Karlsruher Institut für Technologie

Bachelor thesis

Algorithm Configuration for Hypergraph Partitioning

Clemens Öhl

Date: May 31, 2018

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

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

Abstract

Hypergraph partitioners such as KaHyPar-CA have many different parameters that are difficult to configure. To the best of our knowledge, there is no work on algorithm configuration of hypergraph partitioners. In this thesis, we configure KaHyPar-CA for different optimization objectives (running time, quality and Pareto) using the optimizer SMAC. We select training instances such that improvements on training sets lead to general improvements. We benchmark our configurations on a newly generated benchmark set. We consider the effects of randomization of KaHyPar-CA to ensure that improvements of configurations are not obscured. Furthermore we analyze the impact of SMAC's parameters on the required optimization time. Despite the difficulties of optimizing for heterogeneous instance sets, we achieve a significant quality improvement of 0.88% on average or an average speedup of 1.66, depending on the optimization objective. Our quality and Pareto optimized configurations are faster than the state-of-the-art hypergraph partitioner hMetis *and* produce partitions with better quality.

Zusammenfassung

Hypergraphenpartitionierer wie KaHyPar-CA haben viele verschiedene Parameter und sind schwer zu konfigurieren. Unserem Wissen nach gibt es bisher noch keine Forschung zur Konfiguration von Hypergraphenpartitionierern. Das Ziel dieser Arbeit ist es KaHyPar-CA mittels dem Optimierer SMAC für verschiedene Optimierungsziele (Laufzeit, Qualität und Pareto) zu konfigurieren. Wir wählen die Menge unserer Trainingsinstanzen so aus, dass Verbesserungen auf dieser Menge zu allgemeinen Verbesserungen führen. Konfigurationen werden auf unserer neu generierten Menge von Benchmarkinstanzen evaluiert. Wir berücksichtigen die Effekte der Randomisierung von KaHyPar-CA um sicherzustellen, dass Verbesserungen von Konfigurationen nicht überdeckt werden. Außerdem analysieren wir den Einfluss der Parameter von SMAC auf die Optimierungszeit. Trotz der Schwierigkeiten bei der Optimierung für heterogene Mengen von Instanzen erzielen wir auf unseren Benchmarkinstanzen signifikante Qualitätsverbesserungen von 0,88% im Durchschnitt oder eine Laufzeitverbesserung von einem Faktor 1,66, je nach Optimierungsziel. Des Weiteren erzeugen unsere auf Qualität und Pareto optimierten Konfigurationen schneller Partitionen als der state-of-the-art Partitionierer hMetis *und* diese weisen eine bessere Qualität auf.

Danksagung

Ich möchte mich bei allen bedanken, die mich bei der Erstellung dieser Bachelorarbeit unterstützt haben. Das sind zum einen natürlich meine Betreuer, welche mir diese Arbeit erst ermöglicht haben. Sie waren bei allen Problemen stets für mich da. Außerdem möchte ich mich auch bei Freunden und Familie bedanken, die mich immer mental unterstützt haben. Außerdem möchte ich mich besonders beim Steinbuch Centre for Computing (SCC) für die bereitgestellte Rechenzeit bedanken sowie bei Marius Lindauer für seine Unterstützung bei der Verwendung von SMAC.

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.

Ort, den Datum

Contents

1 Introduction

1.1 Motivation

Hypergraphs are a generalization of graphs where each (hyper)edge can connect more than two vertices. The generalization of the graph partitioning problem is the k-way hypergraph partitioning problem: partition the vertex set into k disjoint non-empty blocks with a maximum block size of at most $1 + \epsilon$ times the average block size, while minimizing a metric (e.g. the number of cut edges between the blocks).

Use cases for hypergraph partitioning are for example the acceleration of sparse matrix-vector multiplications [38], the identification of groups of connected variables in SAT solving as a preprocessing step [28, 10] or the physical design of digital circuits for very large-scale integration (VLSI) where it is used to partition circuits into smaller units [1].

Since hypergraph partitioning is NP-hard [26], hypergraph partitioning is very complex and heuristics are used in practice. Often these heuristics work only on some types of instances, so the necessary design decisions behind them are also very difficult. Therefore parameters are introduced to allow algorithm designers and end users to choose a good configuration for their instance sets. However, the problem of finding a good configuration, which is known as the *algorithm configuration problem*, is often tried to be solved manually. On the other hand, automated tools for the algorithm configuration problem have been created recently [18, 20]. They promise to outperform default configurations found manually from algorithm designers even on very large parameter spaces.

Motivated by the success of the optimizer SMAC [18] for optimization of a wide range of algorithms [31, 8, 27], we want to optimize KaHyPar-CA [17], a state-of-the-art hypergraph partitioner. However, with respect to the hypergraph partitioning problem, complex optimizers such as SMAC are not directly usable as a black box: SMAC itself has a various parameters which have to be configured, such as the training set, parameters to handle randomization and the definitions of cost functions. The main problem is to ensure that improvements on training sets lead to general improvements.

1.2 Contribution

We present our approach to optimizing KaHyPar-CA [17] using SMAC [18] in such a way that improvements on the training set lead in general improvements. The approach consists

of the definition of different cost functions for different optimization objectives (quality, running time and Pareto), the selection of appropriate training sets and the selection of a representative benchmark set. We consider the effects of randomization of KaHyPar-CA to ensure that the improvements of configurations are not obscured by randomization. We benchmark the found configurations on our newly generated benchmark set. Despite the difficulties of optimizing for heterogeneous instance sets, we achieve significant improvements of 0.88% on average and average speedups of a factor of 1.66, depending on the optimization objective. Furthermore our quality and Pareto optimized configurations produce significantly better partitions than hMetis [23, 24] *and* is faster.

1.3 Structure of Thesis

First we introduce necessary notations and definitions in Chapter 2. We describe how the hypergraph partitioner KaHyPar-CA [17] works and give an overview over algorithm configuration. In Chapter 3 we present our approach to optimizing KaHyPar-CA [17] with SMAC [18]. Our optimization results are presented and discussed in Chapter 4 and followed by a conclusion of this thesis in Chapter 5.

2 Preliminaries

In the following, important definitions are established in Section 2.1, including the hypergraph partitioning problem as well as the algorithm configuration problem. The hypergraph partitioner KaHyPar-CA [17] is described with special attention to its parameters in Section 2.2. We use SMAC [18] to optimize KaHyPar-CA. A description of SMAC can be found in Section 2.3.

2.1 General Definitions

2.1.1 Hypergraph

An *undirected hypergraph* $H = (V, E, c, \omega)$ is defined by a set of n vertices V, a set of m hyperedges (also called *nets*) E and weight functions c and ω. Each net $e \in E$ is a subset of the vertex set V ($e \subseteq V$). The weight functions c and ω assign a positive weight to each vertex ($c : V \to \mathbb{R}_{>0}$) and to each net ($\omega : E \to \mathbb{R}_{>0}$), respectively. A *pin* is a vertex of a net. For a net e, its size $|e|$ is the number of its pins. The weight of a set of vertices $U \subseteq V$ is $c(U) \coloneqq \sum_{v \in U} c(v)$, the weight of a set of nets $F \subseteq E$ is $\omega(F) \coloneqq \sum_{e \in F} \omega(e)$. A vertex v is *incident* to a net e iff $v \in e$. For a vertex v, $I(v)$ is the set of all incident nets of v. The degree $d(v)$ of a vertex v is defined by $I(v)$: $d(v) \coloneqq |I(v)|$. Two vertices are *adjacent* iff a net e exists which contains both vertices. The neighborhood of a vertex v is defined by $\Gamma(v) \coloneqq \{u \mid \exists e \in E : \{v, u\} \subseteq e\}$.

2.1.2 Hypergraph Partitioning Problem

A *k-way partition* of a hypergraph H is a partition of the vertices into k disjoint non-empty blocks $\mathcal{P} = \{V_1, \ldots, V_k\}$ with $V = \bigcup_{i=1}^{k} V_i$. A partition is *$\varepsilon$-balanced* iff for each block $V_i \in \mathcal{P}$ the *balance constraint* is satisfied: $c(V_i) \leq L_{max} \coloneqq (1 + \epsilon)\lceil \frac{c(V)}{k} \rceil$. The number of pins of a net e in a block $V_i \in \mathcal{P}$ is defined as $\Phi(e, V_i) \coloneqq |\{v \in V_i \mid v \in e\}|$. A net e is *connected* to a block V_i iff $\Phi(e, V_i) > 0$. The *connectivity set* of a net e for a k-way partition $\mathcal{P}$ is defined by $\Lambda(e, \mathcal{P}) \coloneqq \{V_i \in \mathcal{P} \mid \Phi(e, V_i) > 0\}$. The *connectivity* of a net e is defined by $\lambda(e, \mathcal{P}) \coloneqq |\Lambda(e, \mathcal{P})|$. For a partition $\mathcal{P}$, a net e is called *cut* if $\lambda(e, \mathcal{P}) > 1$. All cut nets of a partition $\mathcal{P}$ are contained in $E(\mathcal{P}) \coloneqq \{e \in E \mid \lambda(e, \mathcal{P}) > 1\}$.

The k-*way hypergraph partitioning problem* is to find an ε-balanced k-way partition $\mathcal{P}$ that minimizes an objective function. There are several different objective functions, but the following two are the most commonly used [17, 38]:

$$\text{The } \textit{cut-net} \text{ metric: } \operatorname{cut}(\mathcal{P}) \coloneqq \sum_{e \in E(\mathcal{P})} \omega(e)$$

$$\text{The } \textit{connectivity} \text{ metric: } (\lambda - 1)(\mathcal{P}) \coloneqq \sum_{e \in E(\mathcal{P})} (\lambda(e) - 1)\omega(e)$$

We define $\pi \coloneqq (H, k, \varepsilon)$ as an *instance* of the k-way hypergraph partitioning problem with balance constraint ε. In this thesis we consider $k \in \{2, 4, 8, 16, 32, 64, 128\}$ and $\varepsilon = 0.03$. These are common values in literature [17, 23, 38]. We use $(\lambda - 1)$ as quality metric.

2.1.3 Algorithm Configuration Problem

The algorithm configuration problem can be defined as follows: Given a target algorithm A with a parameter configuration space Θ, a (training) instance set Π and a cost metric $\mathfrak{c}$: Find a configuration $\theta \in \Theta$ which minimizes $\mathfrak{c}$ on Π. The cost of a configuration θ is denoted by $\mathfrak{c}(\theta)$ and can be restricted to one single instance $\pi \in \Pi$ by $\mathfrak{c}(\theta, \pi)$. If A is randomized, the cost can also be restricted further to a specific seed s by $\mathfrak{c}(\theta, \pi, s)$. The empirical cost of θ, based on a restricted number of evaluations, is denoted by $\hat{\mathfrak{c}}(\theta)$ and can also be restricted to a single instance π by $\hat{\mathfrak{c}}(\theta, \pi)$.

A synonym for cost metric is *objective*.

Let θ be a configuration for a hypergraph partitioner, π an instance for the hypergraph partitioning problem and s a seed. For an evaluation of θ on π with s, the consumed time is denoted by $\operatorname{time}(\theta, \pi, s)$ and the connectivity (quality) of the partition found by $(\lambda - 1)(\theta, \pi, s)$. Based on K runs of an instance π with a configuration θ and random seeds s_i, the average running time is denoted by $\operatorname{time}(\theta, \pi)$ and the average connectivity by $(\lambda - 1)(\theta, \pi)$:

$$\operatorname{time}(\theta, \pi) \coloneqq \operatorname{geometric\_mean}(\bigcup_{i=1}^{K} \operatorname{time}(\theta, \pi, s_i))$$

$$(\lambda - 1)(\theta, \pi) \coloneqq \operatorname{geometric\_mean}(\bigcup_{i=1}^{K} (\lambda - 1)(\theta, \pi, s_i))$$

2.2 Hypergraph Partitioning

2.2.1 Multilevel Paradigm

Since hypergraph partitioning is NP-hard [26], heuristics are used to find good partitions in an efficient way. State-of-the-art hypergraph partitioners such as KaHyPar [2, 17, 37], hMetis [23, 24] and PaToH [38] use the *multilevel* paradigm. The goal of the multilevel paradigm is to coarsen the input hypergraph to a smaller, structurally similar, hypergraph. The partition of this coarse hypergraph is used to build a partition of the input hypergraph. Coarsening is done by contracting (merging) nodes recursively in several iterations (called *levels*). That way, a hierarchical structure of smaller hypergraphs is built. After a certain termination criterion is met, an initial partitioning algorithm efficiently computes a high-quality partition of the coarsest hypergraph. The partitioned coarsest hypergraph is then *uncontracted* in reverse order. On each level, a local search algorithm is applied to improve the quality of the partition. The phases of the multilevel paradigm are called *coarsening*, *initial partitioning* and *refinement* phase respectively. A visualization of this process can be found in Figure 2.1.

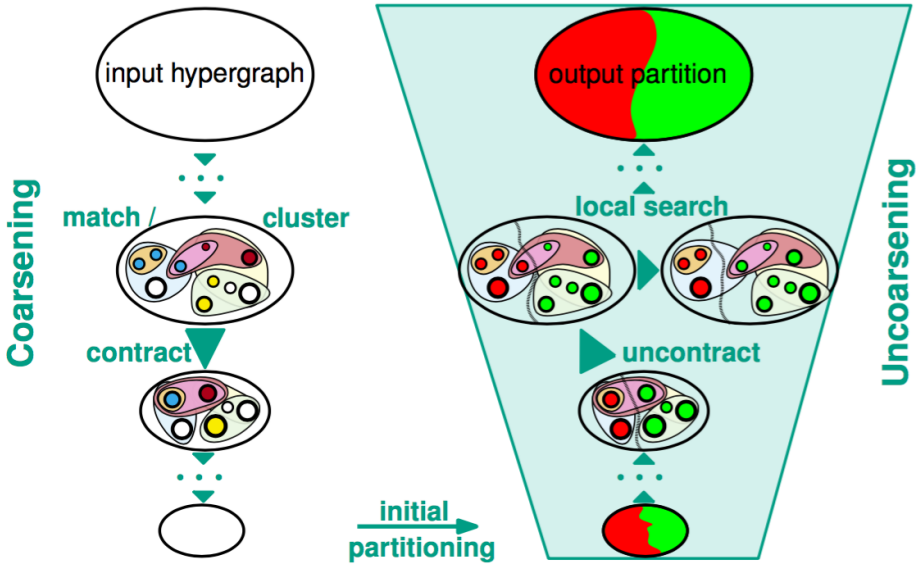


Figure 2.1: Multilevel Paradigm [37, slides]

The coarsening phase is a phase with a high number of different parameters. Coarsening algorithms are designed to fulfill the following goals [24]:

(i) To make move-based local search algorithms more effective during the refinement phase, small nets are needed. Therefore coarsening algorithms should reduce the size of the nets.

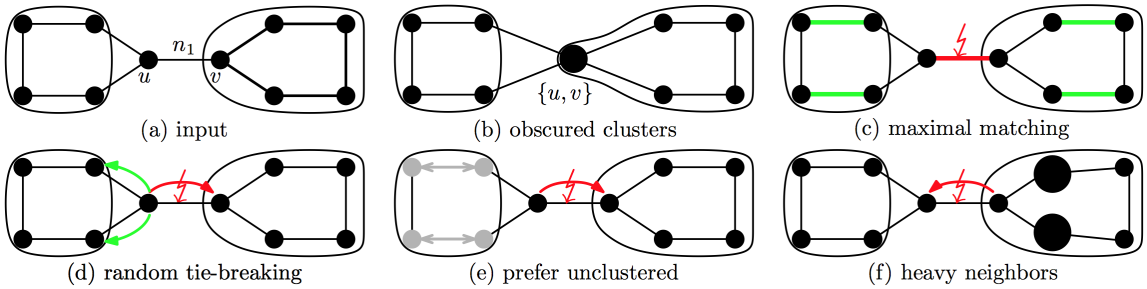


Figure 2.2: Figure from [17]. (a) An input hypergraph with natural cut edge n_1 with cut size 1. (b) The contraction of vertex pair (u, v) obscures this cut edge. (c-f) Coarsening algorithms which can lead to a contraction of (u, v) and thus obscures the structure.

(ii) Since the goal of coarsening is also to produce simpler instances for initial partitioning, the number of nets should be reduced by the coarsening algorithm. This is done by preferring contractions which produce single-vertex nets.

(iii) The coarsest hypergraph should be *structurally similar* to the input hypergraph.

Vertex matchings or clusterings are used by multilevel hypergraph partitioning algorithms for coarsening. To determine which vertices are to be matched or clustered, rating functions are used. In order to fulfill goal one and two, rating functions identify highly connected vertices. One commonly used (e.g. [2, 23, 38, 37]) rating function is *heavy-edge*: For two adjacent vertices u and v, the rating function is defined as $r(u, v) := \sum_{e \in (I(v) \cap I(u))} \frac{\omega(e)}{|e|-1}$. Using maximal matchings to identify contraction vertices, natural existing clusters of vertices can be destroyed [22]. To achieve goal three (structural similarity), instead of strict maximal matching, the formation of vertex clusters is allowed. To avoid imbalanced partitions, a constraint is used to prohibit too heavy vertices.

As depicted by Figure 2.2, the structure of the hypergraph may be obscured: If multiple neighbors for a vertex have the same rating score, a tie-breaking strategy is needed. Possible tie-breaking strategies are for example to choose a random neighbor or to prefer unclustered neighbors. The contraction with the adjacent vertex with the highest score is not allowed if this adjacent vertex is too heavy. Also in this situation, the structure of the graph is obscured. These situations arise because all coarsening decisions are greedily decided, based on *local* information. If a global view of the graph was available, the situations before mentioned could have been avoided.

To partition a hypergraph into more than two blocks, two different approaches are possible: The *recursive* approach and the *direct* approach. If the recursive approach is used, the original hypergraph will be partitioned into two blocks. Then both blocks get partitioned into two blocks themselves. This is done recursively until the original hypergraph is partitioned into k blocks. In contrast, the direct approach partitions the original hypergraph directly into k blocks.

2.2.2 KaHyPar-CA

KaHyPar-CA [17] is a state-of-the-art k-way n-level hypergraph partitioner and is known for its global view on the hypergraph partitioning problem: KaHyPar-CA extracts the structure of the input hypergraph by partitioning its hypernodes into a set of disjoint sets of subgraphs, called *communities*, such that the connections are internally dense but sparser between them [13, 36]. This knowledge is exploited to maintain the structural similarity during coarsening. KaHyPar-CA is thereby able to achieve partitions with a very good quality. Another specialty of KaHyPar-CA is the n-level approach: On each level of coarsening, only one pair of vertices is contracted.

During the coarsening phase, KaHyPar-CA visits vertices in a random order and contracts each vertex immediately with the highest-rated adjacent vertex. Another possible coarsening algorithm consists of saving all ratings in a priority queue and keep it up to date. The rating function used by KaHyPar-CA is the heavy-edge rating function but can be modified to penalty too heavy vertices by multiplying the rating function with $1/(c(u) \cdot c(v))$. The termination criterion is parameterized by the parameter t, coarsening ends if the coarsest hypergraph has at most $k \cdot t$ nodes. Furthermore, the constraint to prohibit imbalanced partitions is parameterized by parameter s: KaHyPar-CA does not allow a vertex v with $c(v) > c_{max} := s \cdot \lceil \frac{c(V)}{k \cdot t} \rceil$ to participate in any further contractions.

As its initial partitioning algorithm, KaHyPar-CA uses KaHyPar-R [37], a recursive n-level hypergraph partitioner. The parameters of KaHyPar-R for coarsening and refinement are the same as for KaHyPar-CA, but some of them have a different default configuration. The initial partitioning itself is done 20 times (configurable) by a pool of different initial partitioning algorithms. The best initial partition is used for refinement.

The goal of local search algorithms during refinement phase is to improve the quality of the partition by moving vertices from one block into another. The k-way local search algorithm used by KaHyPar-CA for the refinement phase is identical with the local search algorithm used by KaHyPar-K [2] and uses similar ideas as the k-way FM-algorithm of Sanchis [35]. Sanchis' algorithm works in phases: In each phase, the vertex with the highest improvement is moved. To get out of local minima, this is even done if the highest improvement is negative. The so far best solution is saved. The algorithm terminates if no further moves are possible. The so far best found solution is then the final solution. If necessary, a rollback operation to the state of this solution is performed.

KaHyPar-CA uses a version of Sanchis' algorithm with several improvements. Note that in the following only one improvement is mentioned, namely the *adaptive stopping rule* of KaHyPar-K. The other improvements (mainly a gain cache, an exclusion of nets to improve running time and the reduction of the number of required priority queues to maintain all possible moves from $k(k-1)$ to k) can be found in the KaHyPar-K paper [2].

The key of the adaptive stopping rule is the analysis whether further improvements are still likely during the current local search. Sanders and Osipov [30] model the gain values in each step as independent identically distributed random variables, based on the previous

p steps. The expectation μ is the average gain since the last improvement and σ^2 is the variance of the improvement of the current local search. Sanders and Osipov showed that it is unlikely that local search can find additional improvements if $p > \sigma^2/(4\mu^2)$.

KaHyPar-K uses a refined version: On each level, at least $\beta := \log n$ steps after an improvement is found are performed. Furthermore, local search continues as long as $\mu > 0$. If after β steps μ is still 0, local search is stopped.

However, instead of the adaptive stopping rule, the *simple stopping rule* can be used: Local search stops if after a constant number of moves no improvements are achieved.

2.3 Algorithm Configuration with SMAC

To the best of our knowledge, there is no work on algorithm configuration for hypergraph partitioning algorithms. Configurations are selected manually based on extensive parameter tuning experiments. In this section, approaches for the algorithm optimization problem are analyzed and some important problems are discussed.

2.3.1 Preliminaries

One very basic but commonly used [7] strategy for algorithm configuration is *grid search*: Let L be the list of all parameters to optimize and let $L_i \in L$ denote a single parameter. A discrete set of values for each L_i is chosen, e.g. by knowledge of the algorithm designer. The configuration space is then the Cartesian product of all parameters. *Every* configuration is evaluated and the best is chosen. The overall number of evaluated configurations is $\Pi_{i=1}^{|L|}|L_i|$. Thus the time needed for grid search grows exponentially with the number of parameters. This effect is known as the *curse of dimensionality* [6]. Although grid search is embarrassingly parallel, its practical use is limited to a few parameters by the curse of dimensionality.

Since grid search is expensive, the set of values chosen for every parameter is relatively small. If the space of good configurations within the configuration space Θ is sparse, then it will be likely that these good configurations are "missed". This effect can be seen on the left side of Figure 2.3.

However, some parameters may have only little impact on the cost function. The *effective dimensionality* of the configuration space is then lower than $|L|$. For instance, cost function $\mathfrak{c}$ with $\mathfrak{c}(x, y) = g(x) + h(y) \approx g(x)$ has only an effective dimensionality of one instead of two. Grid search is unable to exploit this, it relies on the knowledge of the algorithm designer (or somebody else) to distinguish important from unimportant parameters: They have to determine that only parameter x matters and parameter y is only of little importance.

Instead of selecting configurations from a grid, it is possible to select configurations at random. This is called *random search*. Bergstra and Bengio [7] have shown that random

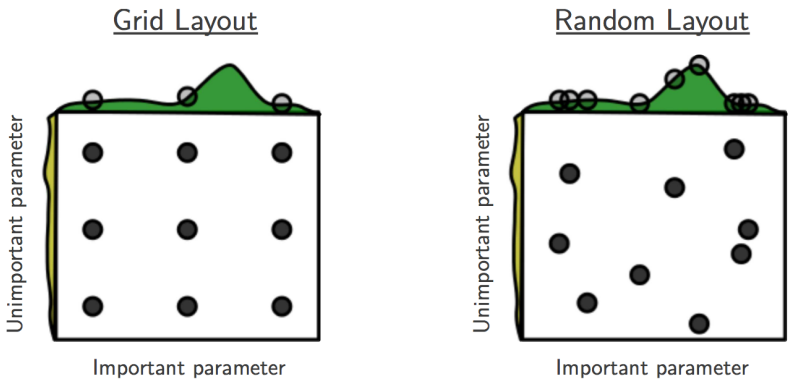


Figure 2.3: Figure from [7]. Optimization of a function $\mathfrak{c}(x, y) = g(x) + h(y)$ with $f(x, y) \approx g(x)$. The green function above each square is $g(x)$, and on the left of each square $h(y)$ is shown in yellow. Each dot represents a configuration evaluated by grid search (left side) or random search (right side).

selection is capable of exploiting low effective dimensionality. An example for this ability can be seen in Figure 2.3: Grid search takes only three different values for each parameter (left side). On the right side, an equal number of configurations (nine) are selected at random. That way, for each parameter nine different values are evaluated.

For this reason, random search is superior to grid search [7]. However, both grid search and random search are very basic approaches and are used only due to their simplicity. However more sophisticated methods for algorithm configuration have been developed. Bergstra and Bengio recommend, among others, to use *Sequential model-based Optimization (SMBO)* methods.

The roots of SMBO are optimization of global continuous ("black box") functions. The goal is to find parameters such that the black box function is minimized. The problem is that each evaluation is costly. Jones et al. [21] solved it by building a model of the black box function. Based on the model, the black box function can be evaluated for promising parameters.

SMBO works similarly: SMBO builds a model M of the cost function $\mathfrak{c}$ based on all runs of the *run history* $R = \{(\theta, \mathfrak{c}(\theta)) \mid \theta \in \Theta \wedge \theta \text{ was already intensified}\}$ of the target algorithm so far. The goal is to predict the cost of a given unintensified configuration θ_{new}. With this knowledge, promising *challenger* configurations are chosen. These challenger configurations are evaluated and compared with the best configuration found so far, the *incumbent* configuration θ_{inc}. If a challenger is better than the incumbent, the challenger becomes the new incumbent. This process is called *intensification*. The runs done during intensification are added to R to improve the model. When the time budget is exhausted, the incumbent configuration is returned as the final configuration. This approach is described by Algorithm 1.

Basic SMBO approaches have several drawbacks: For example, SMBO is unable to cancel

Algorithm 1: Sequential Model-Based Optimization (SMBO) [18]

Data: Target algorithm A with parameter configuration space Θ, instance set Π, cost metric $\mathfrak{c}$
Result: Optimized (incumbent) configuration θ_{inc}

1 $(R, \theta_{inc}) \leftarrow \text{Initialize}(\Theta, \Pi)$
2 repeat
  // t_{fit} and t_{select} is the time spend for FitModel *and* SelectConfiguration *respectively*
3 $(M, t_{fit}) \leftarrow FitModel(R)$
4 $(\overrightarrow{\theta_{new}}, t_{select}) \leftarrow \text{SelectConfigurations}(M, \theta_{inc}, \Theta)$
  // Intensify chooses best configuration from $\overrightarrow{\theta_{new}} \cup \{\theta_{inc}\}$
5 $(R, \theta_{inc}) \leftarrow \text{Intensify}(\overrightarrow{\theta_{new}}, \theta_{inc}, R, t_{itensify} \leftarrow t_{fit} + t_{select}, \Pi, \mathfrak{c}, A)$
6 until *total time budget exhausted*
7 return θ_{inc}

evaluations of the target algorithm that exceed a maximum running time. Furthermore SMBO supports only single instances ($|\Pi| = 1$). This is crucial since configurations will be found which work only on the single training instance but not on others. This effect is known as *over-tuning*.

However, *Sequential Model-based Algorithm Configuration (SMAC)* offers solutions for these problems. SMAC is an instantiation of the SMBO approach and uses *Gaussian Process (GP) models* [32] to build its model. SMAC has recently been used successfully in various application fields, for example: path planning [8], mobile robot localization [27] and real-time railway traffic management [31]. In the following, SMAC's solutions for the support of multiple instances (Section 2.3.2) and for terminating too long runs (Section 2.3.3) are described. Additionally, the procedure of selecting promising configurations (Section 2.3.4) is described. At the end of this section follows some information about parallelization of SMAC (Section 2.3.5), about training sets (Section 2.3.6) and about the possibility to specify instance features to improve the model (Section 2.3.7).

2.3.2 Intensification

As described above, the intensification procedure decides whether the incumbent configuration θ_{inc} or a new challenger configuration θ_{new} is better. As a side effect, the procedure adds runs to the run history R. The procedure implemented by SMAC[1] is shown in Algorithm 2 and described in the following.

The procedure consists of two parts: Lines 3–7 add a run for θ_{inc} to R, lines 10–17 add runs for θ_{new} and decide whether θ_{inc} or θ_{new} is better.

[1]The actual implementation can be found here: `https://github.com/automl/SMAC3/blob/master/smac/intensification/intensification.py`

Algorithm 2: Intensify [18]

Data: Configurations to be evaluated $\overrightarrow{\theta_{new}}$, incumbent configuration θ_{inc}, run history R, time bound $t_{intensify}$, instance set Π, cost metric $\mathfrak{c}$, maximum number of runs for incumbent configuration R_{max}, minimum number of runs of challenger configuration R_{min}, target algorithm A
Result: Incumbent configuration θ_{Inc}, updated R

1 for $i \leftarrow 1$ to $|\overrightarrow{\theta_{new}}|$ do
2 $\theta_{new} \leftarrow \overrightarrow{\theta_{new}}[i]$
3 if R *contains less than* R_{max} *runs with configuration* θ_{inc} then
4 $\Pi' \leftarrow \{\pi' \in \Pi \mid R$ contains less than or equal number of runs using θ_{inc} and π' than using θ_{inc} and any other $\pi'' \in \Pi\}$
5 $\pi \leftarrow$ instance sampled uniformly at random from Π'
6 $s \leftarrow$ seed, drawn uniformly at random
7 $R \leftarrow \text{ExecuteRun}(A, R, \theta_{inc}, \pi, s)$
8 $N \leftarrow R_{min}$
9 while *true* do
10 $S_{missing} \leftarrow (instance, seed)$ pairs for which θ_{inc} was run before, but not θ_{new}
11 $S_{toRun} \leftarrow$ random subset of $S_{missing}$ of size $\min(N, |S_{missing}|)$
12 foreach $(\pi, s) \in S_{toRun}$ do $R \leftarrow \text{ExecuteRun}(A, R, \theta_{new}, \pi, s)$
13 $S_{missing} \leftarrow S_{missing} \setminus S_{toRun}$
14 $\Pi_{common} \leftarrow$ instances for which θ_{inc} and θ_{new} have been ran previously.
  // if the arithmetic mean of the empirical cost of θ_{new} *is higher than of* θ_{inc}
15 if $\mathfrak{c}(\theta_{new}, \Pi_{common}) > \mathfrak{c}(\theta_{inc}, \Pi_{common})$ then break
16 else if $S_{missing} = \emptyset$ then $\theta_{inc} \leftarrow \theta_{new}$; break
17 else $N \leftarrow 2 * N$
18 if *time spent in this call to this procedure exceeds* $t_{intensify}$ *and* $i \geq 2$ then break
19 return (R, θ_{inc})

The run for θ_{inc} will only be performed if R contains at most R_{max} runs for θ_{inc} (line 3). The instance π for this run is not chosen uniformly at random but in such a way that each instance from Π is executed equally often for θ_{inc} (lines 4+5). In lines 10-17, out of all runs for θ_{inc}, a set S_{toRun} with $N = R_{min}$ instance-seed-pairs is chosen at random (line 10+11). The challenger configuration θ_{new} is executed on S_{toRun} (line 12). If the arithmetic mean of the cost of θ_{inc} on S_{toRun} is better or equal than the cost of θ_{new} on S_{toRun}, θ_{new} will be dropped (line 15). Otherwise, θ_{new} will be further intensified with a doubled N and different N instance-seed-pairs from runs of θ_{inc} (line 17). This will be done until the challenger configuration is dropped or no instance-seed-pairs are left. In this case the challenger becomes the new incumbent (line 16).

The intensification procedure will be executed until either all configurations from $\overrightarrow{\theta_{new}}$ are

evaluated or at least two challenger configurations have been evaluated and the time spent on intensification for $\overrightarrow{\theta_{new}}$ is longer than the time spent on generating $\overrightarrow{\theta_{new}}$ (lines 1+18).

For an end user of SMAC, multiple parameters of the intensification procedure are configurable, not only the cost function $\mathfrak{c}$ and the instance set Π:

If the target algorithm is randomized, a potential improvement of a challenger configuration to the incumbent configuration can be obscured by randomization. To handle randomization, it is possible to do more evaluations for each challenger configuration by increasing R_{min}. If the instance set Π is heterogeneous, R_{min} can be increased for the same reason: A challenger configuration may work well on a subset of Π and perform badly on another subset. Thus the selection of the instances for the first R_{min} runs affects the decision whether a challenger is further intensified or not. Note that if Π is homogeneous, this will be rarely the case since a challenger configuration which does not work well on a single instance will also perform badly on another instances.

The number of maximum evaluations per configuration is parameterized by R_{max}. It is important that the final configuration is executed at least once on every instance. Thus $R_{max} \geq |\Pi|$ is required. If the target algorithm is randomized, R_{max} can be set to a high value to achieve more accurate results.

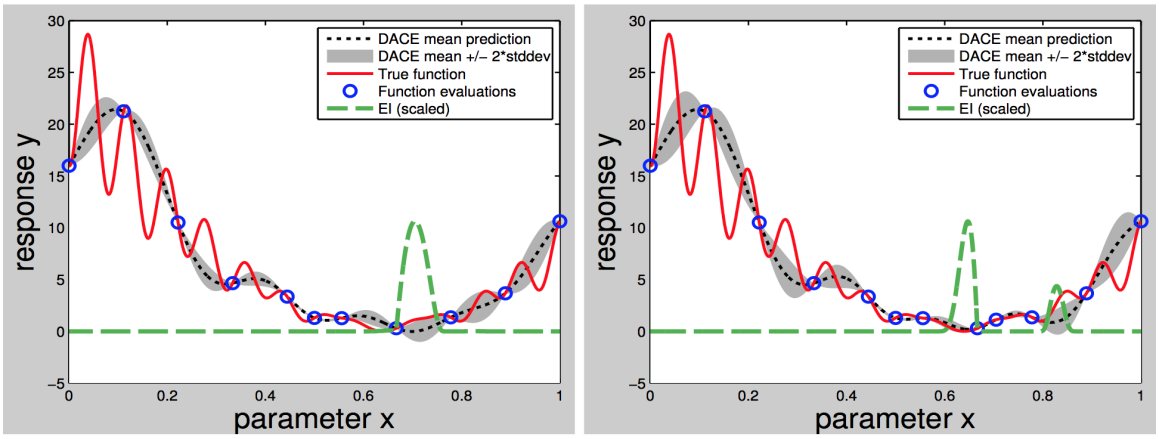
2.3.3 Adaptive Capping

To avoid spending too much time on intensification of time intensive configurations, each call of the target algorithm will be terminated after exceeding a static cutoff time if not finished. If a call of the target algorithm is terminated, the result for this specific run is unknown. A high penalty value is saved for this run in the run history to mark this configuration as "bad" and to avoid selection of other configurations in the neighborhood of θ_i.

As explained in the previous subsection, a challenger configuration must be on average better than the incumbent configuration on the set S_{toRun} used for the current iteration of intensification. If the optimization objective is running time, this can be exploited: If the challenger configuration θ_{new} consumes more time than the incumbent configuration θ_{inc} for a subset of S_{toRun}, the challenger is worse than the incumbent. The intensification of the challenger can be terminated immediately. This mechanism is called *Adaptive Capping* and was introduced in this form by Param-ILS [20] but is also implemented by SMAC.

2.3.4 Selection of Promising Configurations

Since it is not good to intensify configurations from the neighborhood of configurations which are much worse than the incumbent, a criterion is needed to measure the expected improvement of a configuration to the current incumbent configuration. This criterion is called *Expected Improvement (EI)* criterion. SMAC defines the EI-function in such a way



(a) SMBO, step 1 (b) SMBO, step 2

Figure 2.4: Two steps of SMBO for the optimization of a one dimensional noise-free function. The red line is the noise-free (not randomized) true cost function c of the target algorithm with a single parameter x. The blue circles are the results of the evaluation of the target algorithm. The dotted line is the mean of the predicted cost function $\hat{c}$, the gray area denotes its standard deviation. The green dashed line denotes the scaled expected improvement (EI). The figure is taken from Hutter et al. [18].

that EI is high for configurations with expected high variance and an expected low mean and vice versa [18].

An example is illustrated by Figure 2.4: It shows two steps of the optimization process. Some runs (blue circles) were already done. The modeled cost function is denoted by the black line, the expected variance is denoted by the gray area. The red line denotes the true cost function. The EI is, in the first step, only high for the region around $x = 0.7$. There is, according to the model, no other region where a configuration has a chance to be better than the actual best configuration.

SMAC selects configurations which maximize the EI. In case of Figure 2.4a, it is the configuration $x = 0.705$. The configuration is evaluated and as shown in Figure 2.4b, the result is added to the model. The variance of the expected cost function is greatly reduced around the area of the selected configuration by the evaluation. Now there are two different regions with a high EI. They will be evaluated in the next step.

For larger configuration spaces as in Figure 2.4, it becomes costly to consider all possible configurations. In order to handle this problem, it is possible to select a certain number of configurations at random and select those with the highest EI. This will fail if the space of good configurations is sparse. To solve this problem, a multi-start local search algorithm is used by SMAC to find configurations which maximize the EI criterion [18].

2.3.5 Parallelization of SMAC (pSMAC)

As the name suggests, SMAC is a sequential algorithm and is by default unable to execute more than one target algorithm call at once. Since algorithm configuration is costly, a parallelized variant of SMAC is needed. This variant is called *pSMAC* [19]. The solution of pSMAC is to use multiple processes of SMAC with a run history shared via files. Since the run history is shared, all processes work on the same model. Because the selection of promising configurations is randomized, each process intensifies other configurations. However, the incumbent configurations are not shared. Therefore each process may produce a different final configuration.

2.3.6 Homogeneity and Size of Training Sets

The developers of SMAC recommend to use SMAC for homogeneous instance sets [11]. They compare the algorithm configuration problem with the similar *algorithm selection problem*, introduced by Rice [33]. The goal of the algorithm selection problem is to select the best algorithm for a given instance. Often, it is impossible to find an algorithm which performs well on all instances [25]. Because of the similarity of the algorithm selection problem and the configuration selection problem, the selection of a configuration which performs well on all instances is also difficult. Instead of using only one configuration which performs well on all instances, it is possible to use a *portfolio* of configurations (multiple configurations) [42]. An overview of this subject is provided by Kotthoff [25].

The following example helps to understand why heterogeneous instance sets are so difficult to optimize: Let C and D be two subsets of the training set with identical size. Assume that the actual incumbent configuration θ_{inc} has an improvement of $x\%$ on C and $\frac{1}{4}x\%$ on D on average, compared to the default configuration (with $x \gg 0$). Let θ_{new} be a challenger configuration found by SMAC with an improvement of $\frac{1}{2}x\%$ on C and $2x\%$ on D, compared to the default configuration. Thus, on average, θ_{new} is twice as good as θ_{inc}. During intensification, SMAC first runs θ_{new} on R_{min} random instances – suppose they are all from C. Since $\frac{1}{2}x\% < x\%$, θ_{inc} is better than θ_{new} on C. Thus, although θ_{new} is on average twice as good as θ_{inc}, θ_{new} will be dropped by SMAC.

So regarding the first iteration of intensification, the decision whether a challenger configuration is dropped or not depends on the randomly chosen instances. This is avoided by usage of homogeneous training sets. However, it is possible to reduce this effect by increasing R_{min}, because then any θ_{new} will be executed on more instances. Since this is very costly, this should only be done with caution.

To avoid overtuning, the training set must be large enough. The developers of SMAC recommend 50 instances for optimizing homogeneous training sets [11]. In contrast, for heterogeneous instance sets, the developers recommend to use at least 300 instances but more than 1000 instances are advisable [11].

As an extreme example, McPhee et al. [29] used a training set with only one instance because their target algorithm consumed up to 24 hours and not enough computing time was available. The final configuration found by SMAC was better than the default configuration on only one out of four instances. Additionally the performance on one other instance was substantially worse than the default configuration.

2.3.7 Features

To improve the model it is possible to specify a vector of attributes, called *features*, for each training instance. However, this is not required and it is also possible to use some domain independent features like the instance size. Since the features are only needed for the training instances, the effort to calculate the features is usually not too high [18].

Note that SMAC applies a *Principal Component Analysis (PCA)* [14] to reduce the dimensionality of the feature space. Only the first seven components are used by SMAC. This is done to reduce the computational complexity of learning [18].

2.4 Summarizing Benchmark Results

SMAC requires a single number expressing the cost of a configuration θ. By default, SMAC executes a configuration N times and takes the average cost value using the arithmetic mean. In this thesis, we want to compare not absolute cost values (for example, the average connectivity of θ on all training instances) but relative improvements compared to the default configuration. This is because the training instances are of various size with different partitioning times and different minimum cuts. The results of each evaluation of θ are therefore normalized to the default configuration, the cost of θ is then the average improvement of θ compared to the default configuration.

As shown by Flemming and Wallace [12], the arithmetic mean is not an appropriate measure to compare normalized data. Consider the following example: All numbers of Table 2.1 are normalized to R. According to the arithmetic mean, it seems that R is better than M and Z. On the contrary, if the numbers are normalized to M instead of R, like in Table 2.2, the results differ: According to the arithmetic mean, M seems to be better than R and Z. Table 2.2 thus contradicts Table 2.1. This shows the problem of using the arithmetic mean to compare normalized data.

An alternative to the arithmetic mean is the *geometric mean*. The geometric mean of N numbers is defined by the Nth root of the product of the numbers:

$$\text{geometric\_mean}(x_1, \ldots, x_n) = \sqrt[n]{\prod_{i=1}^{n} x_i}$$

Benchmark	Processor		
	R	M	Z
E	417 (1.00)	244 (0.59)	134 (0.32)
F	83 (1.00)	70 (0.84)	70 (0.85)
H	66 (1.00)	153 (2.32)	135 (2.05)
I	39,449 (1.00)	33,527 (0.85)	66,000 (1.67)
K	772 (1.00)	368 (0.48)	369 (0.45)
Arithmetic mean	(1.00)	(1.02)	(1.07)
Geometric mean	(1.00)	(0.86)	(0.84)

Table 2.1: Table taken from Fleming et al. [12]. R, M and Z are in this context configurations and E, F, H, I and K are instances on which these configurations are evaluated. A number before a parenthesis denotes the raw cost of a configuration on an instance. All numbers are normalized to R, the normalized cost value is denoted by the numbers in parentheses. Note that even the absolute values are rounded.

Benchmark	Processor		
	R	M	Z
E	417 (1.71)	244 (1.00)	134 (0.55)
F	83 (1.19)	70 (1.00)	70 (1.00)
H	66 (0.43)	153 (1.00)	135 (0.88)
I	39,449 (1.18)	33,527 (1.00)	66,000 (1.97)
K	772 (2.10)	368 (1.00)	369 (1.00)
Arithmetic mean	(1.32)	(1.00)	(1.08)
Geometric mean	(1.17)	(1.00)	(0.99)

Table 2.2: Table taken from Fleming et al. [12]. Identical to Table 2.1 with one exception: The numbers are normalized to M instead to R.

In contrast to the arithmetic mean, the geometric mean is unaffected by normalization: It is irrelevant to which configuration the raw data is normalized. This property can be seen in Table 2.1 and Table 2.2: The cost of M compared to R is, normalized to R, 0.86. If the data is normalized to M, the cost of R compared to M is 1.17 – this is the reciprocal of 0.86.[2] A similar relationship between the performance of M and Z can also be found. This is because:

$$\text{geometric\_mean}\left(\frac{x_1}{y_1}, \ldots, \frac{x_n}{y_n}\right) = \sqrt[n]{\prod_{i=1}^{n} \frac{x_i}{y_i}} = \frac{1}{\sqrt[n]{\prod_{i=1}^{n} \frac{y_i}{x_i}}} = \frac{1}{\text{geometric\_mean}(\frac{y_1}{x_1}, \ldots, \frac{y_n}{x_n})}$$

Further examples and a proof that the geometric mean is not only correct but is also the only correct average of normalized numbers can be found in [12].

[2] An attentive reader may note that $1/0.86 \approx 1.1628 \not\approx 1.17$ because of rounding errors.

Despite the fact that the arithmetic mean is not appropriate, it can still be used to average the logarithmic of the normalized values. This is due the fact that the geometric mean of n numbers is the exponential of the arithmetic mean of the logarithmic numbers:

$$\text{geometric\_mean}(x_1, \ldots, x_N) = \sqrt[N]{\prod_{i=1}^{n} x_i} = \left(\exp\left(\sum_{i=1}^{n} \log x_i\right)\right)^{1/n} = \exp\left(\frac{1}{n}\sum_{i=1}^{n} \log x_i\right)$$

$$= \exp(\text{arithmetic\_mean}(\log(x_1), \ldots, \log(x_n)))$$

This relationship between geometric and arithmetic mean will be exploited later.

3 Algorithm Configuration for Hypergraph Partitioning

Our goal is to optimize KaHyPar-CA on a large heterogeneous application-oriented instance set[1]. As described in Section 2.3.6, optimization for heterogeneous instance sets is challenging. To make the optimization process for SMAC easier, we partitioned the instance set into subsets with higher homogeneity according to their application origin (also called *type*). The types are:

- VLSI: These instances are from the ISPD98 VLSI Circuit Benchmark Suite [3] and from the DAC 2012 Routability-Driven Placement Contest [39].
- SPM: The sparse matrices of the SuiteSparse Matrix Collection [9] are translated with the row-net model [38] into hypergraphs. Compared to the other subsets, this subset is relatively heterogeneous.
- SAT: The SAT instances are from the international SAT Competition 2014 [5]. They are translated with three different models into hypergraphs: *literal* [1], *primal* and *dual* [28]. Since these models produce very different hypergraphs, we treat literal, primal and dual as separate types.

Different application fields have different requirements: VLSI design for example is a field where small quality improvements may lead to significant savings [40]. Thus the best possible quality is wanted and running time is not of importance as long as it is not too badly. On the other hand, hypergraph partitioning is also used to accelerate sparse matrix-vector multiplications. This application field does not require the best possible partitions, but fast partitioning. Since quality and running time are conflicting objective functions, optimizing quality *and* running time simultaneously is challenging. The goal is to find a configuration such that there exists no other configuration with better quality *and* better running time. This configuration is called *Pareto optimal*. We call the objective to find such a configuration *Pareto*.

To achieve these objectives, we define a cost function for every optimization objective in Section 3.1. In Section 3.2 we analyze the required computing time for the optimization process and how the required computing time can be reduced in particular. With this knowledge a training set is selected for each type in Section 3.3. After that, in Section 3.4, a benchmark set is built to verify the final configurations found by SMAC. Since we use

[1]The hypergraphs for the instance set can be found at: `https://algo2.iti.kit.edu/schlag/sea2017/benchmark_subset.txt`.

pSMAC (see Section 2.3.5), multiple final configurations are found by SMAC. The topic of Section 3.5 is to determine which final configurations are the best. Finally, a mechanism is introduced in Section 3.6 to terminate intensification of badly performing configurations early if the objective is quality or running time (and thus the adaptive capping mechanism is not available).

3.1 Definition of Cost Functions

As described above, we want different cost functions $\mathfrak{c}$ for the three objectives quality, running time and Pareto. The goal of $\mathfrak{c}$ is to aggregate the result of a single evaluation of a challenger configuration θ_{new} on the target algorithm to a single number. The cost of θ_{new} is calculated by taking the arithmetic mean of the cost values of multiple evaluations. The definitions used for expressing running times and connectivities found by θ_{new} can be found in Section 2.1.3.

Imbalanced partitions are not valid. To reflect that, intensification for configurations which produce imbalanced partitions is terminated. Instead, high penalty values are saved in the run history for these configurations in order to prohibit intensification of configurations in their neighborhood.

3.1.1 Quality

We define the cost function $\mathfrak{c}$ for quality as optimization objective as:

$$\mathfrak{c}(\theta_{new}, \pi, s) = \log\left(\frac{(\lambda-1)(\theta_{new}, \pi, s)}{(\lambda-1)(\theta_{default}, \pi)}\right)$$

The cost function expresses the relative quality improvement of an evaluation compared to the default configuration. As described in Section 2.4, we want to use the geometric mean instead of the arithmetic mean. Although SMAC takes the arithmetic mean, it is still possible to take the geometric mean because of the relationship between arithmetic and geometric mean. Let Π_{common} be the instance-seed-pairs to be evaluated and suppose that there are no imbalanced partitions, the cost of a new configuration θ_{new} is:

$$\mathfrak{c}(\theta_{new}, \Pi_{common}) = \frac{1}{|\Pi_{common}|} \sum_{(\pi,s)\in\Pi_{common}} \log\left(\frac{(\lambda-1)(\theta_{new}, \pi, s)}{(\lambda-1)(\theta_{default}, \pi)}\right)$$

$$= \log\left(\text{geometric\_mean}\left(\frac{(\lambda-1)(\theta_{new}, \pi, s)}{(\lambda-1)(\theta_{default}, \pi)}\right)\right)$$

3.1.2 Running Time

If running time is the optimization objective, the adaptive capping mechanism (see Section 2.3.3) can be used. Because we want to use it, the cost function $\mathfrak{c}$ must be identical with the raw running time. Thus $\mathfrak{c}$ is defined as:

$$\mathfrak{c}(\theta_{new}, \pi, s) = \text{time}(\theta_{new}, \pi, s)$$

In contrast to the quality objective cost function, $\mathfrak{c}$ is not log transformed. Thus not the geometric mean but the arithmetic mean is taken. This has the drawback that evaluations with a long running time have a significantly higher weight than evaluations with shorter running times. It is therefore advisable to select only instances for the training set for which $\theta_{default}$ has a comparable running time.

3.1.3 Pareto

We combine the relative quality improvement and the relative speedup to a single cost function $\mathfrak{c}$ to find configurations with quality improvements as well as speedups:

$$\mathfrak{c}(\theta_{new}, \pi, s) = \log\left(\frac{w \cdot \left(\frac{(\lambda-1)(\theta_{new},\pi,s)}{(\lambda-1)(\theta_{default},\pi)}\right)^x + y \cdot \left(\frac{\text{time}(\theta_{new},\pi,s)}{\text{time}(\theta_{default},\pi)}\right)^z}{w+y}\right)$$

It is possible to adjust the weight parameters w, x, y and z in order to weight quality and running time. Quality improvements are harder to achieve as speedups: It is possible to do less local search moves during the refinement phase in order to save computing time. On the other hand, doing *more* local search moves does not necessarily increases the partition results (e.g. if local search is stuck in 0-gain moves, more moves will not change the partition quality). Thus we square the relative quality improvement ($x = 2$) but not the relative speedup ($z = 1$). Furthermore, we set w is set to 10 and y to 1 based on empirical results. Example function values can be found in Table 3.1. A quality improvement of one percent is worth roughly 1.8%, a speedup of 25% roughly 2.2%. We think that this is a good trade-off between quality and running time, but the weights can be changed if necessary.

Note that $\mathfrak{c}$ is also log transformed, for the same reason as the quality cost function.

3.2 Required Optimization Time

The total number of evaluations needed by SMAC is unknown and depends on the training set (heterogeneous training sets require more evaluations) and the configuration space

		quality								
	Pareto	.95	0.96	0.97	0.98	0.99	1.00	1.01	1.02	1.03
running time	0.25	-0.171	-0.150	-0.130	-0.110	-0.090	-0.071	-0.051	-0.032	-0.013
	0.50	-0.144	-0.124	-0.104	-0.085	-0.066	-0.047	-0.028	-0.009	0.010
	0.75	-0.118	-0.099	-0.080	-0.061	-0.042	-0.023	-0.004	0.014	0.032
	1.00	-0.093	-0.074	-0.055	-0.037	-0.018	0.000	0.018	0.036	0.054
	1.25	-0.068	-0.050	-0.031	-0.013	0.005	0.022	0.040	0.058	0.075
	1.50	-0.044	-0.026	-0.008	0.009	0.027	0.044	0.062	0.079	0.096

Table 3.1: Example function values of the Pareto cost function with $w = 10, x = 2, y = 1, z = 1$.

(more parameters require more evaluations). In Section 3.2.1, we analyze which parameters have what impact on the optimization time Afterwards, we study the required *minimum* optimization time depending on some parameters in Section 3.2.2.

With this knowledge, it is then possible to minimize the required optimization time.

3.2.1 Parameters Affecting Optimization Time

There are multiple different parameters with impact on the overall optimization time:

- Selected training set Π
- Configuration space Θ
- Default configuration $\theta_{default} \in \Theta$ with average running time $\mathrm{time}(\theta_{default})$ on Π
- Cost function $\mathfrak{c}$
- Number of minimum evaluations per configuration R_{min} and the number of maximum evaluations per configuration R_{max}

Half of the optimization time is spent by SMAC on model building and selection of challenger configuration, the other half on intensification of challenger configurations (see Algorithm 2). If the evaluations of the target algorithm are costly, the relative time spent on intensification is even higher. Thus reducing the running time for a single evaluation of a challenger configuration greatly reduces the optimization time.

The number of evaluations required to find a good configuration depends on the difficulty of the problem. For example, we know based on empirical results that optimizing the running time of KaHyPar-CA is easier than achieving quality improvements. Also the needed computing time for evaluating a configuration depends on the optimization objective: With running time as optimization objective, configurations found by SMAC are relatively fast. On the contrary, if the quality optimization objective is used, configurations found by SMAC are expensive. However, this effect can be limited by setting cutoff times.

A challenger configuration is evaluated at most R_{max} times to become the new incumbent configuration. This process is not parallelized. If the challenger configuration has roughly

the same running time as the default configuration, this process will take about $R_{max} \cdot \mathrm{time}(\theta_{default})$. For example, if $\mathrm{time}(\theta_{default})$ is 90 seconds and R_{max} is 100, the challenger configuration needs up to 2.5 hours to become the new incumbent.

R_{min} denotes the minimum number of evaluations per challenger configuration. Most challenger configurations will be evaluated only R_{min} times because they are worse than the incumbent configuration. Thus increasing R_{min} will greatly affect the overall optimization time.

As a consequence, the average running time on the training set, the definition of the cost function and the number of minimum evaluations per challenger configuration is very important. Since we are unable to determine the optimization time, we want to know the *minimum* time required for optimization. We asses this in the following section.

3.2.2 Required Minimum Optimization Time

It is important to ensure that the final configuration is executed at least once on each instance of the training set Π. This will be the case if the incumbent configuration is executed at least $|\Pi|$ times. The time needed for this process is considered as *minimum* optimization time. Since the intensification of the incumbent configuration is not parallelized by pSMAC (see Section 2.3.5), the calculated minimum optimization time cannot be reduced by parallelization.

The incumbent configuration is evaluated every time the intensification procedure is called. The number of calls of the intensification procedure is at most the number of challenger configurations tested divided by two. Additionally, the incumbent configuration starts with $R_{min} - 1$ evaluations.[2] The required number of tested configurations is therefore at least $2 \cdot (|\Pi| - R_{min} + 1)$. The number of evaluations per challenger configurations is at least R_{min}.[3] Thus the number of required evaluations to achieve that the final configuration is executed at least once on each instance of Π is at least $2R_{min} \cdot (|\Pi| - R_{min} + 1)$. By multiplication with the average running time, the minimum optimization time is calculated.

For example, if the training set has a size of 50 (like recommended in Section 2.3.6), the average evaluation take 90 seconds and R_{min} is set to 1, the required minimum optimization time is 2.5 hours. In contrast, if R_{min} is set to 8, 17.2 hours are needed.

The minimum number of evaluations per configuration R_{min}, the size of Π and the average running time have roughly the same impact on the minimum required optimization time.

[2] As initial startup, SMAC evaluates the default configuration $R_{min} - 1$ times.

[3] Actually, by usage of the adaptive cutoff mechanism (see Section 2.3.3) or the dynamic cutoff mechanism (see Section 3.6), it is possible to execute a configuration less than R_{min} times. However, not the number of evaluations but the needed time is important. In case of the adaptive mechanism, if there are less than R_{min} evaluations, the time spent on intensification will be equal to the time needed by the default configuration for R_{min} evaluations. In case of the dynamic cutoff mechanism, only half of this time is needed on average. Since there are also configurations with more than R_{min} evaluations, this is not a problem.

In Section 3.3, we will use the insights obtained in this section to build a training set with a low optimization time.

3.3 Training Set Selection

After the definition of the cost functions in Section 3.1 and a discussion of the impact of several parameters on the required optimization time in Section 3.2, the next step is to select the training sets. Since we want to optimize KaHyPar for each type of instance (VLSI, SPM, etc.), one training set per type is needed. As recommended in Section 2.3.6 for homogeneous instance sets, we choose around 50 instances for each training set.

It is important to select training sets which are representative of the whole instance set. This is done in Section 3.3.1. To save computing time, it is also important to exclude instances which are too huge. Additionally, we exclude all instances for which the default configuration's improvement potential is too low. Our approach to exclude these instances can be found in Section 3.3.3.

3.3.1 Features

Every hypergraph has attributes, so–called *features*. The features of a hypergraph describe the hypergraph. We will consider a training set as representative if the feature space of the instance set is well covered.

Features of hypergraphs used by us are e.g. the number of edges and pins. All in all, 20 different hypergraph features are used, which are listed in Section A.1. Since we used 20 features, the feature space has 20 dimensions. It is therefore difficult to analyze whether the feature space is covered well. We use a *Principal Component Analysis (PCA)* [14] to reduce the dimensionality of the feature space down to five (the first five components are responsible for roughly 90% of the variance). Before we apply the PCA, we log transform, scale and center the feature space in order to improve the quality of the PCA.

An instance consists not only of a hypergraph but also of a specific value for k (the number of blocks the input hypergraph must be partitioned into) and ε (the maximum allowed imbalance). Because ε is set to 0.03 for all instances, ε is ignored in the feature analysis. Since k has a significant impact on the partitioning process, we choose the values of k manually.

Features are not only used to verify the representativeness of our training set but are also used by SMAC for improving the model (see Section 3.3.1). Because SMAC supports up to seven different features without a dimensionality reduction, we choose the first six components of the PCA as well as k as features for SMAC. This reflects the great importance of k.

Algorithm 3: Empirical Variance of Instance

`Data`: Instance π, default configuration $\theta_{default}$, number of repetitions $R_{min} \in \mathbb{N}$, cost function $\mathfrak{c}$

`Result`: Minimal detectable improvement with 95% confidence

// Take K times the geometric means of R_{min} runs

1 $\hat{\mathfrak{C}}(\theta_{default}, \pi) \leftarrow \bigcup_{i=1}^{K} \left\{ \text{geo\_mean} \left(\bigcup_{j=1}^{R_{min}} \{\mathfrak{c}(\theta_{default}, \pi, \text{random seed})\} \right) \right\}$ *// $K \gg R_{min}$*

2 $\mathfrak{c}(\theta_{default}, \pi) \leftarrow \text{geo\_mean}(\hat{\mathfrak{C}}(\theta_{default}, \pi))$

3 $\hat{\mathfrak{c}}(\theta_{default}, \pi) \leftarrow Q_{5\%}(\hat{\mathfrak{C}}(\theta_{default}, \pi))$ *// Q is the quantile function*

4 `return` $\mathfrak{c}(\theta_{default}, \pi) / \hat{\mathfrak{c}}(\theta_{default}, \pi)$

3.3.2 Randomization Analysis

As explained in Section 2.3.2, SMAC is capable of optimizing randomized algorithms. This is done by setting the minimum number of evaluations per configuration R_{min} to an appropriate value. SMAC will then take the mean cost of at least R_{min} evaluations of the target algorithm. If the mean cost of these R_{min} evaluations of a challenger configuration is not better than the cost of the incumbent configuration (on the same instance-seed-pairs), the challenger configuration is discarded, otherwise the challenger is further intensified.

The goal of this section is to determine the minimum value for R_{min} to ensure that a challenger configuration with a certain improvement compared to the incumbent configuration is not discarded after R_{min} evaluations. With this knowledge, we also want to determine the instances for which the randomization of the partition results of KaHyPar-CA is especially high to exclude them from training sets.

At first, we define the expected cost $\mathfrak{c}(\theta)$ of a configuration θ as the mean of an infinite number of runs. We approximate this by performing $K \cdot R_{min}$ runs with $K \gg R_{min}$. SMAC does not know this expected cost and uses the mean of R_{min} runs to determine the empirical cost $\hat{\mathfrak{c}}(\theta)$ of θ instead. The empirical cost $\hat{\mathfrak{c}}(\theta)$ can be lower or higher than the expected cost $\mathfrak{c}(\theta)$. This affects the decision whether a challenger configuration θ_{new} is further intensified.

Let θ_{new} be a challenger configuration and let θ_{inc} be an incumbent configuration with $\mathfrak{c}(\theta_{new}) < \mathfrak{c}(\theta_{inc})$. The challenger will be further intensified iff $\hat{\mathfrak{c}}(\theta_{new}) < \hat{\mathfrak{c}}(\theta_{inc})$.

To determine $\hat{\mathfrak{c}}(\theta_{inc})$, we analyze the set $\hat{\mathfrak{C}}(\theta_{inc})$ of all possible values for the empirical cost of θ_{inc}. We take the 5% quantile of $\hat{\mathfrak{C}}(\theta_{inc})$, which is denoted by $Q_{5\%}(\hat{\mathfrak{C}}(\theta_{inc}))$. Then $\hat{\mathfrak{c}}(\theta_{inc})$ is with 95% confidence at least $Q_{5\%}(\hat{\mathfrak{C}}(\theta_{inc}))$.

Thus the challenger configuration θ_{inc} will be further intensified with a confidence of 95% if its empirical cost is lower than $Q_{5\%}(\hat{\mathfrak{C}}(\theta_{inc}))$.

This insight is used by Algorithm 3. We calculate the minimum empirical cost of the *default* configuration with 95% confidence. A challenger configuration must have a lower cost than $\hat{\mathfrak{c}}$ in order to be further intensified after R_{min} runs. In line 4, we take the fraction

of the expected cost of $\mathfrak{c}$ and $\hat{\mathfrak{c}}$ to express the required improvement in percent instead of absolute values.

With this knowledge, we are able to chose R_{min} in such a way that challenger configurations are not discarded if they have a certain improvement (with a confidence of 95%).

Note that we use the default configuration rather than the incumbent configuration in Algorithm 3. This is required since the incumbent configurations are unknown. We assume that incumbent configurations have roughly the same variation as the default configuration. So the results obtained by Algorithm 3 can at least be taken as an indicator.

Algorithm 3 is limited to a single instance π but can be modified to handle a set of instances Π: Instead of executing only runs for π in line 1, it is possible to execute runs for random instances of Π.

For the selection of R_{min}, we have to distinguish between the different optimization objectives because the cost functions are different. There is relatively low randomization for running time as optimization objective, thus we set R_{min} to one for running time. On the other hand, randomization has an impact on the connectivity of partitions found. Based on our results, we set as optimization objective R_{min} to eight for quality and Pareto.

Furthermore we chose to exclude every instance π from training sets for which, using the quality cost function, $\mathfrak{c}(\theta_{default}, \pi)/\hat{\mathfrak{c}}(\theta_{default}, \pi)$ is higher than 1.5%.

3.3.3 Exclusion of Inappropriate Instances

After determining all instances for which the partitioning results of KaHyPar-CA have a high variance, we go ahead and exclude further instances which are inappropriate for the optimization process.

At first, we choose to exclude all instances for which the running time of the default configuration is higher than six minutes. To save even more computing time during optimization, we choose the training sets in such a way that the average running time of the default configuration is not higher than 120 seconds.

Since we want to optimize the coarsening phase, the initial partitioning phase and the refinement phase of KaHyPar-CA simultaneously, we exclude all instances for which the initial partitioning phase consumes more than 40% of the whole partitioning time (regarding the default configuration). For the same reason, all instances for which the default configurations needs less than 15 seconds for the refinement phase are excluded.

We use Algorithm 3 to identify instances for which the partitioning results have a high variance. Since Algorithm 3 is costly, we use as a preliminary requirement that the proportion of best and mean cut of 10 runs of the default configuration using different seeds is at least 0.95. This are arbitrary values, but only around 10 percent of all instances are further excluded by Algorithm 3.

We want that only instances participate in training sets for which the default configuration has an improvement potential. Instances with a difference between best and mean cut have

an improvement potential - at least the mean cut can be reduced to the value of the best cut. With this knowledge, we only choose instances for which the proportion of best and mean cut of 10 runs of the default configuration using different seeds is at most 0.995.

The selected training sets including their PCA analysis can be found in Section A.3. Note that we had to weaken our selection requirements for the VLSI training set: Available VLSI instances are mostly too small or too huge. For the same reasons, we were left with less than the recommended 50 instances. Since the VLSI instances are relatively homogeneous, this is not a problem.

We select the training set for optimizing the whole instance set by choosing around 20 instances from each training set for every type. We select these in total 100 instances in such a way that the whole feature space is well covered. Note that far larger training sets with at least 300 instances are recommended for optimization of heterogeneous instance sets (see Section 2.3.6). Since evaluating configurations of KaHyPar-CA is expensive and computing time is limited we decide to use despite the recommendation 100 instances. We call this training set "full training set".

Note that we also tried to cluster the feature space of the instances using K-Means [15] to improve the representativeness of the training sets. As recommended by Yeung and Ruzzo [43], we applied K-Means without a Principal Component Analysis as preprocessing step. By verification with silhouettes [34], we revealed that our data is not clustered.

3.4 Finding a Better Benchmark Set

It is important to test a final configuration found by SMAC on a benchmark set different from the training set: It is unlikely but possible that the training set is not representative of the whole instance set. We use a benchmark set to verify that the final configuration found by SMAC works on all instances, and not only on the training set. The process of finding a better benchmark set is described in this section.

We select the benchmark set at random, unlike the training sets. The reasons behind this are twofold: Firstly, if the training sets are *not* representative, a benchmark set selected with the same method as the training set will not be representative either. For the same reason, we decide not to use a PCA to verify the representativeness of the benchmark set. Secondly, benchmarking is not as time crucial as optimization[4]: We have the resources to benchmark instances with a running time of above a few minutes. We do not want to exclude them from the benchmark set. Instances for which the default configuration has only a low improvement potential are not excluded since a quality loss is still possible.

Our method to find a new benchmark set with a specific size n can be found in Section 3.4.1. After that, a method to verify the representativeness of a benchmark set can be found in Figure 8.

[4]It makes a difference whether only a single configuration is tested during benchmarking or as many configuration as possible are evaluated during optimization.

3.4.1 Random Selection of Benchmark Set

We want our benchmark selection algorithm to reflect that there are different types of instances (e.g. VLSI, SPM, etc.). Thus Algorithm 4 first selects a type at random and then a hypergraph (not instance) of this type for the benchmark set. This is repeated until the benchmark set consists of the desired number of hypergraphs. For each hypergraph, all $k \in \{2, 4, 8, 16, 32, 64, 128\}$ are chosen to build the instances with ε set to 0.03.

To save computing time, we excluded hypergraphs for which a $k \in \{2, 4, 8, 16, 32, 64, 128\}$ exists for which a single evaluation of the default configuration consumes more than four hours (line 2). Since only a small percentage of all hypergraphs is excluded, this is not an issue.

Algorithm 4: Random Benchmark Set Selection

`Data`: Size of the benchmark set n, set of hypergraphs G, default configuration $\theta_{default}$
`Result`: Benchmark set

1 $Types \leftarrow \{SAT14Dual, SAT14Primal, SAT14Literal, SPM, VLSI\}$
2 $G \leftarrow \{g \in G \mid \forall k \in \{2, 4, 8, 16, 32, 64, 128\} : \text{time}(\theta_{default}, (g, k, 0.03)) < 4h\}$
3 $Result \leftarrow \emptyset$
4 `while` $|Result| < n$ `do`
5 $type \leftarrow$ select one element from $Types$ at random
6 `if` G *has a element with type* $types$ `then`
7 $hypergraph \leftarrow$ select one element from G with type $type$
8 $G \leftarrow G \setminus hypergraph$
9 $Result \leftarrow Result \cup \{graph\}$
10 return $\{(g, k, \varepsilon) \mid g \in Result \wedge k \in \{2, 4, 8, 16, 32, 64, 128\} \wedge \varepsilon = 0.03\}$

3.4.2 Verifying Representativeness of the Benchmark Set

We want to calculate the probability whether a benchmark set of size n randomly generated by Algorithm 4 is representative. For that, we compare KaHyPar-CA with its strongest competitor, hMetis-R [23], regarding the minimum connectivity metric. For all of the following comparisons, the Wilcoxon matched pairs signed rank test [41] is used with a confidence level of 99%, based on ten runs per instance.

KaHyPar-CA has a significantly better minimum connectivity than hMetis-R on the whole instance set. A representative benchmark set leads to the same conclusion on the whole instance set. Hence KaHyPar-CA needs to have a significant better minimum connectivity than hMetis-R on the generated benchmark set. We consider the probability of this as probability that a random generated benchmark set is representative.

This procedure is implemented by Algorithm 5. Since there is a high number of different subsets for each n, we limit the generation of benchmark sets to 1000 per n.

Algorithm 5: Probability of a benchmark set with n hypergraphs to be representative

`Data`: Size of benchmark set n, set of hypergraphs G, default configuration of KaHyPar-CA $\theta_{default}$, default configuration of hMetis-R $\theta'_{default}$
`Result`: Probability of a benchmark set with n hypergraphs is not considered as representative

$count \leftarrow 0$
`for` $i \leftarrow 1$ `to` 1000 `do`
  $R \leftarrow$ Random_Benchmark_Set_Selection(n, G, $\theta_{default}$) *// Algorithm 4*
  $K \leftarrow \bigcup_{\pi \in R}\{\min_{s \in \{1,\dots,10\}}((\lambda-1)(\pi, \theta_{default}, s))\}$ *// min connectivities KaHyPar- CA*
  $H \leftarrow \bigcup_{\pi \in R}\{\min_{s \in \{1,\dots,10\}}((\lambda-1)(\pi, \theta'_{default}, s))\}$ *// min connectivities hMetis- R*
  `if` *K is significantly better than H with 99% confidence* `then`
    count++
return $1 - \frac{count}{1000}$

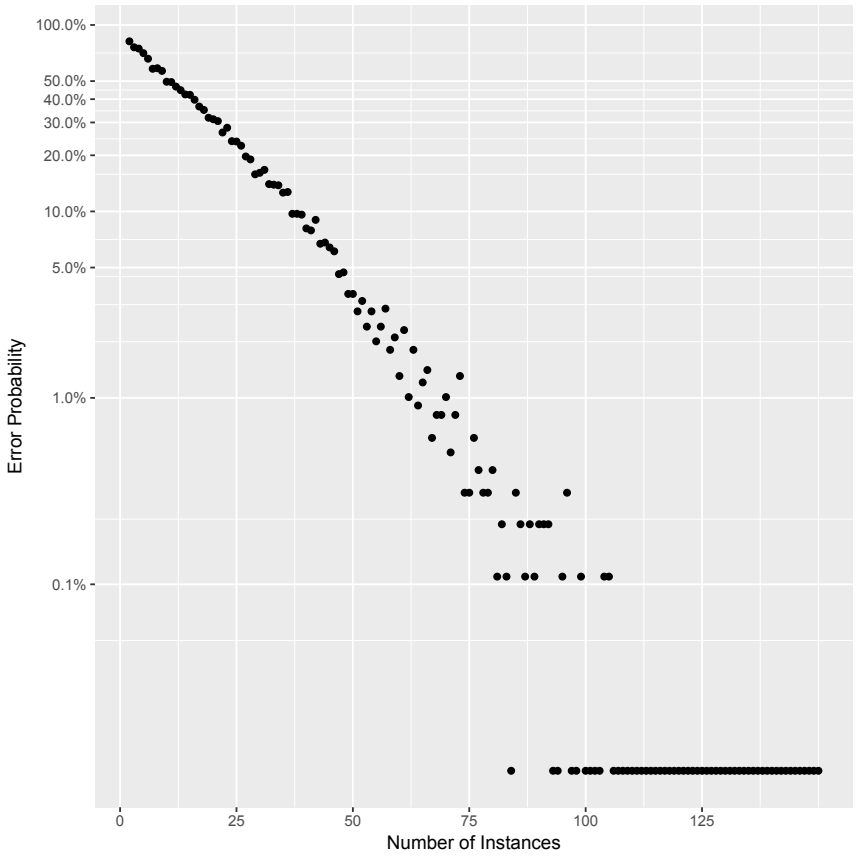


Figure 3.1: Error rate for n instances according to Algorithm 5

For each $n \in \{2, 3, \ldots, 150\}$, the error probability determined by Algorithm 5 is shown by Figure 3.1. The low variance confirms that the evaluation of 1000 different benchmark sets for each n is sufficient. As a trade-off between error probability and size of the new benchmark set, we set the size of the benchmark set to 70. The error rate is then at around one percent. The selected benchmark set can be found in Section A.2.

3.5 Validation of SMAC Configurations

In the previous section, a benchmark set was chosen to verify configurations found by SMAC. Since we use pSMAC (see Section 2.3.5), one configuration per SMAC process will be found. Because benchmarking configurations is expensive, we choose only a few configurations for benchmarking.

One approach is to choose only the configuration with the lowest cost on the training set, however it is not necessarily the best one as explained below:

One issue is over-tuning: Assume a partition of the training set Π into two disjoint non-empty subsets B and C with $\Pi = B \cup C$. Let θ be the configuration with minimum cost on Π. Let the cost of θ be very low on B and relatively high on C. A configuration θ' with a slightly worse cost on Π than θ, but with low costs on B *and* C, is considered to be better than θ by us. However, a configuration with characteristics like θ can only exist if either the size of the training set is too small or the training set is heterogeneous.

Another issue are trade-offs which each configuration makes between quality and running time. These trade-offs affect the decision which configuration is best in practice. Using a cost function to map how good a configuration is to a single value obscures these trade-offs, depending on the optimization objective:

- If the objective is quality, the mean quality gain is known (which is identical to the cost value reported by SMAC), but not the loss in performance.[5]
- If the objective is running time, the mean partitioning time improvement is known (which is identical to the cost value reported by SMAC), but the loss in quality is unknown.
- If the objective is Pareto, the loss/gain of quality *and* performance are unknown.

Hence we select not only the best configurations with the lowest cost but the three different configuration with the lowest costs.

3.6 Dynamic Capping Time

In the following we describe how badly performing executions of the target algorithm can be terminated by SMAC in order to save computing time. Burger et al. [8] emphasize

[5] Although the loss in performance is capped by the dynamic cutoff time.

the importance of correct measurement of performance and planning timeouts: For a specific scenario of Burger et al., nearly every evaluation of a challenger configuration was terminated by SMAC due to a too small timeout.

Because of high variance of the partitioning times of KaHyPar-CA for different instances, choosing a good cutoff time is difficult: In some cases, the partitioning time on the default configuration on the biggest instances is up to 10 times higher than of the smallest instance.

If the optimization objective is running time, this is not a problem due to the adaptive capping mechanism described in Section 2.3.3: In Section 3.3.2 we have set the minimum number of evaluations R_{min} to 1 for running time as optimization objective. This means that during the first iteration of intensification, an evaluation of a challenger configuration on an instance-seed-pair is terminated after the time it took the incumbent configuration to partition this instance-seed-pair.

Also the difficulty of setting proper timeouts is not a problem for quality as optimization objective because high cutoff times are wanted in order to achieve maximum quality.

On the other hand, for Pareto as optimization objective, this problem exists: The Pareto function is designed in such a way that if the running time of a challenger configuration is much worse than the running time of the default configuration, the challenger needs a high quality improvement to become the new incumbent configuration. This is unlikely to happen. Thus a call of the target algorithm should be terminated if the running time gets too high. Since the adaptive capping mechanism works only for running time as optimization objective (as described in Section 2.3.3), another mechanism is needed.

The key to solve this problem is to set the cutoff time not in relation to the current incumbent configuration but in relation to the default configuration:

$$\text{cutoff\_time}(\theta_{default}, \pi) = d \cdot \text{time}(\theta_{default}, \pi),\ d \in \mathbb{R}, d \geq 1$$

An evaluation of a challenger configuration θ_{new} exceeding this time limit is terminated immediately. Like in the adaptive capping mechanism, intensification is terminated for θ_{new} and a high penalty value is saved for this evaluation in the run history R to ensure that no configuration in the neighborhood of θ_{new} will be intensified in the future.

The constant d should be chosen carefully: A configuration will be dropped directly after timing out, previous results are not considered. For example, even if the first 99 runs worked fine and the 100th times out, the configuration will be dropped. To avoid these problems, we chose high values for d: If the optimization objective is Pareto, we set d to 3, for quality as optimization objective we set d to 5. Note that we use the adaptive capping mechanism instead of the dynamic capping mechanism for running time as optimization objective.

This weakness of dropping a configuration after the first timeout is also the strength of the dynamic capping mechanism: For high values of R_{min} (which denotes the minimum number of evaluations per configuration), the dynamic capping mechanism is able to cancel the first evaluation of a configuration. In contrast, the adaptive mechanism cannot. In case

of running time as optimization objective, this is not a problem since we chose $R_{min} = 1$, but for Pareto and quality we chose $R_{min} = 8$, which we consider as high.

4 Experimental Evaluation

In this chapter, we execute SMAC to optimize KaHyPar-CA. Before SMAC can be executed, it is important to define the configuration space Θ. After describing all optimization parameters in Section 4.1, our environment and benchmarking method is described in Section 4.2. Since we want to compare the configurations found by SMAC with the default configuration of KaHyPar-CA, it is important to know what exactly the default configuration is. This is done in Section 4.3. The actual results can be found in Section 4.4.

4.1 Tuning Parameters

In Section 2.2.2, KaHyPar-CA is described with special attention to some of its parameters. These parameters often have a value range from 0 to $(2^{32} - 1)$ (unsigned 32 bit integer). Since most of these values are not usable in practice, we restrict the value range of the parameters to reasonable values.

Also, some parameter choices have no effect under certain circumstances. For instance, if KaHyPar-CA uses the adaptive stopping rule for initial partitioning, then the configuration for the simple stopping rule has no effect since the simple stopping is not used. Parameter combinations for which this is the case are deactivated by us.

Additionally, we reduce the step size s of most parameters: Instead of a discrete parameter x with value range $[d, e]$, we use a discrete parameter x' with value range $[d', e']$ and a step size s such that $e = s(e' - d') + d$ and $d = s \cdot d'$. Continuous parameters are discretized using the same principle.

The parameters used for optimization are listed in the following. Since KaHyPar-CA uses another full hypergraph partitioner for initial partitioning, namely KaHyPar-R, all listed parameters are also available during initial partitioning if not stated otherwise.

4.1.1 Coarsening Parameters

In the following, the optimized coarsening parameters are listed. Please note that "choice of" always refers to an enumeration parameter.

- The choice of the coarsening algorithm: Whether KaHyPar-CA visits vertices in random order or uses a priority queue to determine the next vertex to visit.

- The choice of prefering unmatched vertices for contraction during coarsening if two or more vertices have the best rating score. The alternative tie-breaking strategy is random selection.
- The choice giving a penalty for vertices that are too heavy.
- Coarsening ends if the coarsest hypergraph has at most $k \cdot t$ nodes. We call the maximum allowed number of nodes after coarsening of KaHyPar-CA ct and the maximum allowed number of nodes after coarsening of its initial partitioner KaHyPar-R ict. Since the input hypergraph for the initial partitioner KaHyPar-R has at most ct nodes, it has no effect to configure ict higher than ct.

  To reflect this, we configure ct as $\widetilde{ct}$ and set $ct = ict + \widetilde{ct}$. The value range of ict is set to $[5, 200]$ and the value range of $\widetilde{ct}$ is $[0, 150]$. Using a step size of five, ict translates to a discrete parameter ict' with range $[1, 40]$ and $\widetilde{ct}$ translates to a discrete parameter $\widetilde{ct}'$ with range $[0, 30]$.
- KaHyPar-CA does not allow that a vertex v with $c(V) > c_{max} := s \cdot \lceil \frac{c(v)}{k \cdot t} \rceil$ participates in any further contractions, whereas s is a continuous (floating point) parameter. We set the value range of s to $[1, 7]$ with a step size of $1/4$. As described, this translates into a discrete value range of $[4, 28]$.

4.1.2 Actual Initial Partitioning Parameters

After the coarsening of KaHyPar-CA and after the coarsening of KaHyPar-R as initial partitioner, a pool of initial partitioning algorithms is executed multiple times. The number of repetitions is parameterized. Since the effect of more repetitions decreases with higher values, the effect of this parameter is marked as logarithmic. We do not use any step size and set the value range of this discrete parameter to $[1, 100]$.

Note that this is the only parameter for the actual initial partitioning phase optimized by us.

4.1.3 Refinement Parameters

KaHyPar-CA has support for two different stopping rules for refinement: The simple and the adaptive stopping rule. The simple stopping rule is limited by us to the discrete range $[10, 500]$, the adaptive stopping rule has a continuous range of $[1, 5]$. Using a step size of ten, the value range of the stopping rule translates to the discrete range $[1, 50]$. The adaptive stopping rule is not transformed.

Since the simple stopping rule is very expensive even for small values, we choose to remove it for the refinement phase of KaHyPar-CA.

4.2 Environment and Methodology

All experiments are performed on nodes of bwUniCluster[1]. Each node consists of two Intel Xeon E5-2670 processors clocked at 2.6 GHz and has 64 GB main memory with deactivated swap. In order to ensure that each SMAC process has enough memory, we use only eight of sixteen available cores on each node. The same system is used for benchmarking. In order to increase the robustness and reproducibility of the results, only one core of each node is used for benchmarking.

For each type, we run SMAC with the corresponding training set for each optimization objective on three nodes (in total 24 cores) for two days (running time and Pareto as optimization objectives) or three days (quality as optimization objective). We also optimize KaHyPar-CA for all types simultaneously using the full training set on three nodes for three days (running time) or six days (quality and Pareto).

We compare a configuration θ on the benchmark set regarding three different metrics: The *average* connectivity of θ, the *minimum* connectivity of θ and the average running time of θ. For each instance of the benchmark set, ten runs are performed and the above listed metrics are calculated using the *arithmetic* mean. To weigh instances correctly, the *geometric* mean is used in the following when the mean over different instances is taken. We report the improvements of the new configurations found by SMAC in percent. Additionally, the arithmetic mean of the running time is reported.

The Wilcoxon matched pairs signed rank test [41] is used to determine whether the quality improvements can be considered significant. At a confidence level of 99%, a Z-score with $|Z| < 2.58$ is considered significant. A negative Z-score means that θ performs better than the default configuration, a positive Z-score means that the default configuration performs better than θ. The Z-scores can be found in Section A.5.

To give a more detailed analysis, we use *improvement plots* to compare our configurations with the default configuration of KaHyPar-CA as well as with other hypergraph partitioners. We calculate the improvement of our configuration compared to the corresponding algorithm for each instance in percent. For each algorithm, improvements are sorted in descending order.

4.3 Default Configuration

Since we use the default configuration $\theta_{default}$ in many definitions, it is important to know what exactly the default configuration is. A proper selection of the default configuration is important to ensure that the adaptive capping mechanism (see Section 2.3.3) and the randomization measurement (see Section 3.3.2) work well.

[1] `http://www.scc.kit.edu/dienste/bwUniCluster.php`

As a starting point for our default configuration, we use the default configuration of KaHyPar-CA [2]. We discovered that the community detection is the bottleneck for some instances and consumes a relatively large amount of the overall running time (over 50% for some instances). To reduce the computing time consumed by community detection, we reduce the number of iterations of the Louvain algorithm used for community detection from 100 to 10. As shown in Section 4.3 and Table 4.3, this speeds up KaHyPar-CA by up to 60%, while quality is unchanged. The speedup is especially high for small instances. Thus we use this changed configuration as default configuration for all of our experiments. We call this new default configuration *it10*.

Benchmark	Min $\lambda - 1$			Avg $\lambda - 1$		
Set	base	sig.	Impr. [%]	base	sig.	Impr. [%]
SPM	8904	-	-0.17	9224	-	0.05
VLSI	9150	-	0.18	9412	-	0
Dual	2251	-	-0.09	2339	-	-0.03
Literal	28394	-	0.56	29534	-	0.48
Primal	11277	-	-0.32	11606	-	0.03
*	8929	-	0.03	9240	-	0.11

Table 4.1: Comparison of the quality improvements of our new default configuration over the original default configuration of KaHyPar-CA regarding the average connectivity and the minimum connectivity metric. The column labeled "sig." denotes whether there are significant changes.

4.4 Results

As mentioned in Section 4.2, we report our results for executing SMAC for each training set (SPM, VLSI, SAT Primal/Literal/Dual and the full training set) and for each optimization objective (running time, quality and Pareto) separately. As described previously in Section 3.5, three configurations are benchmarked for every optimization objective and every type. We benchmark these configurations not only on the instances of our benchmark set with equal type but also on the full benchmark set. Finally in Section 4.4.4, for each optimization objective we compare the best performing configuration on the full benchmark set with PaToH and hMetis.

[2] Which can be found here: `https://github.com/SebastianSchlag/kahypar/blob/master/config/km1_direct_kway_sea17.ini`.

Benchmark	Geometric Running Time			Arithmetic Running Time	
Set	base time $[s]$	new time $[s]$	speedup	base time $[s]$	new time $[s]$
SPM	22	18.5	1.19	158.6	150.7
VLSI	42.3	28.3	1.50	215.8	158
Dual	24.8	24.4	1.02	81.8	82.4
Primal	16.9	10.7	1.58	71.3	37.2
Literal	72.6	46.3	1.57	508	451.1
*	31.2	23.3	1.34	209.6	179.1

Table 4.2: Comparison of the speedups of our new default configuration over the original default conifguration of KaHyPar-CA. The column labeled "sig." denotes whether there are significant changes.

4.4.1 Running Time

As shown in Table 4.3, SMAC is able to achieve high speedups for all types except Dual. Since the running time cost function ignores quality, the quality loss is up to around 10 percent on average, depending on the type.

Table 4.3 provides some more insight: The quality loss of the configuration optimized on the Dual training set is relatively low compared to the other configurations, on the other hand a slight speedup is observed. This suggests that the default configuration is already optimized for running time on Dual.

Surprisingly the configuration found by optimizing KaHyPar-CA on all types simultaneously has an average quality loss of around 1.3% on SPM while the configuration optimized for SPM has an average quality loss of over 9%. Nevertheless both configurations have a roughly similar speedup on SPM. We assume that the configuration optimized for all types takes in general more "stable" decisions in order to work on all instances than the configurations optimized for one type only.

We consider the configuration which is optimized for the full training set as the best configuration. It is true that the configurations optimized for Literal and Primal are slightly faster but have worse quality. In Section 4.4.4 we compare this configuration with PaToH and hMetis.

4.4.2 Quality

As shown in Table 4.4, the configurations optimized for a specific type achieve not only significant quality improvements on their type but also on the full benchmark set. The quality improvements are achieved with an average slowdown of a factor of up to 4. This is due to the fact that the quality cost function ignores running time as long as the dynamic

Training Set	Benchmark Set	Min $\lambda-1$			Avg $\lambda-1$			Geometric Running Time			Arithmetic Running Time	
		base	sig.	Impr. [%]	base	sig.	Impr. [%]	base [s]	new [s]	speedup	base [s]	new [s]
SPM	SPM	8919	x	-8.43	9219	x	-9.72	18.5	9.3	1.99	150.7	90.0
	*	configuration was unable to partition 7 SAT instances										
VLSI	VLSI	9134	x	-0.73	9411	x	-2.41	28.3	15.4	1.83	158.0	122.6
	*	8926	x	-1.58	9230	x	-3.21	23.3	15.7	1.49	179.1	215.5
Primal	Primal	11314	x	-5.21	11603	x	-10.4	10.7	5.4	1.97	37.2	18.9
	*	8926	x	-4.11	9230	x	-7.14	23.3	14.3	1.63	179.1	129.3
Literal	Literal	28235	x	-5.03	29393	x	-7.68	46.3	26.7	1.74	451.1	316.1
	*	8926	x	-3.87	9230	x	-6.63	23.3	14.2	1.64	179.1	132.1
Dual	Dual	2253	x	-1.51	2339	x	-2.43	24.4	19.6	1.24	82.4	77.0
	*	8926	x	-1.27	9230	x	-2.45	23.3	16.1	1.45	179.1	172.9
*	SPM	8919	x	-1.11	9219	x	-1.30	18.5	10.5	1.77	150.7	91.1
	VLSI	9134	x	-4.68	9411	x	-7.50	28.3	16	1.77	158.0	112.0
	Primal	11314	x	-4.14	11603	x	-9.25	10.7	5.6	1.90	37.2	19.5
	Literal	28235	x	-5.06	29393	x	-7.61	46.3	26.3	1.76	451.1	317.3
	Dual	2253	x	-3.85	2339	x	-5.19	24.4	23.3	1.05	82.4	95.2
	*	8926	x	-3.69	9230	x	-5.98	23.3	14.5	1.61	179.1	129

Table 4.3: Comparison of our running time optimized configurations with our default configuration. The first column denotes the training set for which the configuration is optimized. The full training set is denoted by '*'. The second column denotes which instances are used for benchmarking, a '*' denotes that the full benchmark set is used. The fourth column labeled "sig." denotes whether the quality changes are significant. Positive values for quality improvements and speedups indicate that our new configuration is better than the default configuration, negative values the opposite.

capping mechanism (see Section 3.6) does not come into effect, which caps the running time loss to a factor of 5. The dynamic capping mechanism is probably the reason why the configuration optimized for all types has a lower quality on the full benchmark set than the Primal and Literal configurations on the full benchmark set: In fact the configurations optimized for Primal or Literal are much slower on the full benchmark set than the configuration optimized for the full training set.

Note that since SMAC optimizes average cost values and not minimum cost values the mean connectivity improvements are with one exception by far better than the minimum connectivity improvements.

We consider the configuration optimized on the full training set as the best configuration. We decide so even though we know that the configurations optimized for Literal and Primal produce nearly 1% better partitions on average, but the slowdown is too high. A comparison with PaToH and hMetis is provided in Section 4.4.4.

4.4.3 Pareto

Our results for Pareto are shown in Table 4.5. We achieve significant quality improvements for VLSI, SPM and Dual. The speedups for these three types are different: While we also manage to achieve a speedup of around 30% for VLSI, there are no speedups for SPM and 10% slower partitioning times for Dual. The reason for this is probably that different types have different optimization potentials.

However, the quality improvements for Literal are not significant and there is no speedup. Only 2 out of 24 SMAC processes were able to find a configuration with a cost lower than the cost of the default configuration on the training set. The best has a reported improvement of 0.81%. Since the reported improvement is low, it is not surprising that the quality improvements are not significant. We gave SMAC another three days to optimize KaHyPar-CA on Literal, but no new configurations were found except one with a reported improvement of 0.03%. We do not know why SMAC has so much difficulty in optimizing KaHyPar-CA on Literal. Since the Literal instances participate in benchmark sets since the first version of KaHyPar, KaHyPar-R, we assume that KaHyPar-CA is already optimized for Literal.

SMAC was unable to find any configurations for Primal. We gave SMAC another three days to optimize KaHyPar-CA on Primal, but still no configurations were found. If we consider the quality improvement of the quality optimized configurations as the maximum possible quality improvement, then the possible quality improvement for Primal is, according to Table 4.4, only around 0.5% on average. This relatively small quality improvement is gained with a 3 times higher partitioning time. Since the possible quality improvement is low and since quality improvements have a high impact on our Pareto cost function (see Section 3.1.3), we assume that the default configuration is already Pareto optimized.

As described in Section 2.3.6, it is difficult to optimize heterogeneous instance sets. Although SMAC had six days for optimizing KaHyPar-CA on the full training set, only a few

Training Set	Benchmark Set	Min $\lambda - 1$			Avg $\lambda - 1$			Geometric Running Time			Arithmetic Running Time	
		base	sig.	Impr. [%]	base	sig.	Impr. [%]	base [s]	new [s]	speedup	base [s]	new [s]
SPM	SPM	8919	x	1.180	9219	x	2.150	18.5	40.4	-2.183	150.7	228.2
	*	8926	x	1.000	9230	x	1.480	23.3	42.9	-1.841	179.1	262.5
VLSI	VLSI	9134	x	1.100	9411	x	1.780	28.3	79.0	-2.791	158.0	182.4
	*	8926	x	0.790	9230	x	1.180	23.3	62.9	-2.701	179.1	223.9
Primal	Primal	11314	x	0.870	11603	x	0.550	10.7	40.5	-3.780	37.2	132.4
	*	8926	x	1.740	9230	x	2.160	23.3	77.8	-3.343	179.1	457.6
Literal	Literal	28235	x	1.300	29393	x	1.920	46.3	134.5	-2.905	451.1	1157.8
	*	8926	x	1.640	9230	x	1.910	23.3	65.7	-2.823	179.1	495.3
Dual	Dual	2253	x	1.410	2339	x	2.360	24.4	36.7	-1.505	82.4	113.6
	*	8926	x	0.890	9230	x	1.210	23.3	32.8	-1.410	179.1	220.2
*	SPM	8919	x	0.810	9219	x	1.470	18.5	36.7	-1.982	150.7	198.7
	VLSI	9134	x	1.160	9411	x	1.670	28.3	54.2	-1.915	158.0	198.5
	Primal	11314	-	0.480	11603	-	-0.520	10.7	19.1	-1.785	37.2	39.6
	Literal	28235	-	1.090	29393	-	1.540	46.3	77.9	-1.681	451.1	660.6
	Dual	2253	x	0.810	2339	x	1.770	24.4	42.8	-1.756	82.4	122
	*	8926	x	0.880	9230	x	1.240	23.3	42.5	-1.826	179.1	248.4

Table 4.4: Comparison of our quality optimized configurations with our default configuration. The first column denotes the training set for which the configuration is optimized. The full training set is denoted by '*'. The second column denotes which instances are used for benchmarking, a '*' denotes that the full benchmark set is used. The fourth column labeled "sig." denotes whether the quality changes are significant. Positive values for quality improvements and speedups indicate that our new configuration is better than the default configuration, negative values the opposite.

configurations with reported improvements of 0.1% (compared to the default configuration) were found. However, despite the difficulties, we achieved significantly better partitioning results than the default configuration on the whole benchmark set with roughly the same partitioning time.

On the other hand, the configuration found for optimizing only Dual has also significantly better partitioning results than the default configuration on the whole benchmark set with an overall better partitioning time. We consider the configuration optimized for Dual only as the overall best Pareto optimized configuration.

4.4.4 Comparison with PaToH and hMetis

In this section, we compare our best running time, quality and Pareto optimized configurations with hMetis and PaToH. As shown by Table 4.6 *all* of our selected configurations are faster than hMetis on average, even our quality optimized configuration. Our quality and Pareto optimized configurations produce the overall best partitions according to Table 4.7 and Table 4.8.

However, Table 4.7 and Table 4.8 also show that partitioning results of hMetis are strongly dependent on the types of instances. For example for Dual instances, hMetis-R produces partitions with on average 25% worse quality than KaHyPar-CA. On the other hand, results of hMetis-R for VLSI, SPM and Primal are comparable to KaHyPar-CA. Thus the geometric mean is not necessarily appropriate for measuring the performance of hMetis. Instead we consider the improvement plots found in Figure 4.1 which provide more information[3]:

Our quality optimized configuration is able to outperform hMetis on 60% to 70% of the instances (depending on whether the minimum or the average connectivity metric is used). Furthermore our Pareto optimized configuration produces comparable results regarding the average connectivity metric and is on 65% of the instances better than hMetis regarding the minimum connectivity metric. Furthermore our Pareto configuration is around 2.5 times faster than hMetis according to Table 4.6.

However, our quality, Pareto and running time optimized configurations are still slower than PaToH by an order of magnitude. It is unlikely that further optimizations will change this since the gap is too big. Furthermore the quality loss of our running time optimized configuration is so high that PaToH could be used instead. Instead, it may be worth it to use another Pareto cost function with a higher weighted running time (see our Pareto configuration in Section 3.2.1) to obtain faster configurations while maintaining quality.

[3] Further improvement plots can be found in the appendix: Section A.6

Training Set	Benchmark Set	Min $\lambda-1$			Avg $\lambda-1$			Geometric Running Time			Arithmetic Running Time	
		base	sig.	Impr. [%]	base	sig.	Impr. [%]	base [s]	new [s]	speedup	base [s]	new [s]
SPM	SPM	8919	-	0.950	9219	x	1.720	18.5	16.6	1.117	150.7	152.0
	*	8926	-	0.270	9230	-	0.170	23.3	20.0	1.165	179.1	186.7
VLSI	VLSI	9134	x	0.830	9411	x	0.920	28.3	21.1	1.343	158.0	125.7
	*	configuration caused imbalanced partitions for 14 SAT instances										
Primal		no configurations found										
Literal	Literal	28235	-	0.170	29393	-	0.470	46.3	42.8	1.083	451.1	526.3
	*	8926	-	0.090	9230	-	0.010	23.3	19.9	1.172	179.1	199.8
Dual	SPM	8919	-	0.380	9219	-	1.000	18.5	16.1	1.151	150.7	106.3
	VLSI	9134	x	0.630	9411	x	0.220	28.3	23.2	1.218	158.0	126.8
	Primal	11314	-	-0.140	11603	-	-1.000	10.7	8.3	1.288	37.2	21.7
	Literal	28235	-	-0.350	29393	-	-0.580	46.3	38.2	1.212	451.1	424.2
	Dual	2253	x	1.280	2339	x	1.480	24.4	27.1	-1.111	82.4	97.1
	*	8926	x	0.370	9230	x	0.280	23.3	20.4	1.141	179.1	157.6
*	SPM	8919	x	0.740	9219	x	1.360	18.5	19.2	-1.037	150.7	194.2
	VLSI	9134	-	0.210	9411	-	0.410	28.3	25.8	1.098	158.0	135.2
	Primal	11314	-	-0.150	11603	-	-1.700	10.7	8.1	1.321	37.2	21.8
	Literal	28235	-	0.180	29393	-	-0.120	46.3	39.5	1.172	451.1	443.6
	Dual	2253	x	0.440	2339	-	0.720	24.4	28.3	-1.164	82.4	102.2
	*	8926	x	0.310	9230	-	0.220	23.3	21.9	1.061	179.1	184.3

Table 4.5: Comparison of our Pareto optimized configurations with our default configuration. The first column denotes the training set for which the configuration is optimized. The full training set is denoted by '*'. The second column denotes which instances are used for benchmarking, a '*' denotes that the full benchmark set is used. The fourth column labeled "sig." denotes whether the quality changes are significant. Positive values for quality improvements and speedups indicate that our new configuration is better than the default configuration, negative values the opposite.

Algorithm/Configuration	SPM	VLSI	Primal	Literal	Dual	*
it10	18.5	28.3	10.7	46.3	24.4	23.3
KaHyPar-CA	22.0	42.3	16.9	72.6	24.8	31.2
hMetis-R	49.6	79.9	34.9	133.0	97.6	71.7
hMetis-K	47.7	55.3	23.7	109.3	58.2	53.6
PaToH-Q	4.0	5.0	3.7	10.4	4.9	5.2
PaToH-D	0.8	1.0	0.6	1.7	1.3	1.0
Runtime	10.5	16.0	5.6	26.3	23.3	14.5
Pareto	16.1	23.2	8.3	38.2	27.1	20.4
Quality	36.7	54.2	19.1	77.9	42.8	42.5

Table 4.6: Comparison of the average running times of different partitioners with our configurations. We use the geometric mean to average across instances on our benchmark set. The '*' refers to the full benchmark set.

Algorithm/Configuration	SPM		VLSI		Primal		Literal		Dual		*	
it10	8919.5		9133.9		11314.0		28235.0		2252.9		8926.4	
KaHyPar-CA	0.17	-	-0.17	-	0.32	-	-0.56	-	0.09	-	-0.03	-
hMetis-R	-4.87	-	-0.69	x	-0.22	-	-4.12	x	-26.63	-	-8.01	x
hMetis-K	-4.33	x	-1.69	x	-1.15	-	-8.36	x	-15.64	-	-6.48	x
PaToH-Q	-3.37	x	-9.24	x	-7.86	x	-12.52	x	-9.73	x	-8.47	x
PaToH-D	-4.05	x	-10.48	x	-7.42	x	-13.07	x	-11.32	x	-9.23	x
Runtime	-1.11	x	-4.68	x	-4.14	x	-5.06	x	-3.85	x	-3.69	x
Pareto	0.38	-	0.63	x	-0.14	-	-0.35	-	1.28	x	0.37	x
Quality	0.81	x	1.16	x	0.48	-	1.09	-	0.81	x	0.88	x

Table 4.7: Comparison of the minimum connectivities of different partitioners with our configurations. We use the geometric mean to average across instances on our benchmark set. The '*' refers to the full benchmark set. The first row denotes the average minimum connectivities of our default configuration. The other rows denote the quality change in percent. Positive numbers indicate that the corresponding algorithm performs better than our default configuration and negative numbers the opposite. An 'x' denotes that quality changes are significant, a '-' the opposite.

Algorithm/Configuration	SPM		VLSI		Primal		Literal		Dual		*	
it10	9219.4		9411.3		11602.6		29393.1		9229.8			
KaHyPar-CA	-0.05	-	-0.00	-	-0.03	-	-0.48	-	0.03	-	-0.11	-
hMetis-R	-2.90	-	0.55	-	0.36	-	-3.06	x	-25.50	-	-6.77	-
hMetis-K	-3.24	x	-0.82	x	-1.74	-	-7.77	x	-16.57	-	-6.25	x
PaToH-Q	-0.13	x	-6.48	x	-5.51	x	-8.93	x	-6.27	x	-5.36	x
PaToH-D	-4.68	x	-12.32	x	-10.62	x	-13.87	x	-12.40	x	-10.67	x
Runtime	-1.30	x	-7.50	x	-9.25	x	-7.61	x	-5.19	x	-5.98	x
Pareto	1.00	-	0.22	x	-1.00	-	-0.58	-	1.48	x	0.28	x
Quality	1.47	x	1.67	x	-0.52	-	1.54	-	1.77	x	1.24	x

Table 4.8: Comparison of the average connectivites, with the same structure as Table 4.7.

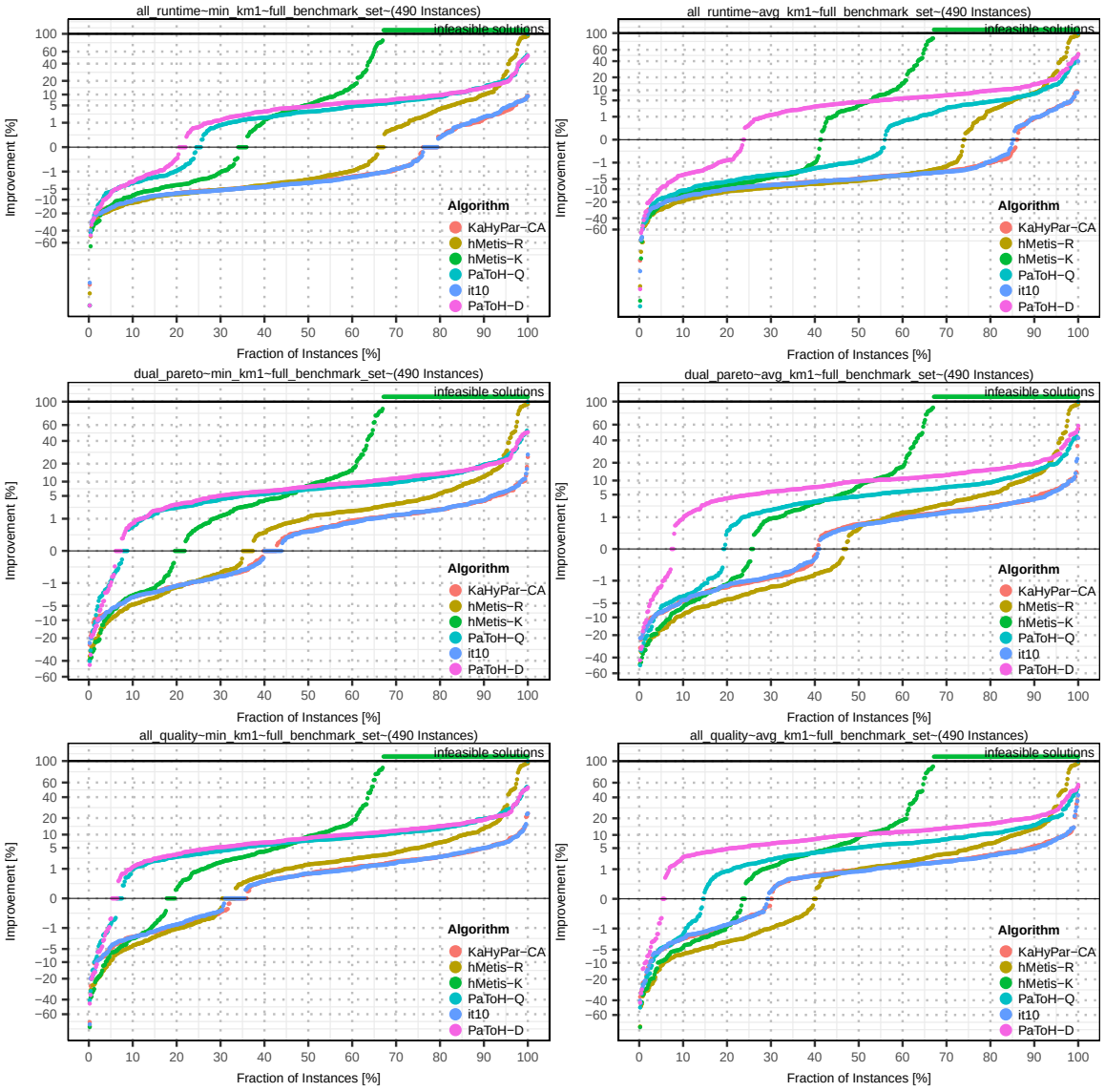


Figure 4.1: Performance plots for our overall best configurations against hMetis and PaToH. Top: Our best running time optimized configuration is taken as base line. Mid: Our Pareto optimized configuration is taken as base line. Bottom: Our quality optimized configuration is taken as base line.

5 Discussion

5.1 Conclusion

We presented our approach to optimizing the hypergraph partitioner KaHyPar-CA [17] using SMAC [18]. We defined cost functions for different optimization objectives (running time, quality and Pareto). Instances chosen for training sets fulfill various requirements:

- We ensured that all phases of KaHyPar (coarsening, initial partitioning and refinement) are optimized simultaneously by exclusion of instances for which initial partitioning consumes more than 40% of the total partitioning time and by exclusion of instances with a refinement phase shorter than 15 seconds (both regarding the default configuration).
- By considering the effects of randomization, we excluded all instances for which the default configuration's improvement potential is too low. In order to ensure that improvements of challenger configurations compared to the incumbent configuration are not obscured by randomization, we excluded all instances for which the default configuration has a high variance.
- To save computing time, we excluded all instances for which the default configuration has a running time higher than six minutes. Additionally we selected the instance set in such a way that the average running time of the default configuration is not higher than 120 seconds.
- We used a Principal Component Analysis to determine whether our selected training instances are representative.

Since we restricted most instances from participating in training sets, we considered the possibility that our training sets are not representative. Thus we evaluated final configurations found by SMAC on a benchmark set different from the training set. We selected our benchmark set randomly, unlike the training sets: If the training sets are not representative, a benchmark set selected using the same method as the training set will not be representative either. We compare partitioning results of KaHyPar-CA with hMetis-R using significance tests to decide whether a randomly generated benchmark set is considered representative or not.

We analyzed the parameters of SMAC and found that there is a minimum optimization time which is not reduced by parallelization using pSMAC. To save computing time, we introduced our dynamic capping mechanism to cap poorly performing evaluations of challenger configurations. The key was to set the cutoff time in relation to the running time of

the default configuration for each instance instead of using a single static cutoff time for all instances.

Despite the difficulties of optimizing for heterogeneous instance sets, we achieved significant quality improvements of 0.88% on average and average speedups of 1.66, depending on the optimization objective. Furthermore our quality and Pareto optimized configurations produce better partitions than hMetis [23, 24] *and* are faster than hMetis.

5.2 Future Work

Although we optimized KaHyPar-CA for a considerable number of parameters, there are still many unoptimized ones, especially the community detection parameters. Since we achieved speedups of 34% on average by optimizing one community parameter manually, we assume that there is still a lot of improvement potential. Furthermore new variants of KaHyPar have been developed, namely KaHyPar-E [4] and KaHyPar-MF [16]. They introduce new parameters which can also be optimized.

We optimized KaHyPar-CA only for $\varepsilon = 0.03$. By adding instances with different values for ε, it is possible to optimize KaHyPar-CA for other values of ε.

Another open problem is the optimization of other hypergraph partitioners such as hMetis [23] and PaToH [38]: The strength of PaToH is its speed. A running time optimization of PaToH may lead to even faster configurations. On the other hand, the strength of hMetis is quality but the partitioning times are high. A Pareto optimization of hMetis may reduce the partitioning times of hMetis while maintaining quality.

A Appendix

A.1 Features of Hypergraphs

These are the used features of hypergraphs, in total 22.

- number of hypernodes
- number of nets
- number of pins
- average net size
- standard deviation of $\{|e| \mid e \in E\}$
- minimum net size
- 90th percentile of the size of $\{|e| \mid e \in E\}$
- first quantile of $\{|e| \mid e \in E\}$
- median of $\{|e| \mid e \in E\}$
- third quantile of $\{|e| \mid e \in E\}$
- maximum net size, $\max\{|e| \mid e \in E\}$
- average hypernode size
- standard deviation of $\{|v| \mid v \in V\}$
- average hypernode degree
- standard deviation of $\{d(v) \mid v \in V\}$
- minimum hypernode degree
- 90th percentile of $\{d(v) \mid v \in V\}$
- maximum hypernode degree
- first quantile of $\{d(v) \mid v \in V\}$
- median of $\{d(v) \mid v \in V\}$
- third quantile of $\{d(v) \mid v \in V\}$
- density

A.2 New Benchmark Set

type	hypergraph	$\|V\|$	$\|E\|$	|Pins|	avg $d(v)$
SPM	circuit_3	12127	12127	48137	3.96941
	bibd_49_3	18424	1176	55272	47
	ca-CondMat	23133	23133	186936	8.08092
	astro-ph	16706	16046	242502	15.1129
	us04	28016	163	297538	1825.39
	lp_nug20	72600	15240	304800	20
	shallow_water2	81920	81920	327680	4
	Franz11	30144	47104	329728	7
	RFdevice	74104	74104	365580	4.93334
	2D_54019_highK	54019	54019	996414	18.4456
	Dubcova2	65025	65025	1030225	15.8435
	li	22695	22695	1350309	59.4981
	pdb1HYS	36417	36417	4344765	119.306
	tmt_unsym	917825	917825	4584801	4.99529
	Chebyshev4	68121	68121	5377761	78.9442
	kkt_power	2063494	2063494	14612663	7.08151
VLSI	ibm03	23136	27401	93573	3.41495
	ibm04	27507	31970	105859	3.3112
	ibm05	29347	28446	126308	4.44027
	ibm08	51309	50513	204890	4.05618
	ibm09	53395	60902	222088	3.64665
	ibm10	69429	75196	297567	3.95722
	ibm12	71076	77240	317760	4.11393
	ibm18	210613	201920	819697	4.05951
	ibm17	185495	189581	860036	4.53651
	superblue19	522482	511685	1713796	3.34932
	superblue16	698339	697458	2280417	3.26961
	superblue2	1010321	990899	3227167	3.25681
	superblue6	1011662	1006629	3387521	3.36521
	superblue12	1291931	1293436	4773600	3.69063
Primal	bob12s02	26294	77920	181812	2.33332
	6s16	31483	91888	214404	2.33332
	gss-19-s100	31435	94548	222806	2.35654
	MD5-28-4	8281	62544	243722	3.89681
	MD5-30-5	8905	68103	266105	3.90739
	ctl_3791_556_unsat_pre	8806	90812	331537	3.65081
	sat14_slp-synthesis-aes-top29	94998	302862	740744	2.44581
	atco_enc2_opt1_05_21	56533	526872	2097393	3.98084
	ACG-20-5p0	324716	1390931	3269132	2.35032
	UTI-20-10p1	260342	1391257	3358569	2.41405
	ACG-20-10p1	381708	1632906	3841867	2.35278
	atco_enc3_opt1_04_50	1613160	6429816	16042866	2.49507
Literal	countbitssrl032	37213	55724	130020	2.33329
	bob12s02	52588	77920	181812	2.33332
	6s11-opt	66552	97312	227060	2.33332
	MD5-30-4	17810	68106	266116	3.90738
	aaai10-planning-ipc5-pathways-17-step21	107838	308235	690466	2.24006
	slp-synthesis-aes-top29	189996	302862	740744	2.44581
	atco_enc2_opt1_05_21	112732	526872	2097393	3.98084
	dated-10-17-u	459088	1070757	2471122	2.30783
	9dlx_vliw_at_b_iq3	139578	968295	2788367	2.87967
	q_query_3_L80_coli.sat	501134	1183233	3415429	2.88652
	post-cbmc-aes-ee-r2-noholes	532170	1575975	4240496	2.69071
	transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.030-NOTKNOWN	705500	1934720	4605620	2.38051
	velev-vliw-uns-2.0-uq5	303338	2465731	7141423	2.89627
	q_query_3_L150_coli.sat	973984	2456708	7147094	2.90922
Dual	AProVE07-01	28770	7502	76290	10.1693
	gss-18-s100	94269	31364	222003	7.07827
	6s9	100384	34317	234228	6.82542
	MD5-30-4	68106	8905	266116	29.8839
	6s130-opt	144361	49327	336841	6.82873
	aaai10-planning-ipc5-pathways-17-step21	308235	53919	690466	12.8056
	hwmcc10-timeframe-expansion-k45-pdtvisns3p02-tseitin	488120	163622	1138944	6.96082
	minandmaxor128	746444	249327	1741700	6.98561
	manol-pipe-c10nid_i	750877	252516	1752045	6.93835
	atco_enc1_opt1_05_21	561784	59517	2167217	36.4134
	UCG-15-10p0	1005834	199304	2392967	12.0066
	UR-15-10p1	1019200	199996	2430990	12.1552
	UTI-20-10p1	1391257	260342	3358569	12.9006
	post-cbmc-aes-ee-r2-noholes	1575975	266199	4240496	15.9298
	hypergraph	$\|V\|$	$\|E\|$	|Pins|	avg $d(v)$

Table A.1: These 70 hypergraphs are used for benchmarking.

A.3 Training Sets

hypergraph	k	time $[s]$	avg $(\lambda-1)$	$\hat{\mathfrak{c}}^1_{\text{time}}[\%]$	$\hat{\mathfrak{c}}^1_{\lambda-1}[\%]$	$\hat{\mathfrak{c}}^8_{\text{time}}[\%]$	$\hat{\mathfrak{c}}^8_{\lambda-1}[\%]$
9dlx_vliw_at_b_iq3	4	106.3	84380.5	24.97	2.25	8.54	1.05
9dlx_vliw_at_b_iq3	8	160.6	144347.2	30.93	2.43	11.15	0.62
9dlx_vliw_at_b_iq3	16	195.0	192573.0	24.72	1.88	7.6	0.61
9dlx_vliw_at_b_iq3	32	211.3	232039.9	23.2	1.72	7.6	0.56
9dlx_vliw_at_b_iq3	64	224.0	284696.5	15.05	1.56	4.95	0.44
ACG-20-10p1	16	53.5	25884.1	1.04	1.42	0.78	0.45
ACG-20-10p1	32	63.2	34849.9	1.29	2.18	0.43	0.73
ACG-20-10p1	64	78.2	46004.6	1.41	2.62	0.79	0.82
ACG-20-5p0	32	54.6	31679.2	1.22	2.36	0.63	0.94
ACG-20-5p0	64	69.2	43307.0	1.41	2.53	1.25	0.87
ACG-20-5p0	128	91.5	63832.2	1.36	2.34	0.42	0.68
ACG-20-5p1	16	47.4	23836.4	1.8	1.58	1.06	0.51
ACG-20-5p1	32	55.4	31623.6	2.68	2.44	1.83	0.74
ACG-20-5p1	64	69.0	43341.6	1.2	2.42	0.73	0.77
ACG-20-5p1	128	92.4	64401.4	1.27	2.25	0.52	0.71
c10bi_i	32	42.9	48909.7	5.56	3.25	2.56	1.26
c10bi_i	64	56.8	68998.1	4.32	1.64	2.62	0.49
itox_vc1130	64	50.0	79468.8	8.4	2.06	3.26	0.74
manol-pipe-c10nid_i	32	80.1	67789.2	5.89	4.35	2.51	1.08
manol-pipe-c10nid_i	64	105.6	101249.0	4.8	2.03	2.27	0.65
manol-pipe-c10nid_i	128	139.3	146543.8	4.16	1.2	2.2	0.45
manol-pipe-c8nidw	32	91.5	69785.9	7.05	3.2	3.09	1.17
manol-pipe-c8nidw	64	112.0	104189.7	3.72	1.56	1.58	0.65
manol-pipe-c8nidw	128	149.1	156006.3	3.34	1.43	1.58	0.43
manol-pipe-g10bid_i	32	83.8	72075.5	4.6	3.73	2.09	0.89
manol-pipe-g10bid_i	64	113.0	108860.9	5.44	2.18	2.59	0.7
manol-pipe-g10bid_i	128	148.5	152571.4	4.65	1.4	2.13	0.45
minandmaxor128	8	41.4	41715.7	10.22	2.84	3.48	1.1
minandmaxor128	16	48.6	59677.9	5.78	1.7	2.53	0.63
minandmaxor128	32	61.4	74124.5	4.03	1.25	1.45	0.45
minandmaxor128	64	79.3	87225.8	3.42	1.4	1.53	0.49
minandmaxor128	128	102.5	106780.1	2.97	2.21	1.86	0.83
openstacks-...3.025-NOTKNOWN	8	57.8	24311.7	65.69	3.76	22.18	1.25
openstacks-...3.025-NOTKNOWN	32	121.3	82627.5	18.41	1.77	6.3	0.44
openstacks-...3.025-NOTKNOWN	64	109.1	121969.5	6.9	1.33	2.65	0.55
openstacks-...3.025-NOTKNOWN	128	119.9	145747.9	5.31	1.27	1.78	0.37
post-cbmc-aes-ee-r2-noholes	128	143.3	46756.4	2.01	3.96	0.86	1.09
slp-synthesis-aes-top29	32	48.2	45436.9	13.97	1.68	5.03	0.69
slp-synthesis-aes-top29	64	63.3	57081.0	8.48	1.51	3.47	0.53
UCG-20-5p0	16	45.9	27892.2	1.92	3.35	1.06	0.99
UR-20-5p0	16	47.8	32118.1	1.99	2.94	0.78	1.19
UR-20-5p0	32	57.1	40492.9	1.92	2.63	0.87	0.9
UR-20-5p0	64	73.8	55603.3	2.25	2.03	1.46	0.73
UTI-20-10p1	64	67.1	50078.6	1.88	2.15	1.48	0.56
Σ 44 instances		∅82.3		∅7.52	∅2.22	∅3.02	∅0.73

Table A.2: Our selected instances for the Literal training set. We report the average running times of the default configuration as well as the average connectivities of the partitions found by the default configuration. The randomization is determined using Algorithm 3 and is denoted in percent by $\hat{\mathfrak{c}}^1$ for $R_{min} = 1$ and $\hat{\mathfrak{c}}^8$ for $R_{min} = 8$. This is done for running time as cost function as well as for quality. We use the geometric mean to average numbers for all values of this table.

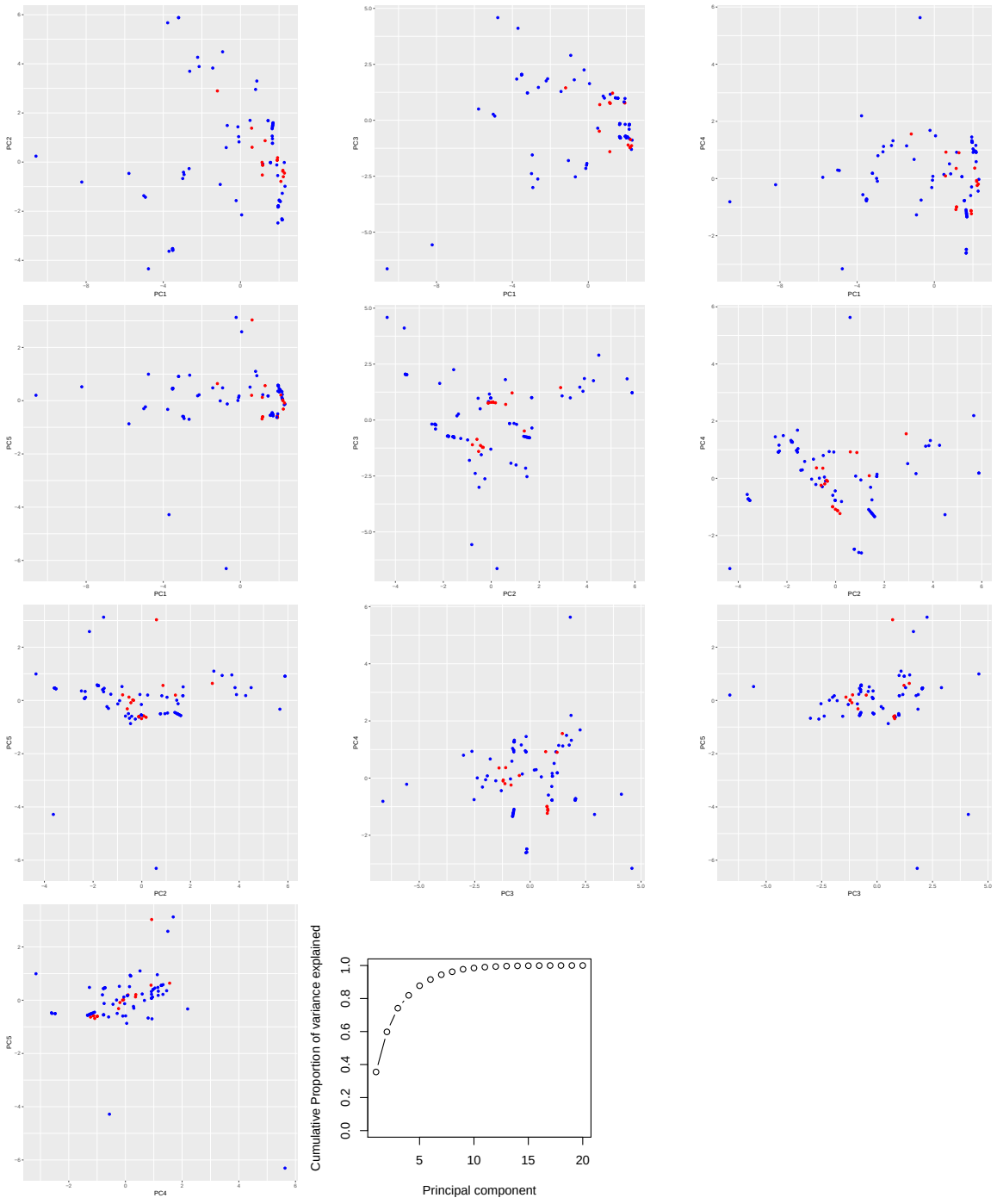


Figure A.1: The PCA plots for our selected literal hypergraphs. Red dots represent selected hypergraphs, all other literal hypergraphs are represented by blue dots.

hypergraph	k	time $[s]$	avg $(\lambda-1)$	$\hat{\mathfrak{c}}^1_{\text{time}}[\%]$	$\hat{\mathfrak{c}}^1_{\lambda-1}[\%]$	$\hat{\mathfrak{c}}^8_{\text{time}}[\%]$	$\hat{\mathfrak{c}}^8_{\lambda-1}[\%]$
as-caida	16	34.4	4063.9	13.99	2.83	3.61	1.01
as-caida	32	65.3	5808.4	8.3	1.58	3.71	0.59
as-caida	64	131.9	7646.2	6.98	1.2	2.93	0.48
as-caida	128	133.7	10050.5	5.59	1.01	2.45	0.31
av41092	16	52.9	7574.8	16.68	3.26	5.0	1.06
av41092	32	67.1	12688.3	18.54	2.59	4.77	0.82
av41092	64	71.6	18973.8	15.89	1.89	5.47	0.52
av41092	128	60.2	27036.6	7.65	1.37	2.67	0.49
bibd_49_3	16	26.0	5382.0	66.77	1.93	14.91	0.48
bibd_49_3	32	51.6	7098.2	62.44	2.09	19.05	0.67
bibd_49_3	64	29.5	9118.2	8.22	0.92	4.57	0.32
bloweya	2	27.1	4999.7	29.96	2.64	10.06	0.52
c-64	16	112.5	24448.0	54.59	4.09	16.39	1.25
c-64	32	173.2	31234.1	57.31	3.14	16.37	1.34
cnr-2000	32	98.2	14685.9	12.47	3.48	5.79	1.15
H2O	8	121.9	47852.3	9.21	1.74	4.06	0.71
H2O	16	195.6	75439.1	13.55	1.32	5.23	0.41
H2O	32	308.0	112899.7	11.04	1.33	4.3	0.5
H2O	64	412.1	164474.7	7.32	0.72	2.45	0.29
HTC_336_9129	8	53.1	9361.7	16.81	3.33	6.64	1.33
HTC_336_9129	16	77.2	12067.8	20.23	2.29	7.05	0.78
HTC_336_9129	32	110.6	14348.6	22.91	1.87	5.79	0.47
HTC_336_9129	64	158.4	16947.5	36.73	1.98	11.36	0.48
HTC_336_9129	128	184.5	21303.5	32.78	1.1	12.9	0.37
language	2	115.3	13552.5	20.44	1.48	6.59	0.36
language	4	338.5	28518.9	15.65	1.91	5.36	0.65
mono_500Hz	8	43.2	25262.0	5.72	2.59	2.33	0.83
mono_500Hz	16	71.9	38092.5	6.59	1.83	1.84	0.53
mono_500Hz	32	115.2	55839.1	8.88	1.1	3.13	0.4
mono_500Hz	64	171.6	79787.8	7.19	0.88	2.74	0.25
nd12k	2	69.7	10533.3	5.72	1.68	2.34	0.78
nd12k	4	105.1	22472.1	9.85	1.41	4.24	0.65
nd12k	8	143.4	38844.6	10.33	2.26	3.12	0.71
nd12k	16	182.5	62339.4	5.92	1.41	2.39	0.6
pds-90	32	101.3	17616.9	11.12	3.3	3.22	1.26
pds-90	64	137.7	24674.6	7.97	2.02	3.73	0.53
pds-90	128	170.8	32349.6	5.83	1.14	2.32	0.37
pre2	64	155.5	75732.5	3.02	1.88	1.39	0.67
pre2	128	222.0	102345.6	2.55	1.14	1.09	0.41
sparsine	4	91.9	36807.5	23.97	2.43	8.3	1.1
sparsine	8	166.0	57666.5	34.9	2.34	11.1	0.83
sparsine	16	322.0	86957.6	23.86	2.6	8.97	0.94
StocF-1465	2	74.9	4728.2	0.56	2.17	0.33	1.01
StocF-1465	4	81.2	10250.0	0.69	2.15	0.42	0.83
StocF-1465	8	107.9	45935.0	1.02	1.49	0.71	0.61
StocF-1465	16	146.2	92115.7	1.65	2.02	1.08	0.87
StocF-1465	32	197.4	146332.8	1.59	1.28	0.88	0.39
us04	16	25.3	1015.9	12.71	2.52	4.05	0.83
us04	32	44.6	1828.6	27.32	2.98	7.21	0.98
Σ 49 instances		∅101.4		∅15.6	∅1.99	∅5.35	∅0.69

Table A.3: Our selected instances for the SPM training set. We report the average running times of the default configuration as well as the average connectivities of the partitions found by the default configuration. The randomization is determined using Algorithm 3 and is denoted in percent by $\hat{\mathfrak{c}}^1$ for $R_{min} = 1$ and $\hat{\mathfrak{c}}^8$ for $R_{min} = 8$. This is done for running time as cost function as well as for quality. We use the geometric mean to average numbers for all values of this table.

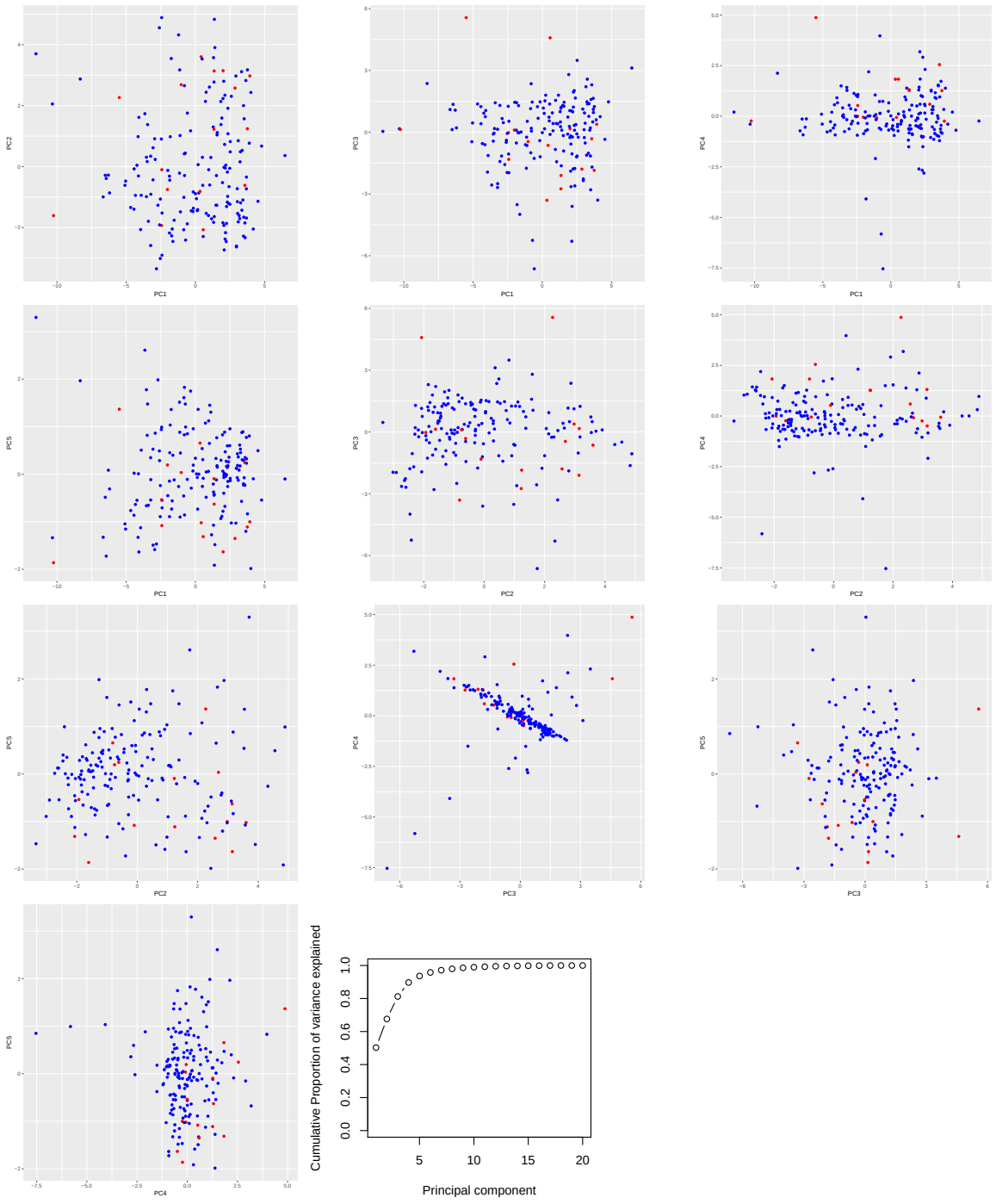


Figure A.2: The PCA plots for our selected SPM hypergraphs. Red dots represent selected hypergraphs, all other SPM hypergraphs are represented by blue dots.

hypergraph	k	time $[s]$	avg $(\lambda-1)$	$\hat{\mathfrak{c}}^1_{\text{time}}[\%]$	$\hat{\mathfrak{c}}^1_{\lambda-1}[\%]$	$\hat{\mathfrak{c}}^8_{\text{time}}[\%]$	$\hat{\mathfrak{c}}^8_{\lambda-1}[\%]$
superblue16	8	133.4	17504.3	10.11	4.63	2.78	1.52
superblue6	8	169.3	14037.3	8.36	3.06	3.11	1.42
ibm12	4	5.0	4181.0	6.4	5.33	2.48	1.26
ibm12	8	7.8	6651.3	7.51	2.6	4.81	1.18
ibm16	4	11.7	4462.7	3.35	2.77	1.06	1.1
ibm16	16	25.7	12192.8	3.92	3.45	2.75	0.86
ibm16	32	37.8	18490.4	4.62	2.02	3.69	0.77
ibm17	2	8.5	2391.8	2.4	3.53	0.84	1.71
ibm17	4	13.5	6172.7	3.09	4.46	1.58	1.62
ibm17	8	20.9	11044.6	2.88	4.53	1.25	1.42
ibm17	16	30.9	17008.2	4.58	3.48	2.74	1.28
ibm17	32	44.5	23385.2	5.02	2.07	2.74	0.51
ibm18	2	9.7	1972.6	4.75	2.74	1.7	1.83
ibm18	4	13.6	3280.4	3.38	2.78	2.56	1.22
ibm18	8	22.1	6269.2	7.07	2.9	3.61	0.7
ibm18	16	33.6	10221.9	7.42	2.23	3.1	0.77
ibm18	32	52.2	15648.6	10.01	1.68	4.6	0.49
ibm18	64	78.0	23610.5	9.6	1.18	5.33	0.36
Σ 18 instances		$\varnothing$24.9		$\varnothing$5.77	$\varnothing$3.07	$\varnothing$2.81	$\varnothing$1.11

Table A.4: Our selected instances for the VLSI training set. We report the average running times of the default configuration as well as the average connectivities of the partitions found by the default configuration. The randomization is determined using Algorithm 3 and is denoted in percent by $\hat{\mathfrak{c}}^1$ for $R_{min} = 1$ and $\hat{\mathfrak{c}}^8$ for $R_{min} = 8$. This is done for running time as cost function as well as for quality. We use the geometric mean to average numbers for all values of this table.

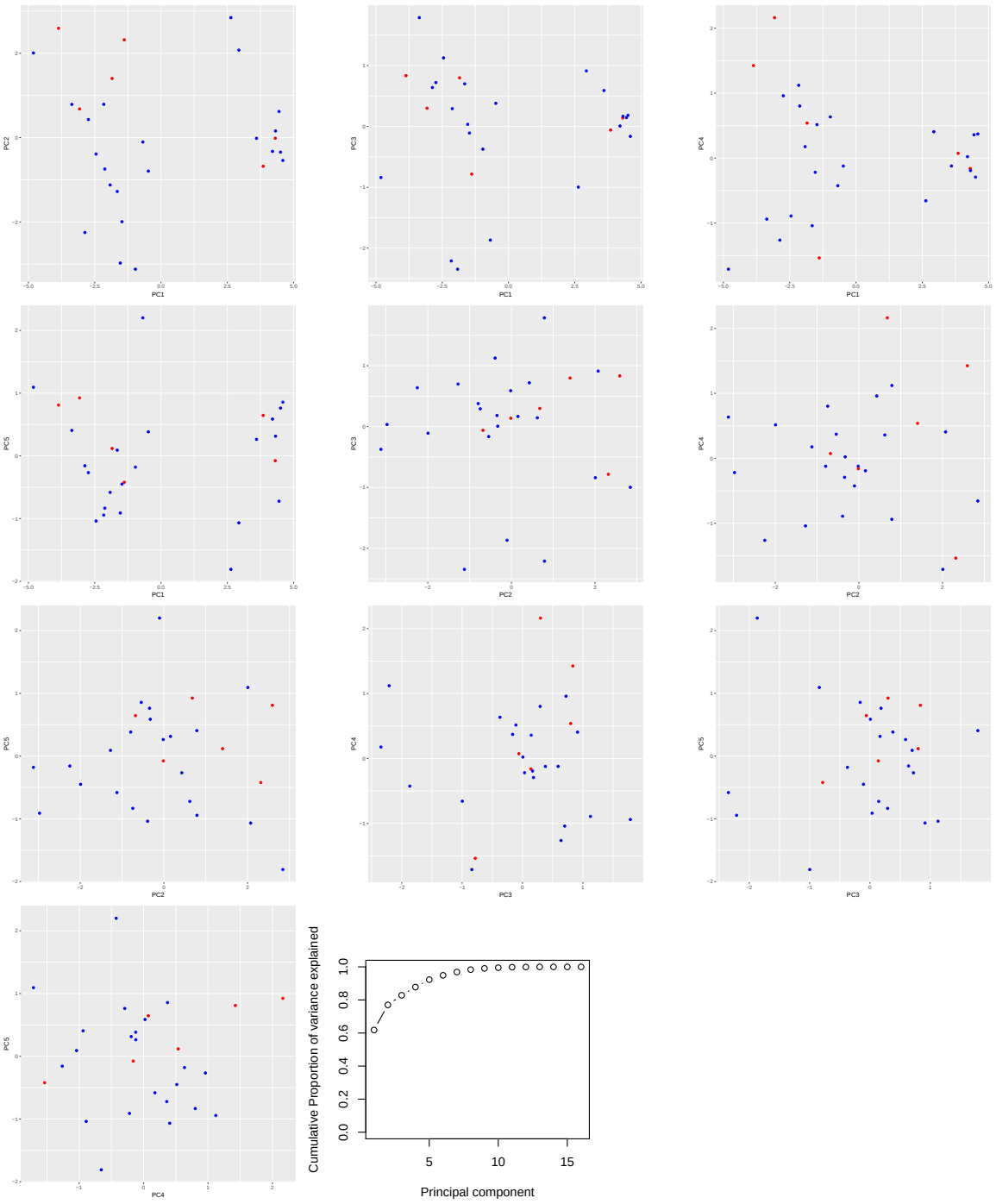


Figure A.3: The PCA plots for our selected VLSI hypergraphs. Red dots represent selected hypergraphs, all other VLSI hypergraphs are represented by blue dots. Four features are identical for all hypergraphs and are therefore removed for this instance set.

hypergraph	k	time $[s]$	avg $(\lambda-1)$	$\hat{\mathfrak{c}}^1_{\text{time}}[\%]$	$\hat{\mathfrak{c}}^1_{\lambda-1}[\%]$	$\hat{\mathfrak{c}}^8_{\text{time}}[\%]$	$\hat{\mathfrak{c}}^8_{\lambda-1}[\%]$
sat14_ACG-20-5p1	8	48.3	5353.3	8.32	3.84	3.02	1.43
sat14_ACG-20-5p1	16	70.5	8169.1	4.69	1.2	1.58	0.37
sat14_ACG-20-5p1	32	91.5	10978.4	3.58	2.36	1.46	0.61
sat14_ACG-20-5p1	64	117.5	14901.2	2.6	2.22	1.17	0.73
sat14_ACG-20-5p1	128	155.8	21327.3	2.58	1.43	1.69	0.56
sat14_c10bi_i	4	32.8	1480.9	27.39	4.09	8.99	1.1
sat14_c10bi_i	8	45.1	2696.8	12.75	2.64	4.86	1.06
sat14_c10bi_i	16	74.1	4478.4	9.48	3.42	3.83	1.16
sat14_c10bi_i	32	126.9	6877.4	13.69	2.49	5.19	0.74
sat14_c10bi_i	64	184.6	9910.4	10.94	1.47	4.23	0.64
sat14_c10bi_i	128	221.0	14420.3	6.29	1.19	2.1	0.42
sat14_itox_vc1130	16	59.1	1534.0	8.26	3.74	2.89	1.16
sat14_itox_vc1130	32	89.7	2553.9	8.64	2.59	2.62	0.96
sat14_itox_vc1130	64	139.8	4200.1	8.52	1.58	2.75	0.64
sat14_itox_vc1130	128	206.1	6465.9	8.25	1.38	2.41	0.37
sat14_manol-pipe-c10nid_i	4	79.8	1854.9	51.15	2.17	14.23	1.01
sat14_manol-pipe-c10nid_i	8	91.9	3755.3	17.5	4.24	7.03	1.31
sat14_manol-pipe-c10nid_i	16	146.3	6757.9	18.58	2.87	6.5	1.0
sat14_manol-pipe-c10nid_i	32	225.2	10267.8	15.37	2.59	5.34	0.87
sat14_minandmaxor128	2	48.5	840.3	12.69	3.18	3.26	0.92
sat14_minandmaxor128	16	156.2	3969.6	7.91	4.5	2.97	1.32
sat14_minandmaxor128	32	207.7	5578.3	5.67	2.77	2.31	0.79
sat14_minandmaxor128	64	266.1	7717.8	5.49	1.98	1.59	0.8
sat14_minandmaxor128	128	330.8	10705.0	4.53	1.52	1.49	0.63
sat14_openstacks-p30_3.085-SAT	16	170.6	8846.2	4.16	3.66	1.55	0.92
sat14_SAT_dat.k70-24_1_rule_1	4	108.1	1774.2	2.53	1.27	1.04	0.68
sat14_SAT_dat.k70-24_1_rule_1	8	141.2	4171.4	2.3	1.23	0.74	0.43
sat14_SAT_dat.k70-24_1_rule_1	16	208.5	8827.8	2.36	3.8	1.11	0.93
sat14_SAT_dat.k80-24_1_rule_1	4	123.1	1753.0	2.14	0.66	0.67	0.37
sat14_SAT_dat.k80-24_1_rule_1	8	154.3	4107.4	2.3	0.97	0.57	0.42
sat14_SAT_dat.k80-24_1_rule_1	16	221.5	8841.5	2.29	1.26	0.7	0.48
sat14_slp-synthesis-aes-top29	4	33.7	2917.0	11.72	4.58	5.3	1.45
sat14_slp-synthesis-aes-top29	8	57.2	4741.6	11.25	3.53	4.15	1.24
sat14_slp-synthesis-aes-top29	16	93.4	7023.4	15.81	2.62	5.07	0.74
sat14_slp-synthesis-aes-top29	32	146.3	9074.1	14.63	2.34	4.46	0.83
sat14_slp-synthesis-aes-top29	64	252.2	11708.7	12.62	1.92	4.57	0.86
sat14_UCG-20-5p0	8	43.3	5051.6	6.5	3.54	1.98	1.3
sat14_UCG-20-5p0	16	66.7	8125.3	3.82	1.02	1.22	0.39
sat14_UCG-20-5p0	32	87.7	11373.4	3.1	1.68	1.45	0.51
sat14_UCG-20-5p0	64	116.1	16341.1	2.64	2.1	1.2	0.62
sat14_UCG-20-5p0	128	162.3	23359.4	2.03	1.25	1.13	0.4
sat14_UR-15-10p1	8	50.6	5571.2	3.86	2.43	1.71	0.98
sat14_UR-15-10p1	16	65.4	7555.8	3.1	1.45	1.29	0.55
sat14_UR-15-10p1	32	81.9	10300.3	3.08	2.24	1.03	0.94
sat14_UR-15-10p1	64	106.2	14985.4	2.61	2.25	1.13	0.57
sat14_UR-15-10p1	128	146.9	21350.8	2.1	0.99	1.58	0.37
sat14_UR-20-5p0	4	50.6	4711.0	8.11	4.81	3.25	1.48
sat14_UR-20-5p0	8	62.3	6332.4	5.24	2.24	1.65	1.05
sat14_UR-20-5p0	16	78.1	8345.5	3.79	1.53	1.12	0.57
sat14_UR-20-5p0	32	96.7	10949.7	3.54	2.38	1.46	0.8
sat14_UR-20-5p0	64	122.1	15363.3	2.57	2.75	1.08	0.99
sat14_UR-20-5p0	128	164.3	22431.7	1.93	1.25	1.07	0.37
Σ 52 instances		∅106.3		∅7.75	∅2.36	∅2.79	∅0.8

Table A.5: Our selected instances for the Dual training set. We report the average running times of the default configuration as well as the average connectivities of the partitions found by the default configuration. The randomization is determined using Algorithm 3 and is denoted in percent by $\hat{\mathfrak{c}}^1$ for $R_{min} = 1$ and $\hat{\mathfrak{c}}^8$ for $R_{min} = 8$. This is done for running time as cost function as well as for quality. We use the geometric mean to average numbers for all values of this table.

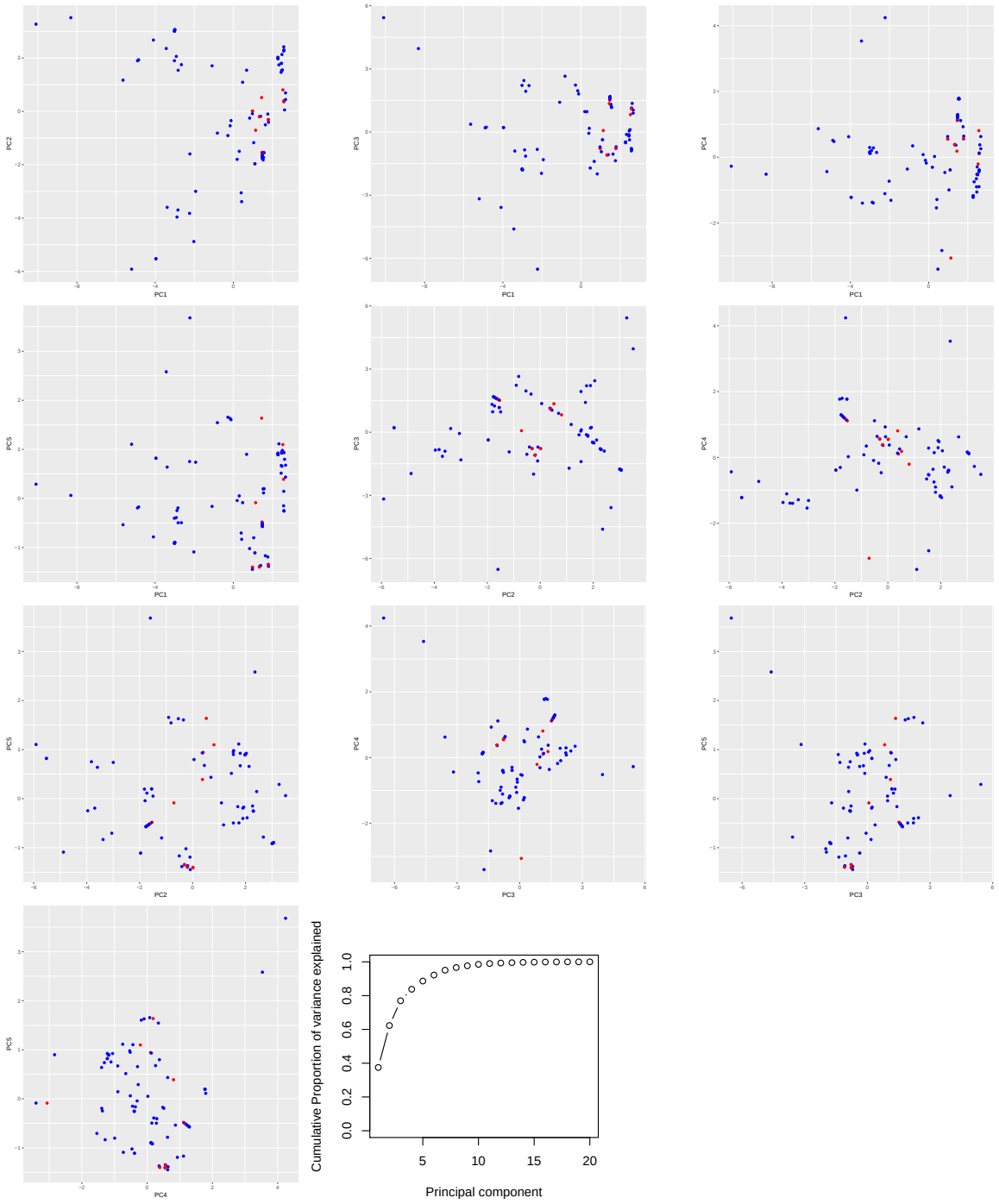


Figure A.4: The PCA plots for our selected dual hypergraphs. Red dots represent selected hypergraphs, all other dual graphs are represented by blue dots.

hypergraph	k	time $[s]$	avg $(\lambda-1)$	$\hat{\mathfrak{c}}^1_{\text{time}}[\%]$	$\hat{\mathfrak{c}}^1_{\lambda-1}[\%]$	$\hat{\mathfrak{c}}^8_{\text{time}}[\%]$	$\hat{\mathfrak{c}}^8_{\lambda-1}[\%]$
sat14_10pipe_q0_k	8	205.8	468346.5	53.14	3.17	18.86	1.38
sat14_10pipe_q0_k	16	229.6	707716.5	16.02	1.48	4.88	0.48
sat14_9dlx_vliw_at_b_iq3	4	96.1	129364.5	33.34	3.39	9.35	1.33
sat14_9dlx_vliw_at_b_iq3	8	127.1	189082.3	37.42	2.1	9.72	0.61
sat14_9dlx_vliw_at_b_iq3	16	156.4	244050.6	20.04	2.42	5.67	0.53
sat14_9dlx_vliw_at_b_iq3	32	167.6	295464.6	15.09	1.54	5.23	0.64
sat14_ACG-20-10p1	32	47.9	40365.8	2.36	2.61	1.53	0.9
sat14_ACG-20-10p1	64	59.2	55457.6	2.32	3.05	1.17	0.89
sat14_ACG-20-10p1	128	76.5	82055.2	1.43	2.65	0.73	0.93
sat14_ACG-20-5p0	128	67.5	80409.0	2.08	2.41	1.37	0.68
sat14_manol-pipe-c10nidw	32	75.8	104923.7	5.02	2.41	2.3	0.82
sat14_manol-pipe-c10nidw	64	97.7	164003.0	4.52	2.03	2.24	0.62
sat14_manol-pipe-g10bid_i	32	44.0	70003.0	3.34	2.48	1.71	1.01
sat14_minandmaxor128	32	39.4	71757.4	5.38	1.06	2.02	0.29
sat14_minandmaxor128	64	48.1	87349.7	5.16	1.04	1.83	0.28
sat14_minandmaxor128	128	61.3	106363.6	3.06	1.24	1.26	0.51
sat14_openstacks-p30_3.085-SAT	16	75.2	74461.2	6.87	2.38	2.67	0.94
sat14_openstacks-p30_3.085-SAT	32	106.0	159307.0	4.01	1.41	1.72	0.51
sat14_openstacks-p30_3.085-SAT	64	161.7	317836.8	4.09	0.78	1.46	0.31
sat14_openstacks-p30_3.085-SAT	128	332.0	495191.7	5.67	1.14	1.9	0.39
sat14_openstacks-...3.025-NOTKNOWN	32	59.4	131407.1	9.61	1.59	2.95	0.57
sat14_openstacks-...3.025-NOTKNOWN	64	91.7	180360.9	8.84	0.82	3.63	0.28
sat14_openstacks-...3.085-SAT	16	74.8	74453.3	6.09	2.51	2.18	1.1
sat14_openstacks-...3.085-SAT	32	106.0	159213.9	4.19	1.42	1.91	0.49
sat14_openstacks-...3.085-SAT	64	161.7	317718.7	4.83	0.76	1.75	0.34
sat14_q_query_3_L80_coli.sat	16	218.7	361910.7	13.59	3.41	3.16	0.86
sat14_q_query_3_L80_coli.sat	32	265.5	504127.3	19.13	1.15	5.73	0.42
sat14_SAT_dat.k75-24_1_rule_3	8	102.9	9101.2	1.12	2.94	0.45	1.2
sat14_SAT_dat.k75-24_1_rule_3	32	148.6	40806.2	2.62	1.88	0.84	0.53
sat14_SAT_dat.k75-24_1_rule_3	64	179.9	79026.7	2.68	2.15	1.26	0.82
sat14_SAT_dat.k80-24_1_rule_1	4	100.0	3519.7	0.91	0.81	0.4	0.53
sat14_SAT_dat.k85-24_1_rule_2	2	100.9	1167.5	1.01	0.31	0.48	0.26
sat14_SAT_dat.k90.debugged	16	142.4	19641.4	1.55	4.43	0.8	1.4
sat14_velev-vliw-uns-4.0-9	2	302.3	77051.0	37.7	2.2	11.76	0.74
sat14_velev-vliw-uns-4.0-9	4	261.1	299981.3	48.12	3.75	14.37	1.25
sat14_velev-vliw-uns-4.0-9	8	362.2	456443.7	51.65	2.31	15.92	0.76
Σ 36 instances		∅115.2		∅11.44	∅2.03	∅3.94	∅0.71

Table A.6: Our selected instances for the Primal training set. We report the average running times of the default configuration as well as the average connectivities of the partitions found by the default configuration. The randomization is determined using Algorithm 3 and is denoted in percent by $\hat{\mathfrak{c}}^1$ for $R_{min} = 1$ and $\hat{\mathfrak{c}}^8$ for $R_{min} = 8$. This is done for running time as cost function as well as for quality. We use the geometric mean to average numbers for all values of this table.

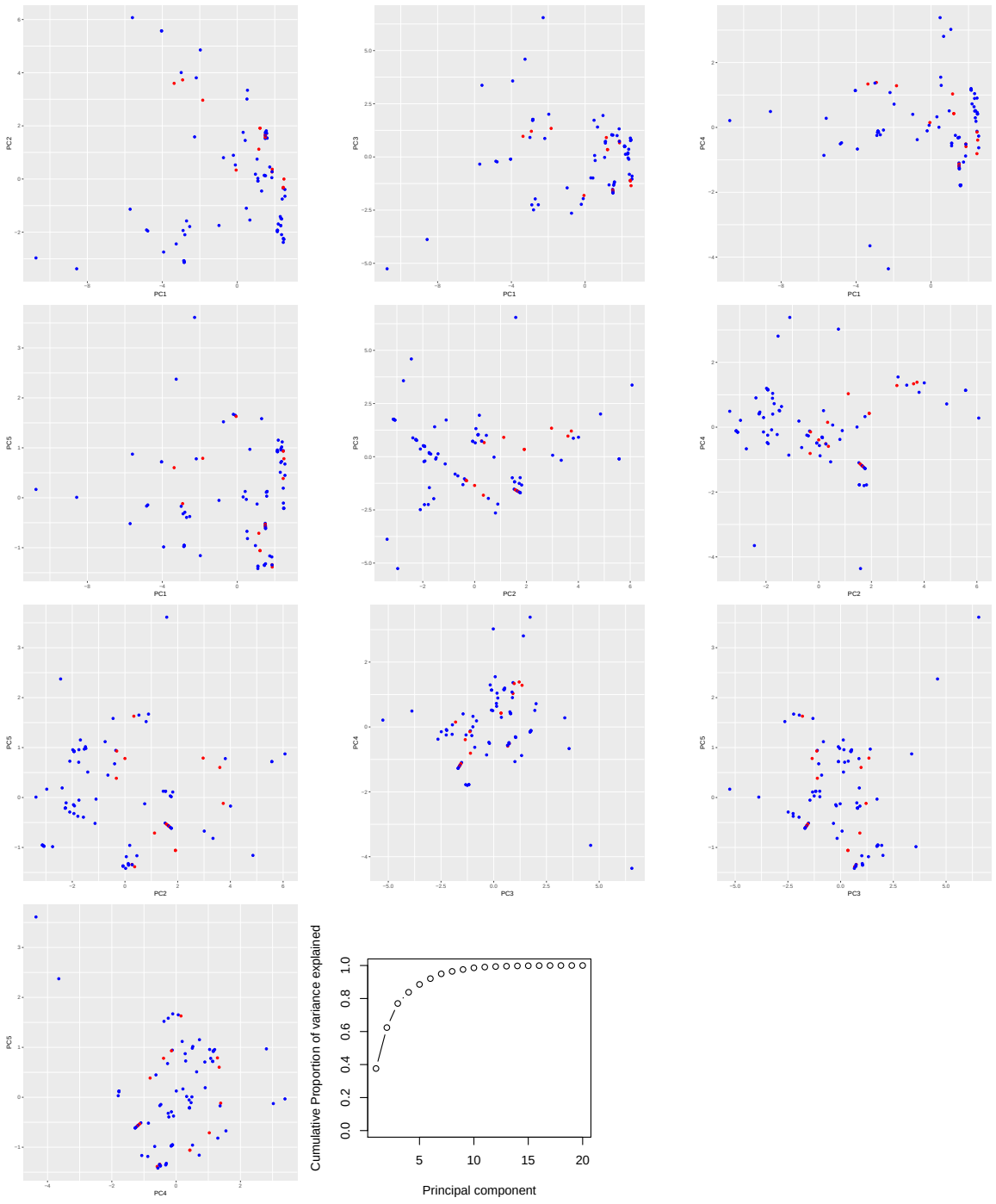


Figure A.5: The PCA plots for our selected primal hypergraphs. Red dots represent selected hypergraphs, all other primal hypergraphs are represented by blue dots.

type	hypergraph	k	time [s]	avg $(\lambda-1)$	$\hat{\epsilon}^1_{\text{time}}$[%]	$\hat{\epsilon}^1_{\lambda-1}$[%]	$\hat{\epsilon}^8_{\text{time}}$[%]	$\hat{\epsilon}^8_{\lambda-1}$[%]
Dual	ACG-20-5p1	8	48.3	5353.3	1.0832	1.0384	1.0302	1.0143
	ACG-20-5p1	16	70.5	8169.1	1.0469	1.012	1.0158	1.0037
	ACG-20-5p1	32	91.5	10978.4	1.0358	1.0236	1.0146	1.0061
	ACG-20-5p1	64	117.5	14901.2	1.026	1.0222	1.0117	1.0073
	ACG-20-5p1	128	155.8	21327.3	1.0258	1.0143	1.0169	1.0056
	UR-20-5p0	4	50.6	4711.0	1.0811	1.0481	1.0325	1.0148
	UR-20-5p0	8	62.3	6332.4	1.0524	1.0224	1.0165	1.0105
	UR-20-5p0	16	78.1	8345.5	1.0379	1.0153	1.0112	1.0057
	UR-20-5p0	32	96.7	10949.7	1.0354	1.0238	1.0146	1.008
	UR-20-5p0	64	122.1	15363.3	1.0257	1.0275	1.0108	1.0099
	itox_vc1130	16	59.1	1534.0	1.0826	1.0374	1.0289	1.0116
	itox_vc1130	32	89.7	2553.9	1.0864	1.0259	1.0262	1.0096
	itox_vc1130	64	139.8	4200.1	1.0852	1.0158	1.0275	1.0064
	itox_vc1130	128	206.1	6465.9	1.0825	1.0138	1.0241	1.0037
	ACG-20-5p1	8	48.3	5353.3	1.0832	1.0384	1.0302	1.0143
	ACG-20-5p1	16	70.5	8169.1	1.0469	1.012	1.0158	1.0037
	ACG-20-5p1	32	91.5	10978.4	1.0358	1.0236	1.0146	1.0061
	ACG-20-5p1	64	117.5	14901.2	1.026	1.0222	1.0117	1.0073
	ACG-20-5p1	128	155.8	21327.3	1.0258	1.0143	1.0169	1.0056
	UCG-20-5p0	8	43.3	5051.6	1.065	1.0354	1.0198	1.013
	UCG-20-5p0	16	66.7	8125.3	1.0382	1.0102	1.0122	1.0039
	UCG-20-5p0	32	87.7	11373.4	1.031	1.0168	1.0145	1.0051
	UCG-20-5p0	64	116.1	16341.1	1.0264	1.021	1.012	1.0062
	UCG-20-5p0	128	162.3	23359.4	1.0203	1.0125	1.0113	1.004
	SAT_dat.k70-24_1_rule_1	4	108.1	1774.2	1.0253	1.0127	1.0104	1.0068
	SAT_dat.k70-24_1_rule_1	8	141.2	4171.4	1.023	1.0123	1.0074	1.0043
	SAT_dat.k70-24_1_rule_1	16	208.5	8827.8	1.0236	1.038	1.0111	1.0093
	SAT_dat.k80-24_1_rule_1	4	123.1	1753.0	1.0214	1.0066	1.0067	1.0037
	SAT_dat.k80-24_1_rule_1	8	154.3	4107.4	1.023	1.0097	1.0057	1.0042
	SAT_dat.k80-24_1_rule_1	16	221.5	8841.5	1.0229	1.0126	1.007	1.0048
Literal	9dlx_vliw_at_b_iq3	16	195.0	192573.0	1.2472	1.0188	1.076	1.0061
	9dlx_vliw_at_b_iq3	4	106.3	84380.5	1.2497	1.0225	1.0854	1.0105
	9dlx_vliw_at_b_iq3	8	160.6	144347.2	1.3093	1.0243	1.1115	1.0062
	9dlx_vliw_at_b_iq3	32	211.3	232039.9	1.232	1.0172	1.076	1.0056
	9dlx_vliw_at_b_iq3	64	224.0	284696.5	1.1505	1.0156	1.0495	1.0044
	ACG-20-5p1	16	47.4	23836.4	1.018	1.0158	1.0106	1.0051
	ACG-20-5p1	32	55.4	31623.6	1.0268	1.0244	1.0183	1.0074
	ACG-20-5p1	64	69.0	43341.6	1.012	1.0242	1.0073	1.0077
	ACG-20-5p1	128	92.4	64401.4	1.0127	1.0225	1.0052	1.0071
	UCG-20-5p0	16	45.9	27892.2	1.0192	1.0335	1.0106	1.0099
	openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.025-NOTKNOWN	8	57.8	24311.7	1.6569	1.0376	1.2218	1.0125
	openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.025-NOTKNOWN	32	121.3	82627.5	1.1841	1.0177	1.063	1.0044
	openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.025-NOTKNOWN	64	109.1	121969.5	1.069	1.0133	1.0265	1.0055
	openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.025-NOTKNOWN	128	119.9	145747.9	1.0531	1.0127	1.0178	1.0037
	minandmaxor128	8	41.4	41715.7	1.1022	1.0284	1.0348	1.011
	minandmaxor128	16	48.6	59677.9	1.0578	1.017	1.0253	1.0063
	minandmaxor128	32	61.4	74124.5	1.0403	1.0125	1.0145	1.0045
	minandmaxor128	64	79.3	87225.8	1.0342	1.014	1.0153	1.0049
	minandmaxor128	128	102.5	106780.1	1.0297	1.0221	1.0186	1.0083
Primal	SAT_dat.k75-24_1_rule_3	8	102.9	9101.2	1.0112	1.0294	1.0045	1.012
	SAT_dat.k75-24_1_rule_3	32	148.6	40806.2	1.0262	1.0188	1.0084	1.0053
	SAT_dat.k75-24_1_rule_3	64	179.9	79026.7	1.0268	1.0215	1.0126	1.0082
	9dlx_vliw_at_b_iq3	4	96.1	129364.5	1.3334	1.0339	1.0935	1.0133
	9dlx_vliw_at_b_iq3	8	127.1	189082.3	1.3742	1.021	1.0972	1.0061
	9dlx_vliw_at_b_iq3	16	156.4	244050.6	1.2004	1.0242	1.0567	1.0053
	9dlx_vliw_at_b_iq3	32	167.6	295464.6	1.1509	1.0154	1.0523	1.0064
	openstacks-p30_3.085-SAT	16	75.2	74461.2	1.0687	1.0238	1.0267	1.0094
	openstacks-p30_3.085-SAT	32	106.0	159307.0	1.0401	1.0141	1.0172	1.0051
	openstacks-p30_3.085-SAT	64	161.7	317836.8	1.0409	1.0078	1.0146	1.0031
	ACG-20-10p1	64	59.2	55457.6	1.0232	1.0305	1.0117	1.0089
	minandmaxor128	32	39.4	71757.4	1.0538	1.0106	1.0202	1.0029
	minandmaxor128	64	48.1	87349.7	1.0516	1.0104	1.0183	1.0028
	minandmaxor128	128	61.3	106363.6	1.0306	1.0124	1.0126	1.0051
	openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.025-NOTKNOWN	32	59.4	131407.1	1.0961	1.0159	1.0295	1.0057
	openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.025-NOTKNOWN	64	91.7	180360.9	1.0884	1.0082	1.0363	1.0028

Primal	openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.085-SAT	16	74.8	74453.3	1.0609	1.0251	1.0218	1.011
	openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.085-SAT	32	106.0	159213.9	1.0419	1.0142	1.0191	1.0049
	openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.085-SAT	64	161.7	317718.7	1.0483	1.0076	1.0175	1.0034
VLSI	ibm12	4	5.0	4181.0	1.064	1.0533	1.0248	1.0126
	ibm12	8	7.8	6651.3	1.0751	1.026	1.0481	1.0118
	ibm16	4	11.7	4462.7	1.0335	1.0277	1.0106	1.011
	ibm16	16	25.7	12192.8	1.0392	1.0345	1.0275	1.0086
	ibm16	32	37.8	18490.4	1.0462	1.0202	1.0369	1.0077
	ibm17	2	8.5	2391.8	1.024	1.0353	1.0084	1.0171
	ibm17	4	13.5	6172.7	1.0309	1.0446	1.0158	1.0162
	ibm17	8	20.9	11044.6	1.0288	1.0453	1.0125	1.0142
	ibm17	16	30.9	17008.2	1.0458	1.0348	1.0274	1.0128
	ibm17	32	44.5	23385.2	1.0502	1.0207	1.0274	1.0051
	ibm18	2	9.7	1972.6	1.0475	1.0274	1.017	1.0183
	ibm18	4	13.6	3280.4	1.0338	1.0278	1.0256	1.0122
	ibm18	8	22.1	6269.2	1.0707	1.029	1.0361	1.007
	ibm18	16	33.6	10221.9	1.0742	1.0223	1.031	1.0077
	ibm18	32	52.2	15648.6	1.1001	1.0168	1.046	1.0049
	ibm18	64	78.0	23610.5	1.096	1.0118	1.0533	1.0036
	superblue6	8	169.3	14037.3	1.0836	1.0306	1.0311	1.0142
	superblue16	8	133.4	17504.3	1.1011	1.0463	1.0278	1.0152
SPM	HTC_336_9129	8	53.1	9361.7	1.1681	1.0333	1.0664	1.0133
	HTC_336_9129	16	77.2	12067.8	1.2023	1.0229	1.0705	1.0078
	HTC_336_9129	32	110.6	14348.6	1.2291	1.0187	1.0579	1.0047
	HTC_336_9129	64	158.4	16947.5	1.3673	1.0198	1.1136	1.0048
	HTC_336_9129	128	184.5	21303.5	1.3278	1.011	1.129	1.0037
	StocF-1465	2	74.9	4728.2	1.0056	1.0217	1.0033	1.0101
	StocF-1465	4	81.2	10250.0	1.0069	1.0215	1.0042	1.0083
	StocF-1465	8	107.9	45935.0	1.0102	1.0149	1.0071	1.0061
	StocF-1465	16	146.2	92115.7	1.0165	1.0202	1.0108	1.0087
	StocF-1465	32	197.4	146332.8	1.0159	1.0128	1.0088	1.0039
	av41092	16	52.9	7574.8	1.1668	1.0326	1.05	1.0106
	av41092	32	67.1	12688.3	1.1854	1.0259	1.0477	1.0082
	av41092	64	71.6	18973.8	1.1589	1.0189	1.0547	1.0052
	av41092	128	60.2	27036.6	1.0765	1.0137	1.0267	1.0049
	mono_500Hz	8	43.2	25262.0	1.0572	1.0259	1.0233	1.0083
	mono_500Hz	16	71.9	38092.5	1.0659	1.0183	1.0184	1.0053
	mono_500Hz	32	115.2	55839.1	1.0888	1.011	1.0313	1.004
	mono_500Hz	64	171.6	79787.8	1.0719	1.0088	1.0274	1.0025
	language	2	115.3	13552.5	1.2044	1.0148	1.0659	1.0036
	pre2	64	155.5	75732.5	1.0302	1.0188	1.0139	1.0067
	pre2	128	222.0	102345.6	1.0255	1.0114	1.0109	1.0041
	Σ 107 instances	∅77		∅1.0814	∅1.0217	∅1.0304	∅1.0075	

Table A.7: Our selected instances for the full training set. We report the average running times of the default configuration as well as the average connectivities of the partitions found by the default configuration. The randomization is determined using Algorithm 3 and is denoted in percent by $\hat{\mathfrak{c}}^1$ for $R_{min} = 1$ and $\hat{\mathfrak{c}}^8$ for $R_{min} = 8$. This is done for running time as cost function as well as for quality. We use the geometric mean to average numbers for all values of this table.

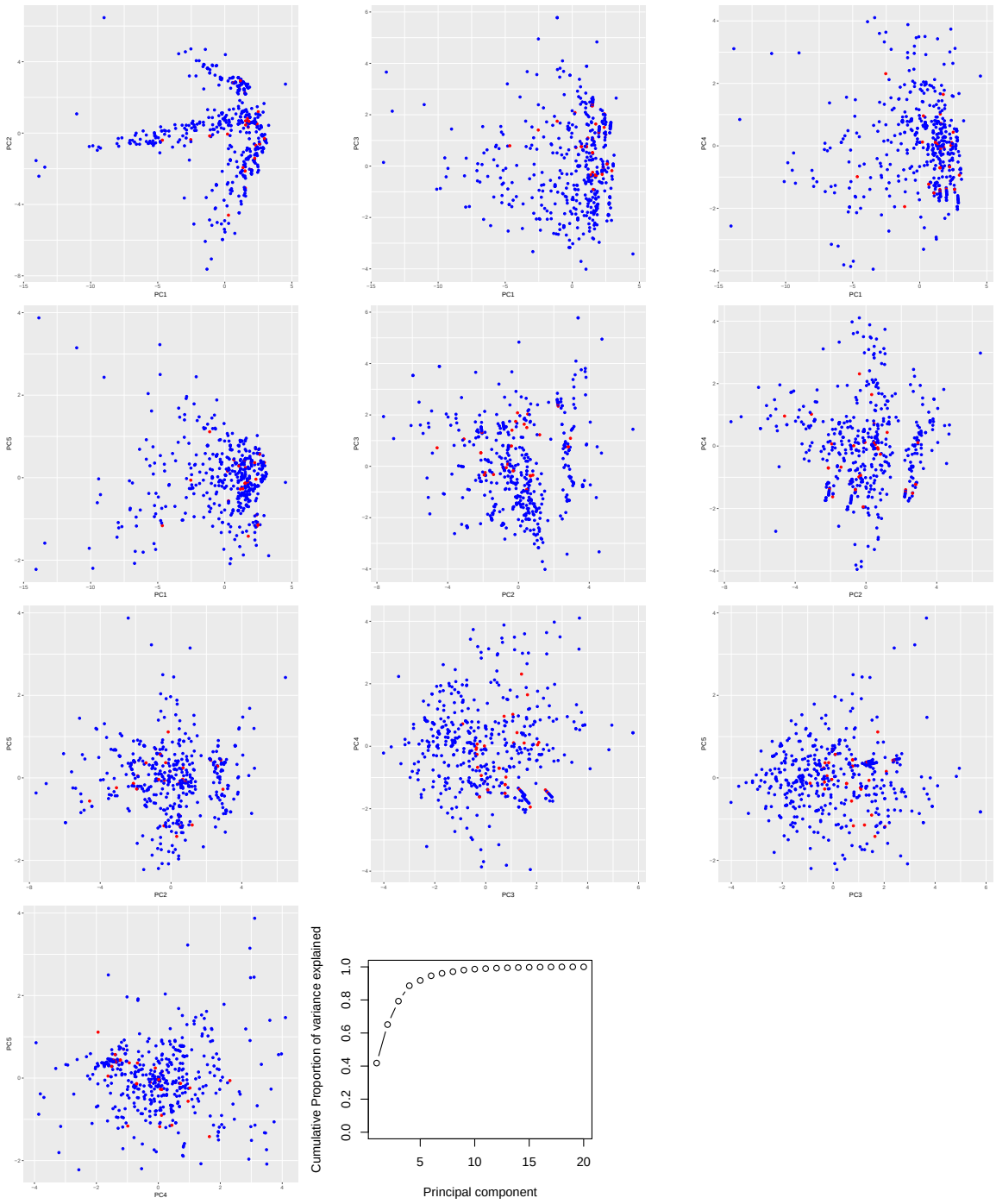


Figure A.6: The PCA plots for our selected hypergraphs for the whole instance set. Red dots represent selected hypergraphs, all other hypergraphs are represented by blue dots.

A.4 Configurations

Parameter	SPM	VLSI	Primal	Literal	Dual	*
c-type	heavy lazy	heavy lazy	heavy lazy	heavy lazy	ml style	heavy lazy
i-c-type	ml style	ml style	ml style	ml style	ml style	heavy lazy
c-heavy_node	no penalty	penalty	penalty	penalty	no penalty	penalty
i-c-heavy_node	no penalty	penalty	penalty	penalty	penalty	no penalty
c-tie-breaking	random	random	random	random	unmatched	random
i-c-tie-breaking	random	random	random	random	unmatched	random
c-s	1.00	6.25	1.00	1.00	4.25	1.00
i-c-s	1.00	2.75	1.00	1.25	1.00	1.00
c-t	340	165	350	320	350	340
i-c-t	200	25	200	200	200	200
i-runs	1	1	1	1	2	2
i-r-fm-stop-policy	adaptive	adaptive	adaptive	adaptive	simple	adaptive
i-r-fm-stop-value	1.01	1.10	1.33	1.27	30	1.00
r-fm-stop-alpha	1.00	1.50	2.31	3.35	1.02	1.26
louvain-iterations	10	10	10	10	10	10
Parameter	SPM	VLSI	Primal	Literal	Dual	*

Table A.8: Our running time optimized configurations.

Parameter	SPM	VLSI	Primal	Literal	Dual	*
c-type	heavy lazy	heavy lazy		ml style	heavy lazy	heavy lazy
i-c-type	heavy lazy	heavy lazy		heavy lazy	ml style	ml style
c-heavy_node	penalty	penalty		no penalty	penalty	penalty
i-c-heavy_node	penalty	penalty		penalty	penalty	penalty
c-tie-breaking	random	random		unmatched	random	random
i-c-tie-breaking	unmatched	random	no	unmatched	unmatched	random
c-s	2.75	7.00	configuration	1.25	3.00	1.00
i-c-s	3.5	2.50	found	1.50	1.00	1.25
c-t	205	195		245	330	105
i-c-t	165	45		110	195	90
i-runs	24	19		19	12	10
i-r-fm-stop-policy	adaptive	simple		adaptive	simple	simple
i-r-fm-stop-value	1.51	30		2.43	50	100
r-fm-stop-alpha	4.47	1.21		2.14	1.59	1.28
louvain-iterations	10	10	10	10	10	10
Parameter	SPM	VLSI	Primal	Literal	Dual	*

Table A.9: Our Pareto optimized configurations.

Parameter	SPM	VLSI	Primal	Literal	Dual	*
c-type	heavy lazy	heavy lazy	ml style	ml style	heavy lazy	heavy lazy
i-c-type	heavy lazy	heavy lazy	heavy lazy	heavy lazy	heavy lazy	ml style
c-heavy_node	penalty	penalty	no penalty	no penalty	penalty	penalty
i-c-heavy_node	penalty	no penalty	no penalty	no penalty	no penalty	no penalty
c-tie-breaking	random	random	unmatched	unmatched	random	random
i-c-tie-breaking	random	random	random	random	unmatched	unmatched
c-s	4.50	4.50	2.50	3.25	4.25	7.00
i-c-s	6.25	1.00	1.00	1.75	2.25	2.00
c-t	200	330	255	245	175	235
i-c-t	195	185	145	120	110	140
i-runs	62	63	71	52	26	37
i-r-fm-stop-policy	simple	simple	simple	simple	simple	simple
i-r-fm-stop-value	160	500	300	440	230	280
r-fm-stop-alpha	2.43	1.04	3.94	4.90	2.00	2.44
louvain-iterations	10	10	10	10	10	10
Parameter	SPM	VLSI	Primal	Literal	Dual	*

Table A.10: Our quality optimized configurations.

A.5 Significance Tests

Optimization Objective	Training Set	Benchmark Set	min $(\lambda - 1)$		avg $(\lambda - 1)$	
			Z	p	Z	p
Pareto	SPM	SPM	-2.54	0.0111	-3.15	0.00161
		*	0.51	0.607	1.44	0.15
	VLSI	VLSI	-3.26	0.00111	-4.12	3.77e-05
		*	-0.88	0.378	-0.62	0.538
	Literal	Literal	-0.10	0.924	0.10	0.922
		*	-1.51	0.132	0.41	0.679
	Dual	SPM	-0.80	0.424	-1.47	0.14
		VLSI	-3.05	0.00232	-3.37	0.000742
		Primal	-0.67	0.501	0.88	0.378
		Literal	0.29	0.774	0.83	0.406
		Dual	-3.85	0.000116	-4.28	1.91e-05
		*	-3.08	0.00209	-2.63	0.00853
	*	SPM	-3.53	0.000419	-3.11	0.00189
		VLSI	-1.78	0.0755	-2.53	0.0113
		Primal	0.53	0.597	2.29	0.0223
		Literal	0.97	0.332	1.21	0.225
		Dual	-2.65	0.00794	-2.56	0.0104
		*	-2.61	0.00915	-1.87	0.0618
Quality	SPM	SPM	-3.95	7.76e-05	-5.40	6.62e-08
		*	-7.21	5.46e-13	-8.38	5.15e-17
	VLSI	VLSI	-5.51	3.62e-08	-7.06	1.66e-12
		*	-4.96	6.94e-07	-6.71	1.96e-11
	Primal	Primal	-4.52	6.18e-06	-4.28	1.88e-05
		*	-11.54	7.83e-31	-14.97	1.2e-50
	Literal	Literal	-3.07	0.00217	-5.57	2.57e-08
		*	-8.61	7.17e-18	-12.10	9.94e-34
	Dual	Dual	-4.58	4.71e-06	-6.23	4.73e-10
		*	-6.11	9.96e-10	-7.38	1.63e-13
	*	SPM	-3.79	0.000153	-4.57	4.92e-06
		VLSI	-5.25	1.54e-07	-6.61	3.83e-11
		Primal	-0.79	0.427	0.39	0.698
		Literal	-2.03	0.0425	-1.79	0.0727
		Dual	-3.56	0.000375	-5.06	4.23e-07
		*	-6.65	2.98e-11	-7.48	7.22e-14
Running Time	SPM	SPM	6.66	2.76e-11	6.89	5.39e-12
		*	17.11	1.25e-65	17.83	3.91e-71
	VLSI	VLSI	4.36	1.28e-05	7.21	5.77e-13
		*	8.40	4.58e-17	12.59	2.5e-36
	Literal	Literal	6.83	8.26e-12	7.31	2.77e-13
		*	13.62	2.99e-42	16.15	1.2e-58
	Primal	Primal	6.39	1.7e-10	7.52	5.61e-14
		*	14.07	5.88e-45	16.11	2.01e-58
	Dual	Dual	4.12	3.81e-05	5.90	3.67e-09
		*	8.76	2.01e-18	12.95	2.49e-38
	*	SPM	3.19	0.00142	3.87	0.000111
		VLSI	8.14	4.02e-16	8.58	9.14e-18
		Primal	5.47	4.39e-08	7.27	3.67e-13
		Literal	6.68	2.42e-11	7.07	1.58e-12
		Dual	4.92	8.79e-07	6.56	5.22e-11
		*	13.14	1.9e-39	15.31	6.63e-53

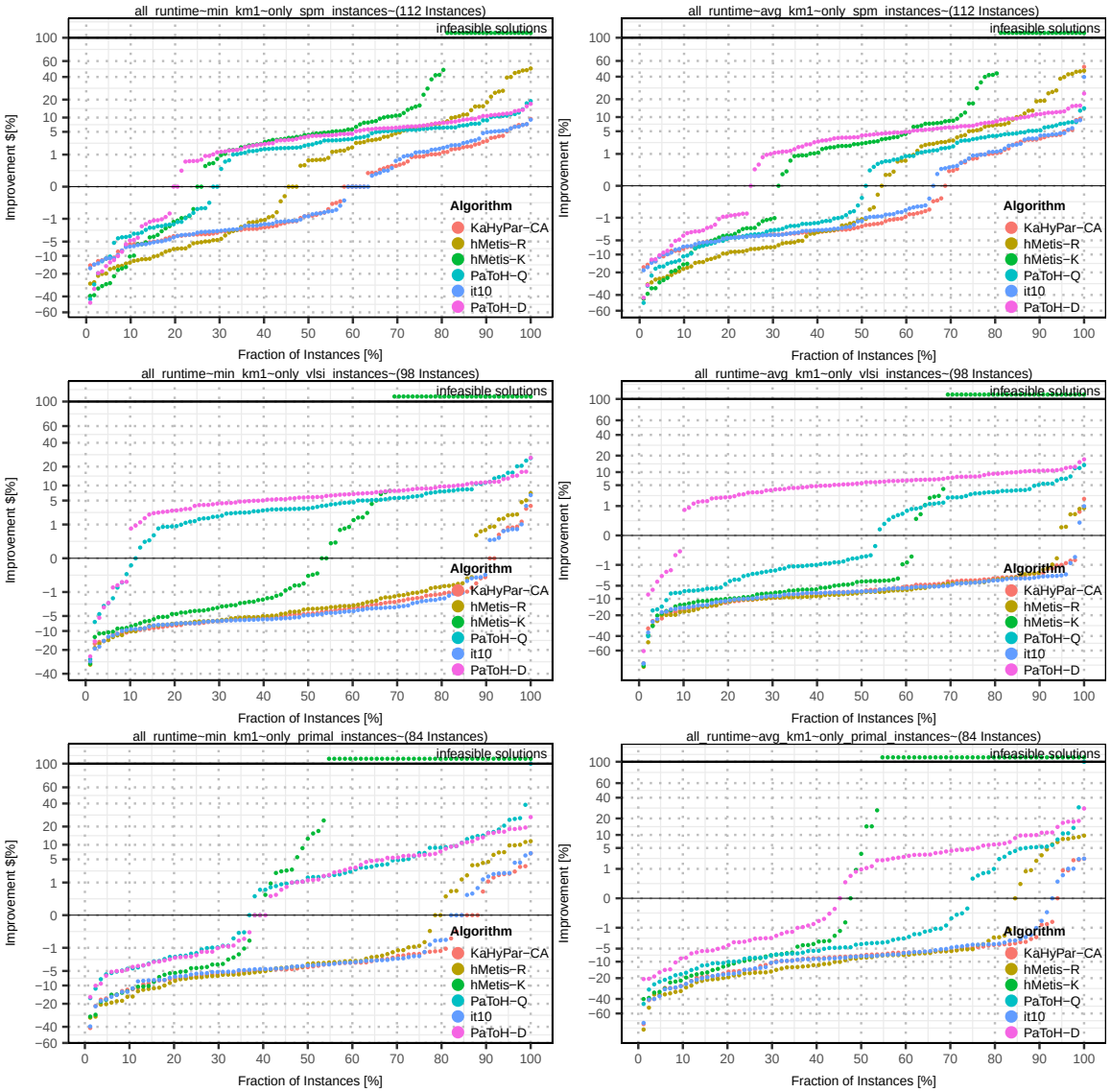
Table A.11: Results of Wilcoxon matched pairs signed rank test comparing our default configuration with benchmarked configurations. At a confidence level of 99%, a $|Z| > 2.58$ is considered as significant. A negative Z-score indicates that the corresponding configuration is better than the default configuration. The full benchmark set as well as the full training set are denoted by '*'.

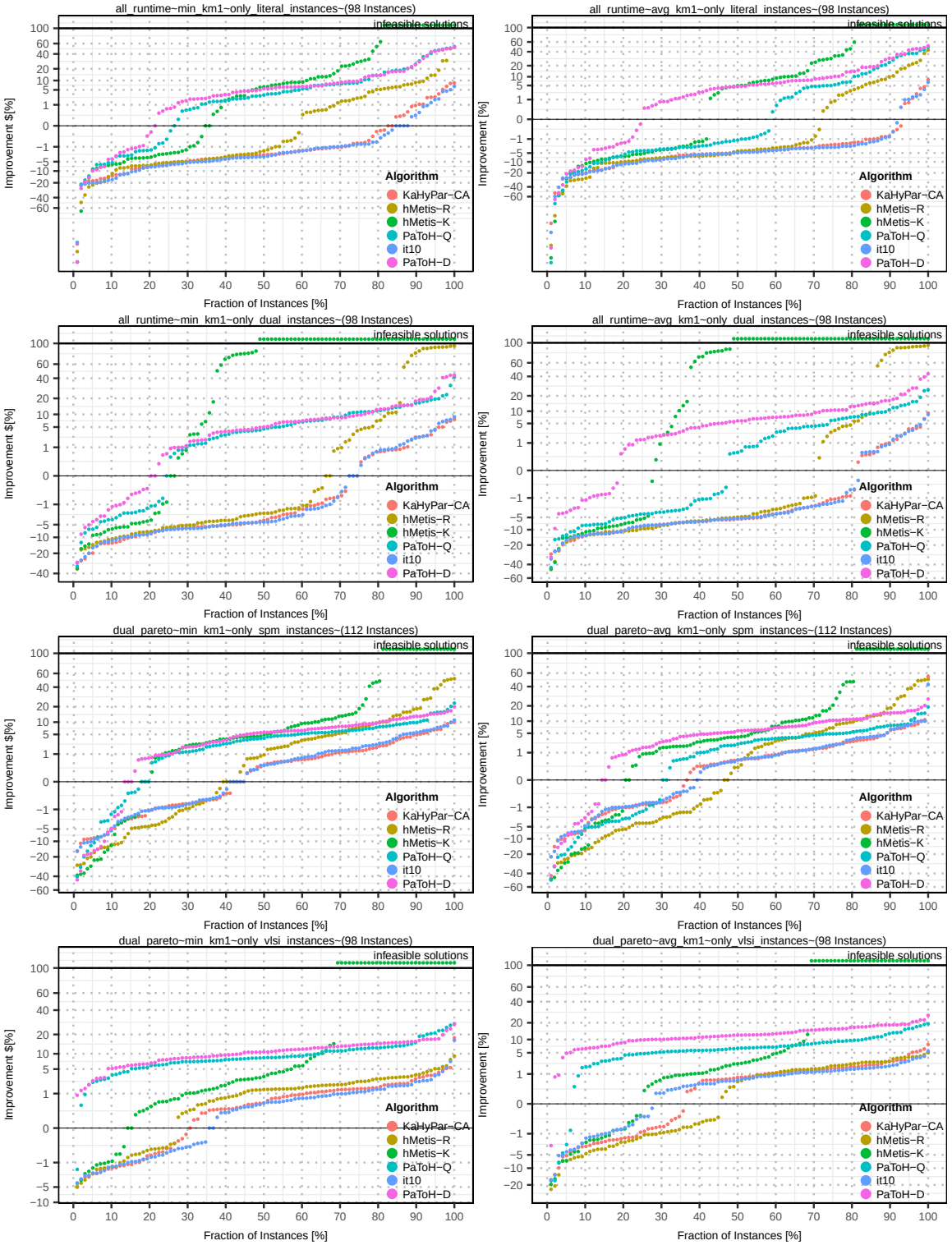
Benchmark Set	Algorithm/ Configuration	min $(\lambda-1)$		avg $(\lambda-1)$	
		Z	p	Z	p
SPM	runtime	3.19	0.00142	3.87	0.000111
	Pareto	-0.80	0.424	-1.47	0.14
	quality	-3.79	0.000153	-4.57	4.92e-06
	KaHyPar-CA	-0.38	0.704	-0.12	0.908
	hMetis-R	1.77	0.0763	0.26	0.795
	hMetis-K	4.28	1.85e-05	3.33	0.000873
	PaToH-Q	5.79	6.9e-09	3.07	0.00214
	PaToH-D	5.96	2.45e-09	6.06	1.36e-09
VLSI	runtime	8.14	4.02e-16	8.58	9.14e-18
	Pareto	-3.05	0.00232	-3.37	0.000742
	quality	-5.25	1.54e-07	-6.61	3.83e-11
	KaHyPar-CA	1.44	0.149	0.15	0.882
	hMetis-R	3.13	0.00173	-0.84	0.4
	hMetis-K	5.24	1.64e-07	3.71	0.00021
	PaToH-Q	8.58	9.42e-18	8.14	4.02e-16
	PaToH-D	8.51	1.69e-17	8.59	8.59e-18
Primal	runtime	5.47	4.39e-08	7.27	3.67e-13
	Pareto	-0.67	0.501	0.88	0.378
	quality	-0.79	0.427	0.39	0.698
	KaHyPar-CA	-1.46	0.143	-0.25	0.806
	hMetis-R	1.42	0.157	-0.48	0.632
	hMetis-K	1.80	0.0716	1.06	0.287
	PaToH-Q	6.91	4.76e-12	6.13	9.04e-10
	PaToH-D	7.05	1.8e-12	7.29	3.22e-13
Literal	runtime	6.68	2.42e-11	7.07	1.58e-12
	Pareto	0.29	0.774	0.83	0.406
	quality	-2.03	0.0425	-1.79	0.0727
	KaHyPar-CA	1.14	0.254	0.78	0.436
	hMetis-R	3.86	0.000114	2.83	0.00461
	hMetis-K	5.70	1.22e-08	4.93	8.04e-07
	PaToH-Q	8.10	5.31e-16	6.41	1.5e-10
	PaToH-D	8.02	1.1e-15	8.10	5.55e-16
Dual	runtime	4.92	8.79e-07	6.56	5.22e-11
	Pareto	-3.85	0.000116	-4.28	1.91e-05
	quality	-3.56	0.000375	-5.06	4.23e-07
	KaHyPar-CA	-0.82	0.414	-0.04	0.966
	hMetis-R	1.77	0.0764	1.12	0.262
	hMetis-K	0.55	0.584	0.38	0.703
	PaToH-Q	7.57	3.76e-14	6.56	5.35e-11
	PaToH-D	8.08	6.51e-16	8.23	1.87e-16
*	runtime	13.14	1.9e-39	15.31	6.63e-53
	Pareto	-3.08	0.00209	-2.63	0.00853
	quality	-6.65	2.98e-11	-7.48	7.22e-14
	KaHyPar-CA	0.18	0.861	0.49	0.623
	hMetis-R	5.45	4.99e-08	1.63	0.103
	hMetis-K	8.35	6.86e-17	6.43	1.29e-10
	PaToH-Q	16.65	2.94e-62	13.29	2.66e-40
	PaToH-D	16.80	2.55e-63	17.12	1.03e-65

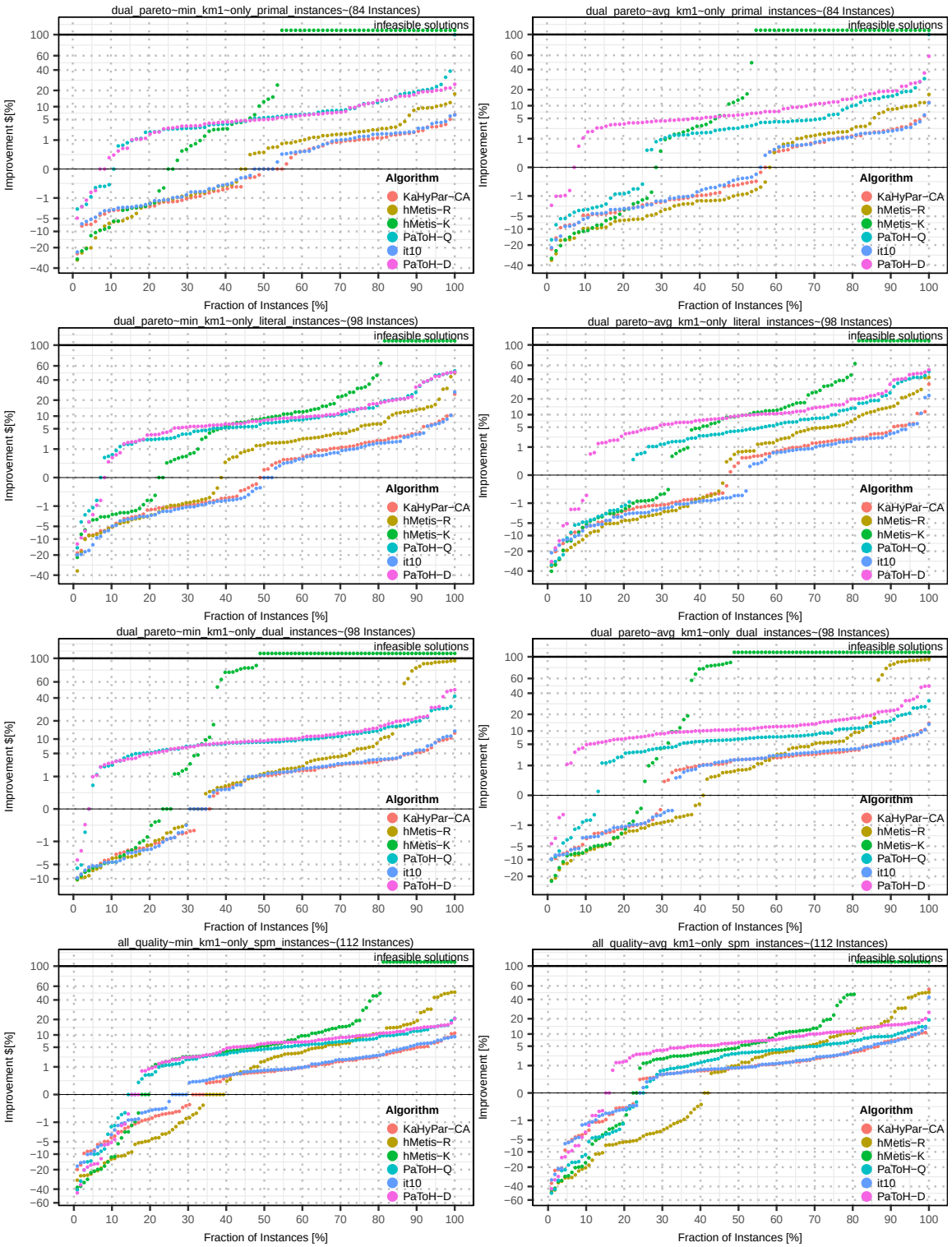
Table A.12: Results of Wilcoxon matched pairs signed rank test comparing our default configuration with our best configurations as well as hMetis and PaToH. At a confidence level of 99%, a $|Z| > 2.58$ is considered as significant. A negative Z-score indicates that the corresponding configuration is better than the default configuration. The full benchmark set is denoted by '*'.

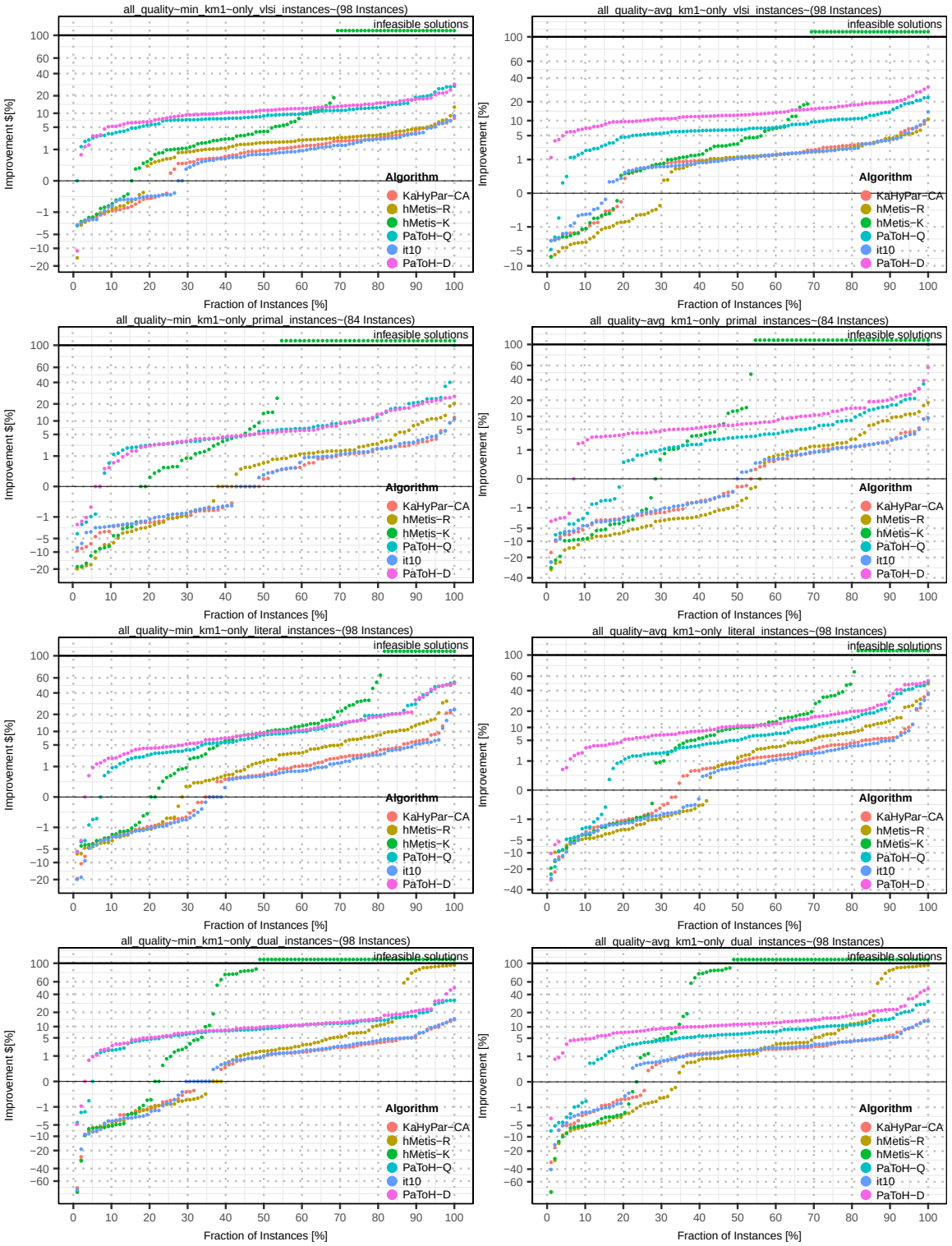
A.6 Improvement Plots

In Section 4.4.4, we compare our best configurations against hMetis and PaToH on the full benchmark set. The missing improvement plots for using only subsets of the benchmark set with the same type are provided here. Note that the name of the configuration as well as the subset of the benchmark set is provided in the headline of each plot.




all_runtime~min_km1~only_literal_instances~(98 Instances)
all_runtime~avg_km1~only_literal_instances~(98 Instances)
all_runtime~min_km1~only_dual_instances~(98 Instances)
all_runtime~avg_km1~only_dual_instances~(98 Instances)
dual_pareto~min_km1~only_spm_instances~(112 Instances)
dual_pareto~avg_km1~only_spm_instances~(112 Instances)
dual_pareto~min_km1~only_vlsi_instances~(98 Instances)
dual_pareto~avg_km1~only_vlsi_instances~(98 Instances)
infeasible solutions
Improvement $[%]
Improvement [%]
Fraction of Instances [%]
Algorithm
KaHyPar−CA
hMetis−R
hMetis−K
PaToH−Q
it10
PaToH−D


dual_pareto~min_km1~only_primal_instances~(84 Instances)
dual_pareto~avg_km1~only_primal_instances~(84 Instances)
dual_pareto~min_km1~only_literal_instances~(98 Instances)
dual_pareto~avg_km1~only_literal_instances~(98 Instances)
dual_pareto~min_km1~only_dual_instances~(98 Instances)
dual_pareto~avg_km1~only_dual_instances~(98 Instances)
all_quality~min_km1~only_spm_instances~(112 Instances)
all_quality~avg_km1~only_spm_instances~(112 Instances)
infeasible solutions
Improvement $[%]
Improvement [%]
Fraction of Instances [%]
Algorithm
KaHyPar−CA
hMetis−R
hMetis−K
PaToH−Q
it10
PaToH−D


all_quality~min_km1~only_vlsi_instances~(98 Instances)
all_quality~avg_km1~only_vlsi_instances~(98 Instances)
all_quality~min_km1~only_primal_instances~(84 Instances)
all_quality~avg_km1~only_primal_instances~(84 Instances)
all_quality~min_km1~only_literal_instances~(98 Instances)
all_quality~avg_km1~only_literal_instances~(98 Instances)
all_quality~min_km1~only_dual_instances~(98 Instances)
all_quality~avg_km1~only_dual_instances~(98 Instances)
infeasible solutions
Improvement $[%]
Improvement [%]
Fraction of Instances [%]
Algorithm
KaHyPar−CA
hMetis−R
hMetis−K
PaToH−Q
it10
PaToH−D

Bibliography

[1] David A. Papa and Igor Markov. Hypergraph partitioning and clustering. 05 2007.

[2] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a direct k-way hypergraph partitioning algorithm. In *19th Workshop on Algorithm Engineering and Experiments, (ALENEX 2017)*, pages 28–42, 2017.

[3] 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.

[4] Robin Andre, Sebastian Schlag, and Christian Schulz. Evolutionary hypergraph partitioning. 2017.

[5] Marijn J.H. Heule Anton Belov, Daniel Diepold. The sat competition 2014, 2014. `http://www.satcompetition.org/2014/`.

[6] Richard Bellman. *Adaptive control process: a guided tour*. 1961.

[7] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(1):281–305, February 2012.

[8] Ruben Burger, Mukunda Bharatheesha, Marc van Eert, and Robert Babuška. Automated tuning and configuration of path planning algorithms. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4371–4376, May 2017.

[9] 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.

[10] Vijay Durairaj and Priyank Kalla. Guiding cnf-sat search via efficient constraint partitioning. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pages 498–501, Nov 2004.

[11] Katharina Eggensperger, Marius Lindauer, and Frank Hutter. Pitfalls and best practices in algorithm configuration. *CoRR*, abs/1705.06058, 2017.

[12] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.

[13] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[14] Karl Pearson F.R.S. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[15] John Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[16] Tobias Heuer, Peter Sanders, and Sebastian Schlag. Network flow-based refinement for multilevel hypergraph partitioning. *arXiv preprint arXiv:1802.03587*, 2018.

[17] Tobias Heuer and Sebastian Schlag. Improving Coarsening Schemes for Hypergraph 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.

[18] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, pages 507–523, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[19] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Parallel algorithm configuration. In *Proc. of LION-6*, pages 55–70, 2012.

[20] Frank Hutter, Thomas Stützle, Kevin Leyton-Brown, and Holger H. Hoos. Paramils: An automatic algorithm configuration framework. *CoRR*, abs/1401.3492, 2014.

[21] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13(4):455–492, Dec 1998.

[22] George Karypis. *Multilevel Hypergraph Partitioning*, pages 125–154. Springer US, Boston, MA, 2003.

[23] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: application in vlsi domain. *Proceedings - Design Automation Conference*, pages 526–529, 1 1997.

[24] 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.

[25] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. *CoRR*, abs/1210.7959, 2012.

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

[27] O. Lima and R. Ventura. A case study on automatic parameter optimization of a mobile robot localization algorithm. In *2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 43–48, April 2017.

[28] Zoltán Ádám Mann and Pál András Papp. Formula partitioning revisited. 2014.

[29] Nicholas Freitag McPhee, Thomas Helmuth, and Lee Spector. Using algorithm configuration tools to optimize genetic programming parameters: A case study. In

Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '17, pages 243–244, New York, NY, USA, 2017. ACM.

[30] Vitaly Osipov and Peter Sanders. n-level graph partitioning. In Mark de Berg and Ulrich Meyer, editors, *Algorithms – ESA 2010*, pages 278–289, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[31] Paola Pellegrini, Grégory Marlière, Raffaele Pesenti, and Joaquin Rodriguez. Recife-milp: An effective milp-based heuristic for the real-time railway traffic management problem. *IEEE Transactions on Intelligent Transportation Systems*, 16(5):2609–2619, Oct 2015.

[32] Carl E. Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[33] John R. Rice. The algorithm selection problem. volume 15 of *Advances in Computers*, pages 65 – 118. Elsevier, 1976.

[34] Peter Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.*, 20(1):53–65, November 1987.

[35] Laura A. Sanchis. Multiple-way network partitioning. *IEEE Trans. Comput.*, 38(1):62–81, January 1989.

[36] Satu Elisa Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[37] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. *k*-way hypergraph partitioning via *n*-level recursive bisection. In *18th Workshop on Algorithm Engineering and Experiments, (ALENEX 2016)*, pages 53–67, 2016.

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

[39] Natarajan Viswanathan, Charles Alpert, Cliff Sze, Zhou Li, and Yaoguang Wei. The dac 2012 routability-driven placement contest and benchmark suite. In *DAC Design Automation Conference 2012*, pages 774–782, June 2012.

[40] Sverre Wichlund. On multilevel circuit partitioning. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '98, pages 505–511, New York, NY, USA, 1998. ACM.

[41] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[42] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Hydra-mip: Automated algorithm configuration and selection for mixed integer programming. 2011.

[43] Ka Yee Yeung and Walter L. Ruzzo. Principal component analysis for clustering gene expression data. *Bioinformatics*, 17(9):763–774, 2001.