliography

[1] bwunicluster. https://www.urz.uni-heidelberg.de/en/bwunicluster. Accessed: 23.09.2018.

[2] Bwunicluster hardware and architecture. https://www.bwhpc-c5.de/wiki/index.php/BwUniCluster_Hardware_and_Architecture#Architecture_of_bwUniCluster. Accessed: 23.09.2018.

[3] Detailed experimental results for sea'17 publication "improving coarsening schemes for hypergraph partitioning by exploiting community structure". https://algo2.iti.kit.edu/schlag/sea2017/. Accessed: 23.09.2018.

[4] The sat competition 2014. http://www.satcompetition.org/2014/. Accessed: 23.09.2018.

[5] 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.

[6] T.N. Bui and C. Jones. A heuristic for reducing fill-in in sparse matrix factorization. 12 1993.

[7] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. *Recent Advances in Graph Partitioning*, pages 117–158. Springer International Publishing, Cham, 2016.

[8] 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.

[9] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *J. Mach. Learn. Res.*, 7:1–30, December 2006.

[10] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th Design Automation Conference*, pages 175–181, June 1982.

[11] Vitali Henne, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, and Christian Schulz. n-level hypergraph partitioning, 2015.

[12] Tobias Heuer, Peter Sanders, and Sebastian Schlag. Network Flow-Based Refinement for Multilevel Hypergraph Partitioning. In Gianlorenzo D'Angelo, editor, *17th International Symposium on Experimental Algorithms (SEA 2018)*, volume 103 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:19, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[13] Tobias Heuer and Sebastian Schlag. Partitioning by Exploiting Community Structure. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors,

16th International Symposium on Experimental Algorithms (SEA 2017), volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:19, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

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

[16] 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.

[17] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, Feb 1970.

[18] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Vieweg and Teubner Verlag, Stuttgart, 1992.

[19] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3):153 – 159, 1992.

[20] Alex Pothen. *Graph Partitioning Algorithms with Applications to Scientific Computing*, pages 323–368. Springer Netherlands, Dordrecht, 1997.

[21] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms – ESA 2011*, pages 469–480, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[22] Sebastian Schlag, Yaroslav Akhremtsev, Tobias Heuer, and Peter Sanders. Engineering a direct k-way hypergraph partitioning algorithm, 01 2017.

[23] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way hypergraph partitioning via n-level recursive bisection, 2015.

[24] Aleksandar Trifunovi? and William J. Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563 – 581, 2008.

[25] Chris Walshaw. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research*, 131(1):325–372, Oct 2004.