Karlsruher Institut für Technologie

Bachelor thesis

Evolutionary Hypergraph Partitioning

Robin Andre

Date: 28. November 2017

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

Institute of Theoretical Informatics, Algorithmics
Department of Informatics

Abstract

The NP-hard hypergraph partitioning problem has many applications like processor load balance or VLSI design. As such meta heuristics are used as approximation to increase solution quality. In this thesis we create an effective evolutionary algorithm improving the existing multilevel hypergraph partitioner KaHyPar. We introduce operations using both the evolutionary and multilevel heuristic. The benefits of the algorithm are evaluated on 90 hypergraphs, showing that the evolutionary component improves solution quality by 2.2% on average.

Abstrakt

Das NP-schwere Hypergraphpartitionierungsproblem besitzt viele Anwendungsgebiete wie Prozessorlastverteilung oder Schaltkreisdesign. Die gängige Strategie ist es Metaheuristiken einzusetzen, um eine Verbessung der Lösungsqualität zu erreichen. In dieser Thesis entwickeln wir einen effektiven evolutionären Algorithmus, der den bestehenden Multilevelpartitionierer KaHyPar verbessert. Wir führen Operationen ein, die sowohl den evolutionären Aspekt als auch den Multilevelaspekt benutzen. Die Vorteile des evolutionären Algorithmus werden auf 90 Hypergraphen evaluiert, wobei wir zeigen dass eine Verbesserung von durchschnittlich 2.2% erzielt wird.

Acknowledgments

I'd like to thank Timo Bingmann for the supply of Club-Mate. And my supervisors for showing superhuman patience.

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

Evolutionary algorithms are inspired by evolution. Similar to the biological counterpart they attempt to simulate an enclosed space where several actors, or individuals, try to compete for survival and reproduction in an isolated setting over the time span of multiple generations. Evolution theory states that individuals having more helpful traits, like special beaks to assist in acquiring food, are more likely to survive longer and thus more likely to pass these helpful traits onto the next generation. Additionally some traits occur randomly through changing the genetic information erratically. These are called mutations and are even present in humans. Some can be harmful like the sickle-cell disease or beneficial like the ability to consume lactose. In evolution theory mutations are usually a factor that introduces previously nonexistent traits, which would not be able to be recreated using reproduction alone. Repeating the cycle of survival and reproduction the mutations that are helpful will more likely be passed on and established. Based on this principle evolutionary algorithms are essentially applying the process described above onto a mathematical problem.

Evolutionary algorithms are a faithful recreation of the basic evolutionary concept. They implement a system which mimics all of the actions found in a gene pool. Mutation, reproduction and selection. In Darwin's theory the individuals ability to survive and reproduce is strictly based on the benefits and detriments of the genes. However, this is not considering the fact that the survival chance of the individual may improve beyond the biological constraint. We humans learn to interpret the world around us, what berries are poisonous, what animals should be hunted, when can the road safely be crossed. All of these factors are improving the survival chance, independent from biological information. A similar concept in computer science are memetic algorithms [31], enhancing the solutions of the evolutionary algorithm by additionally applying local search algorithms to improve the solution.

Evolutionary algorithms are primarily used as a meta-heuristic to generate good solutions for difficult problems. In this thesis we present a memetic algorithm to tackle one of those difficult, NP-hard problems [21] - hypergraph partitioning.

Additionally, we integrate our memetic algorithm into an existing hypergraph partitioner, which is already utilizing the multilevel heuristic [13] and capable of generating solutions of very high quality in comparison to other hypergraph partitioning tools [37].

1.1 Contribution

In this thesis we augment the existing hypergraph partitioner KaHyPar [37] with an evolutionary framework designed to further improve the solution quality generated by KaHyPar. Using ideas presented in the evolutionary graph partitioner KaffPaE [36], we add combination and mutation operations as well as selection and replacement strategies to create an evolutionary algorithm.

1.2 Structure of Thesis

We establish definitions required to understand hypergraph partitioning, related work on the hypergraph partitioning problem as well as the basic work flow of KaHyPar in Chapter 2. Afterwards we introduce and explain the algorithmic components that augment KaHyPar to a memetic algorithm in Chapter 3. Then these algorithmic components are analyzed regarding their usefulness in Chapter 4. Finally we conclude the thesis and give a preview of further improvements to the memetic algorithm in Chapter 5.

2 Preliminaries

In this chapter we establish the mathematical concept of a hypergraph used in this thesis, as well as an overview of related work in the field of evolutionary hypergraph partitioning. We also present the workflow of the partitioner KaHyPar to allow a deeper understanding how the evolutionary algorithm integrates into KaHyPar.

2.1 General Definitions

A hypergraph $H = (V, E, c, w)$ is defined as a set of vertices V, a set of hyperedges E where each hyperedge is a subset of the vertices $e \subseteq V$. The weight of a vertex is measured by $c : V \rightarrow \mathbb{R}_{>0}$. Similarly the weight of a hyperedge is defined by $w : E \rightarrow \mathbb{R}_{>0}$. The set extensions of c and w are defined as $c(V') = \sum_{v \in V'} c(v)$ and $w(E') = \sum_{e \in E'} w(e)$. Two vertices u, v are adjacent if $\exists\ e \in E\ :\ u, v \in e$. The vertices in e are called pins. A vertex u is incident to a hyperedge e if $u \in e$. $I(u)$ is the set of all incident hyperedges of node u. The size $|e|$ of a hyperedge e is the number of pins in e. A k-way partition of a hypergraph H is a partition of V into k disjoint blocks $V_1, ..., V_k$. A vertex u is assigned a partition block by the function $part(u) : V \rightarrow [1, k]$. A k-way partition is balanced when $\forall\ 1 \leq i \leq k\ :\ c(V_i) \leq (1 + \epsilon)\lceil \frac{c(V)}{k} \rceil$ for an imbalance parameter ϵ. A valid solution is a balanced k-way partition. An invalid solution is a partition where the balance criterion is not met. The number of vertices in a hyperedge that are located in V_i are measured by $\Psi(e, V_i) := |\{v \in V_i \mid v \in e\}|$. Given a partition $\mathcal{P}$ the connectivity set of a hyperedge e is $\Phi(e, \mathcal{P}) := \{V_i \mid \Psi(e, V_i) > 0\}$. A hyperedge e is a cut edge in a partition $\mathcal{P}$ when $\Phi(e, \mathcal{P}) > 1$. Let $\mathcal{E}$ be the set of cut edges from a partition $\mathcal{P}$. The cut metric of $\mathcal{P}$ is defined as $cut(\mathcal{P}) := w(\mathcal{E})$. The connectivity metric $(\lambda - 1)$ is defined as $(\lambda - 1)(\mathcal{P}) := \sum_{e \in \mathcal{E}} (\Psi(e) - 1)w(e)$. Both metrics can be used to measure the quality of a solution. We use the connectivity as metric.

2.2 Related Work

There are several hypergraph partitioning algorithms, originating from various backgrounds such as processor communication balancing [14], circuit partitioning [3] or database storage sharding [25].

Two approaches are used for hypergraph partitioning. The first approach is the bisectioning of the hypergraph, where the partition is fixed to $k = 2$ as in MLPart [3]. By recursively bisecting the resulting partitions, k can assume values other than 2. This is implemented in tools like PaToH [14], Mondrian [39], Zoltan [19] and hMetis [26]. The other approach is to skip the recursion and directly partition the hypergraph into k blocks. This is called a direct k-way partition and is used in hMetis-Kway [27], kPatoH [7] and SHP [25] (also implementing a recursive bisection). Note that except SHP [25] all tools are utilizing the multilevel paradigm [13].

Of course this is only a collection of the most common hypergraph partitioners, which is why we would like to refer to the surveys [4, 9, 32, 38] for an extensive overview.

Saab and Rao [33] present one of the first evolutionary approaches to hypergraph partitioning by comparing the gain of a vertex move to a random threshold. They utilize bin packing with iterative improvement to determine the vertex moves. Hulin [24] presents a genetic algorithm maintaining multiple solutions using a two dimensional representation of circuits and introduces a problem specific crossover operator as well as a mutation operator. A more sophisticated memetic algorithm for the hypergraph partitioning problem was created by Bui and Moon [12], in which solutions are preprocessed and optimized using the Fiduccia-Mattheyses [20] local search algorithm as well as a new replacement strategy both considering the bit-wise difference of the child and the parent partitions as well as solution quality.

Chan and Mazmudner [15] provide an genetic algorithm for bipartitioning that assigns better solutions a higher chance to be selected for the crossover operation. The crossover operation splits both input partitions at the same point, combining the first split of the first partition and the second split of the second partition. Areibi [5] gives another memetic algorithm for the k-way hypergraph partitioning problem. Using a variation of FM (Fiduccia-Mattheyses) designed for k-way optimization [35] as well as a 4-point crossover operation, which splits the input partitions at 4 points and alternates between the blocks. Kim et al. [28] translate the lock gain local search [29] for graphs to hypergraphs and use solution quality and hamming distance as a more potent replacement strategy as well as roulette selection to determine the solutions used in the crossover. Sait et al. [34] compare the meta-heuristics tabu search, simulated annealing and genetic algorithms for k-way hypergraph partitioning. Their result is that tabu search is outperforming a genetic algorithm in quality and running time. Armstrong et al. [6] analyze the quality and running time performance of parallel memetic algorithms comparing a bounded amount of local search against an unbounded local search, stopping only when no improvement can be made. All referenced works that use a crossover operator do so by splitting the input partitions and selecting alternating block fragments.

Sanders and Schulz present an evolutionary framework [36] for the existing graph partitioner KaFFPa [23], introducing different combination and mutation operations for graph partitioning. Their approach modifies a multilevel framework to provide effective combine and mutation operations.

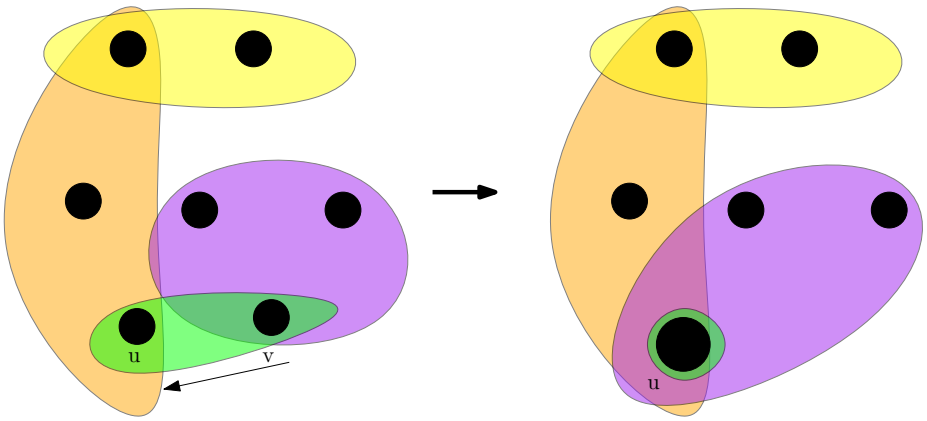

Figure 2.1: An example of a contraction. Note that the hyperedges incident to the contracted node v are modified to contain u.

2.3 KaHyPar

The hypergraph partitioner KaHyPar optimizes the connectivity metric using direct k-way partitioning [1] as well as the cut metric using recursive bisection [37]. KaHyPar also uses a multilevel approach for partitioning (see Figure 2.2). The original hypergraph H is coarsened by repeatedly contracting nodes u, v until either no more contractions are possible or that the minimum number of nodes required has been reached. The coarsened hypergraph is referenced as H_c. KaHyPar is a n-level algorithm meaning that during each step of the coarsening only one pair of nodes u, v is contracted. All of hyperedges containing v are mapped to u in the process. See Figure 2.1 for an example. On H_c a partitioning algorithm is generates an initial partitioning for the coarsened hypergraph. Afterwards contractions are reversed and during each step of the uncoarsening phase local search algorithms are used to improve the solution quality of H. The local search is using a variant of the Fiduccia-Mattheysis algorithm [20]. Heuer and Schlag [22] improved KaHyPar by analyzing and exploiting community structures in hypergraphs, showing that KaHyPar-CA (community aware KaHyPar) generates solutions of superior quality compared to other established hypergraph partitioners.

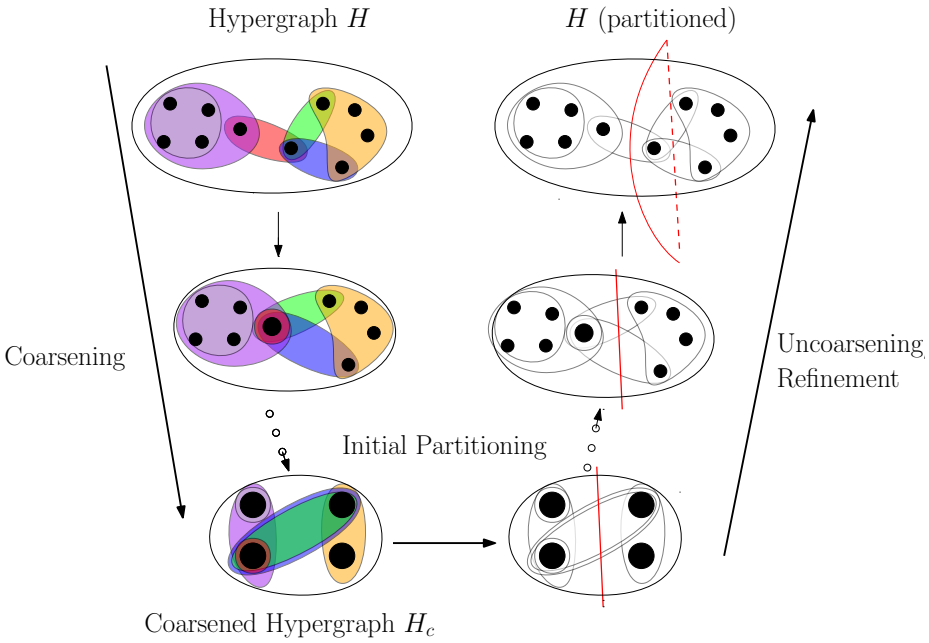

Figure 2.2: An example of the multilevel paradigm utilized by KaHyPar

In Figure 2.2 the multilevel approach of KaHyPar is displayed. The multilevel heuristic [13] begins by reducing the original problem to a smaller problem with a similar structure. Due to the reduced complexity, the smaller problem can be solved efficiently. To convert the solution of the small problem on the original problem, the reduction is undone. To further improve the solution, local search can be applied during this process. Because of the reduction that retains some structure and the local search improvement, the resulting solution for the original problem is of high quality. For hypergraph partitioning the reduction is to contract nodes until the resulting hypergraph is sufficiently small. The solution generation for the small problem is called initial partitioning. The reduction is undone by reverting the process of contraction.

3 KaHyPar-E

In this Chapter we first outline the general procedure of an evolutionary algorithm. Then we explain the methodology used to represent hypergraph partitioning in an evolutionary framework. We also describe the concept of diversity to compare two different partitions on similar properties. Finally operators for combination and mutation are introduced as well as strategies for selection and replacement.

3.1 Overview

In order to apply an evolutionary algorithm we first have to generate an initial population. Following the principle of evolutionary algorithms after the initial solutions are generated the same steps are repeated until a stopping criterion has been met. Evolutionary algorithms are usually repeating 4 steps that attempt to simulate evolution. First some individuals have to be chosen for recombination. Then the chosen individuals have to be combined with each other generating offspring. As third step mutations are performed on some individuals and as fourth step the individuals surviving the iteration (generation) are selected by a corresponding metric, also called fitness. For hypergraph partitioning we consider a partition as an individual and the fitness of said individual is the cut or connectivity metric. We alternate the evolutionary scheme a bit in a sense that we perform combination or mutation exclusively during an iteration and additionally only generate one new solution during said iteration and then replace an existing individual with the new offspring. In Sections 3.2 - 3.4 we explain the tools required for maintaining and selecting individuals from the population.In Section 3.5 we introduce the two combine operators used to generate offspring. In Section 3.6 we describe the mutation operations implemented in KaHyPar-E to complete the evolutionary framework. Finally we explain in Section 3.7 the process to insert the solutions generated by combination and mutation into the population.

3.2 Population

For the purposes of evolutionary algorithms we will begin to use the term individual. For hypergraph partitioning we reference a partition of a hypergraph as individual. KaHyPar-E will produce multiple individuals, which are inserted and removed from the population. Only a fixed number of individuals are in the population. This number is the maximum

population size. Further individuals have to compete for a place in the population. The population size is an important parameter, as a small value limits the exploration capability and a high value limits convergence [16]. We use KaHyPar to create the initial population. This means that unlike most evolutionary algorithms we use high quality solutions instead of random solutions as initial population. However generating an initial population is time consuming and needs a time bound to ensure that the evolutionary components are used within the running time limit for the algorithm. In order to select a proper population size for the running time we attempt to allocate a fair amount of time towards the creation of the initial population.

By measuring the duration of one iteration t_1 and comparing it to the total running time t_{total} we can estimate the amount of iterations $\frac{t_{total}}{t_1}$. Since hypergraph instances vary greatly in time required to partition, using a fixed population size will most likely be inadequate for most instances. As a solution we try to use a fixed percentage of the total time for generating individuals for the initial population. Evaluating the value calculated above we spend approximately 15% of the allotted time towards creating the initial population and consequently the population size is determined by $\delta = 0.15 \cdot \frac{t_{total}}{t_1}$. However, we introduce lower and upper bounds for the population size to ensure a proper size for evolutionary operators as well as convergence. That being said the population size has to be at least 3 and 50 at most.

3.3 Diversity

In biological evolution a population with a highly miscellaneous gene pool is considered healthy, because the variation between the individuals ensures that few to none characteristics are carried by every individual and a high diversity of the gene pool is assured. In that case bad characteristics can be removed through means of reproduction. As reverse conclusion bad characteristics will not be removed if each individual shares said characteristics. The same principle is applicable to evolutionary algorithms in a sense that bad characteristics are unable to be removed if they are shared by all solutions. For the hypergraph partitioning problem such a characteristic would be a hyperedge that is a cut edge in *every* solution.

Maintaining diversity is highly recommended [8], as it ensures a strong perturbation of the solutions and therefore allows for a greater exploration of the solution space. Additionally it prevents characteristics from manifesting in the entire population. We introduce diversity as a tool for measuring the different characteristics of two individuals. As described above the characteristic influencing the quality of a partition are the cut edges. In evolutionary graph partitioning, KaffPaE [36] determines the difference of two partitions $\mathcal{P}_1$, $\mathcal{P}_2$ by counting the edges that are cut edges in exclusively one of the partitions $cutdiff(\mathcal{P}_1, \mathcal{P}_2) := \sum_{e \in E} |cut(e, \mathcal{P}_1) - cut(e, \mathcal{P}_2)|$. This approach can be used for hypergraphs, but is not entirely accurate for hypergraphs because cut hyperedges might extend

into multiple blocks,(see Figure 3.1). Instead we also count the amount of blocks that are different between the hyperedges $conndiff(\mathcal{P}_1, \mathcal{P}_2) := \sum_{e \in E} ||\Phi(e, \mathcal{P}_1)| - |\Phi(e, \mathcal{P}_2)||$. This is a more natural representation for the connectivity metric than cut edges.

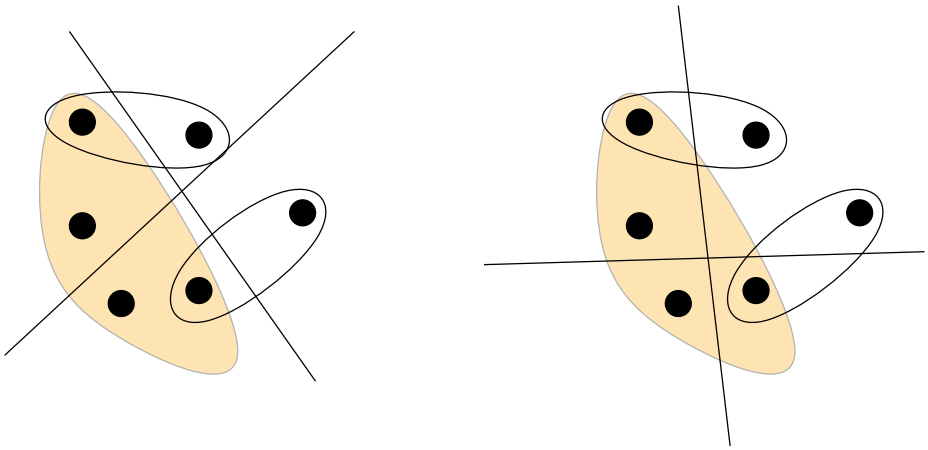

Figure 3.1: Two different partitions of the same hypergraph

Figure 3.1 shows two different partitions of the same example hypergraph. Since all edges are cut edges in both partitions the cut difference is 0. However the partitions are not to be considered equal since the highlighted edge has a different connectivity. By using the connectivity difference this issue can be avoided.

3.4 Selection Strategies

For an evolutionary algorithm we attempt to generate new, improved solutions by using existing solutions. A logical conclusion is that good individuals have good characteristics that may be passed on to child individuals. We select our individuals using tournament selection [11], meaning that the individuals are competing for their chance of recombination based on their fitness. In a long term perspective this ensures that good individuals have a higher chance of reproduction as stated by the evolutionary theory. By then selecting two random individuals and choosing the one with the better solution quality we extract one individual I_1. For operators requiring two separate individuals, we can simply repeat this step to select a new individual I_2. In the unlikely case that the two selected individuals are the same we instead use the worse individual from the second tournament selection round.

3.5 Combine Operations

Combine operations generate a new individual by using two or more individuals as input. We present two different combine operators. The first operator C1 combines two partitions

which are determined using tournament selection. While C1 uses two parents, the second operator C2 is capable of combining a variable number of X individuals. These individuals are selected from the population by choosing the best X individuals from the population instead of a tournament selection. Both operators use the replacement strategy introduced in Section 3.7 to insert the newly generated individuals.

3.5.1 Basic Combine (C1)

The basic combine operator C1 uses two parent partitions $\mathcal{P}_1$, $\mathcal{P}_2$ in order to create a child individual C. This is achieved by only allowing contractions of nodes u, v when these nodes are in the same block for both parent partitions as seen in Figure 3.2. Contractions performed in the same block do not modify the quality of a partition, because the block assignments cannot change, and as a result the connectivity sets remain stable. This ensures that the solution quality for the coarsened hypergraph H_c does not fluctuate for either parent partition. After coarsening we do not perform an initial partitioning. Instead we consider the coarsened hypergraph H_c and apply the parent partition with better solution quality to H_c.

This operation is different than a V-cycle in Section 3.6.1, since the coarsening condition is more strict due to the consideration of both parents partitions. The local search algorithms during the refinement phase do not worsen the solution quality. This local search assurance in combination with using the better partition of the two parents ensures that the child solution is at least as good as the best parent solution. The basic combine is benefiting from highly diverse parent partitions since it passes on more characteristic cut edges, resulting in more exploration of the solution space. This operation results in high convergence due to the quality assurance and strong limitation during the coarsening.

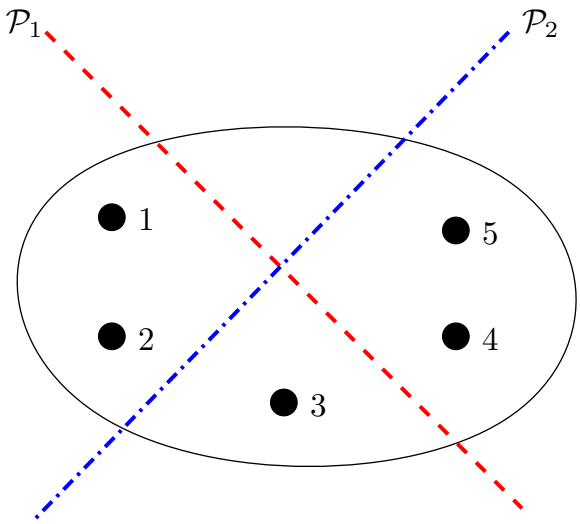

Figure 3.2: An example of allowed contractions with two partitions

In Figure 3.2 we explain the limitation used during the coarsening of operator C1. The only possible contractions are vertices 1 & 2, as well as 4 & 5. Other contractions will violate the restriction. The bertices 2 & 3 for example cannot be contracted because it violates $\mathcal{P}_2$, even though they share the same block in partition $\mathcal{P}_1$.

3.5.2 Edge Frequency Multicombine (C2)

Our multi-combine operator is capable of combining multiple individuals $I_1..I_n, n \leq |P|$ into a new child individual. By analyzing whether an edge e is a cut edge in $I_1..I_n$ we can calculate the edge frequency $f(e) := \sum_{i=1}^{n} cut(e, I_i)$ [41] of an edge e. We use the best $n = \sqrt{|P|}$ individuals from P for determining edge frequency as a standard parameter [18]. Assuming that the frequent cut edges of the best solutions are most likely a good characteristic, these edges should remain cut edges. The contraction of the nodes in a frequent cut edge is most likely not able to push the edge out of the cut, because the cut edge is sufficiently influencing the solution quailty. Therefore the most likely result is the limitation or unenforceability of other possible contractions. By prolonging the contraction of nodes in these frequent cut edges, other contractions may be performed before contracting the nodes in the cut edge. Therefore we penalize contractions of nodes incident to a high frequency edge during the multilevel partition approach by using the rating function

$$r(u,v) = \sum_{e \in I(u) \cap I(v)} \frac{e^{-\gamma * f(e)}}{(w(u)w(v))^{\rho}}$$

to disincentivize early contractions of nodes in edges of high frequency. The value $\rho = 1.2$ on the rating function is a tuning parameter taken from [18] as well as $\gamma = 0.5$. This rating

function is replacing the normal rating function used during the coarsening of KaHyPar. The edge frequency operator is not using the input partitions for H_c. Instead a new initial partitioning is performed and refined in the uncoarsening phase using local search algorithms. Since this operation is generating a new initial partitioning there is no quality assurance opposed to C1.

3.6 Mutation operations

The main objective of mutations is to create more diverse solutions and to avoid premature population convergence towards a local optimum. We propose three different mutation operations. The first operator M1 is intended to improve the solution quality of a partition by reapplying local search algorithms. The second operator M2 is a variation of M1, capable of generating new features. The third operator M3 is intentionally enforcing new characteristics, trying to provide diversity.

3.6.1 V-Cycle (M1)

A V-cycle is a KaHyPar iteration with the difference that the hypergraph is already partitioned. Similar to the combine operator C1 in Section 3.5.1 during coarsening nodes u, v may only be contracted if $part(u) = part(v)$. Since the hypergraph is already partitioned there is no need for initial partitioning. The main benefit comes from refinement during uncoarsening. Due to randomization in coarsening, the structure of the coarsened hypergraph can vary and allow improvements previously not found during local search.

Using an individual I as partition for the hypergraph this operation results in a similar individual I_{new} which has been improved on during the refinement. Due to the fact that neither refinement nor coarsening worsen solution quality I_{new} will have a quality at least equal to I. This is a weak mutation, since the difference of I and I_{new} is small. This operation will cause convergence, as multiple applications of a V-cycle will eventually no longer improve the solution.

3.6.2 V-Cycle + New Initial Partitioning (M2)

Similar to a V-cycle, we coarsen an already partitioned hypergraph H, but instead of immediately starting the refinement we drop the partition and perform a new initial partitioning on the coarsened hypergraph. This operation perturbs the original more strongly because the vertices are no longer forced to keep their assigned block. Since the partition is dropped the algorithm used to generate a new partition might produce a worse solution as before. Therefore this operator can create worse solutions and as a result the operator is capable of increasing diversity in the population.

3.6.3 Stable Nets (M3)

Lim et al. [30] introduce the concept of stable net removal, stating that hyperedges that remain cut edges throughout successive multilevel iterations are trapping the FM-algorithm in a local minimum. Their solution is to force these hyperedges from the cut by forcing them into one block. We use this approach to similarly force recurring cut edges into one block. Opposed to edge frequency operator where the recurring cut edges should remain cut edges, we attempt to force the high frequency edges from the cut. Again the $\sqrt{|P|}$ best individuals are analyzed regarding edges most frequent in these solutions.

We consider an edge stable if it is in the cut in at least 75% of individuals inspected. These edges are then attempted to be forced into the block with the smallest weight in order to maintain the balance constraint. This is done by assigning all nodes $v \in e$ the block id of the smallest block. Each node may only be moved once. These solutions have most likely significantly worse quality.In order to keep these solutions competitive we therefore perform a V-cycle after removing the stable nets. This operator is intended to create individuals with significantly different characteristics.

3.7 Replacement Strategies

Regardless of the operator, the new individuals have to be inserted into the population in order to be used in upcoming iterations. The replacement strategy is the only method of removing an individual from the population. Therefore the replacement strategy is the driving factor of selection pressure and must maintain a strict constraint on the fitness of the individuals to ensure convergence towards good solutions. The naive approach is to remove the worst element from the population and insert the individual in its place, with the intention to ensure a vast majority of the best generated solutions. The consequence is that the population is rapidly converging towards a local optimum and only covering a small part of the solution space. Another approach is to replace one of the elements used in the operator.But this approach neglects fitness and is suboptimal for operators using more than one parent element.

We use a different strategy maintaining the competitive pressure of the selection whilst also avoiding premature convergence. Similar to the naive approach we consider the fitness of the newly generated individual to replace an individual with worse quality. However we do not replace the worst existing individual, instead we replace the most similar individual with a worse quality using $conndiff(\mathcal{P}_1, \mathcal{P}_2)$ as measurement for similarity. By only replacing elements of worse quality the population is slowly improving in quality and converging towards optima. However this approach ensures a more diverse population which boosts the combine operator effectiveness and avoids premature convergence.

4 Experimental Evaluation

In this chapter the previously described algorithmic components are tested in an experimental setting. At first the conditions of the experiment are established and the methodology for evaluating the results is described. Then the experimental results are presented and discussed.

4.1 Experimental Setup

We use two benchmark sets for evaluation. Both sets are using instances from the benchmark set of Heuer and Schlag [22], which consists of various hypergraphs from the ISPD98 benchmark [2], university of Florida Sparse Matrices [17], the DAC benchmark [40] and SAT competition instances [10].

The first set is called the *tuning subset*. It consists of 25 hypergraph instances. The instances were chosen to accurately represent the complete benchmark set. However no instance requiring a high partition time was chosen. This was done to ensure that the instances of the tuning subset can display the effects of the evolutionary algorithm within a smaller time window. The running time for partitions on the tuning subset is 2 hours. The instances are partitioned for $k = 32$ and $\epsilon = 0.03$. Each partitioning run of the tuning subset is repeated 3 times with a different seed for the randomization. This results in 150 CPU-hours required for each data set produced on the tuning subset.

The second set is called the *benchmark subset*. It consists of 90 hypergraph instances from the benchmark set. All instances in the benchmark subset are partitioned for $\epsilon = 0.03$ and $k = \{2, 4, 8, 16, 32, 64, 128\}$. The running time for each partitioning on the benchmark subset is 8 hours, and each run is repeated 5 times. This results in 25200 CPU-hours necessary for each data set produced on the benchmark subset.

Due to the high time requirements of the experiments, we use simple parallelization to generate results within reasonable time. As such all physical cores of the machines are performing a seperate partition instance. Due to shared memory access the timestamps are not completely synchronized. This causes slight fluctuation in the plots. The reason for repeating the runs is to balance out possible outliers due to randomization. This process is described more detailed in Section 4.2. The tuning subset is used to evaluate the algo-

rithmic components and tune the respective parameters within a reasonable amount of time while avoiding overtuning. The benchmark subset is used to replicate the results of the tuning subsets, ensuring statistical significance by using a higher amount hypergraphs and more detailed analysis. The purpose of using two different sets is that the tuning subset is capable of analyzing multiple algorithmic components within a reasonable amount of time, whereas the benchmark subset is able to express more powerful conclusions due to its size.

We use the Wilcoxon Signed Rank Test [42] to evaluate statistical significance between seperate experiments. The Z-value and p-value will be listed in the corresponding experiments. We consider $p \leq 0.05$ significant. The purpose of this experimental evaluation is to show an improvement of solution quality when comparing the evolutionary algorithm with KaHyPar. (And due to the results of [1] indirectly with other hypergraph partitioning tools) We compare our algorithms with the nonevolutionary algorithm KaHyPar-CA [22], which will be referenced as K_N-CA for readability purposes. Additionally K_N-CA can be improved using V-cycles as described in Section 3.6.1. Similar to [6] applying local search until no improvement has been found, K_N-CA can perform V-cycles until no further improvement can be found. Such a stopping criterion is implemented in KaHyPar. We set the number of V-cycles to be performed in K_N-CA $\#V - cycles$ to 100. This algorithm configuration is called K_N-CA-V. Neither K_N-CA nor K_N-CA-V are designed to produce multiple solutions within a fixed time. In order to allow a fair comparison with the evolutionary algorithm on the benchmark sets, both K_N-CA and K_N-CA-V are restarted repeatedly to ensure a proper usage of the running time.

4.2 Instances & Methodology

Table 4.1 displays the instances in the tuning subset, as well as their basic properties. n is the number of vertices, m is the number of hyperedges and p is the number of pins. The instances are sorted by their respective classes. Similarly Table 4.2 displays the instances of the benchmark subset.

Table 4.1: Properties of the hypergraphs used in the tuning subset.

Hypergraph	n	m	p	Hypergraph	n	m	p
ISPD98				SAT14Primal			
ibm06	32498	34826	128182	6s153	85646	245440	572692
ibm07	45926	48117	175639	aaai10-planning-ipc5	53919	308235	690466
ibm08	51309	50513	204890	atco_enc2_opt1_05_21	56533	526872	2097393
ibm09	53395	60902	222088	dated-10-11-u	141860	629461	1429872
ibm10	69429	75196	297567	hwmcc10-timeframe	163622	488120	1138944
SAT14Dual				SPM			
6s133	140968	48215	328924	laminar_duct3D	67173	67173	3833077
6s153	245440	85646	572692	mixtank_new	29957	29957	1995041
6s9	100384	34317	234228	mult_dcop_01	25187	25187	193276
dated-10-11-u	629461	141860	1429872	RFdevice	74104	74104	365580
dated-10-17-u	1070757	229544	2471122	vibrobox	12328	12328	342828
SAT14Literal							
6s133	96430	140968	328924				
6s153	171292	245440	572692				
aaai10-planning-ipc5	107838	308235	690466				
atco_enc2_opt1_05_21	112732	526872	2097393				
dated-10-11-u	283720	629461	1429872				

Table 4.2: Properties of the hypergraphs used in the benchmark subset.

Hypergraph	n	m	p
DAC2012			
superblue19	522 482	511 685	1 713 796
superblue13	630 802	619 815	2 048 903
superblue14	698 339	697 458	2 280 417
superblue3	917 944	898 001	3 109 446
ISPD98			
ibm09	53 395	60 902	222 088
ibm11	70 558	81 454	280 786
ibm10	69 429	75 196	297 567
ibm12	71 076	77 240	317 760
ibm13	84 199	99 666	357 075
ibm14	147 605	152 772	546 816
ibm15	161 570	186 608	715 823
ibm16	183 484	190 048	778 823
ibm18	210 613	201 920	819 697
ibm17	185 495	189 581	860 036
SAT14Dual			
AProVE07-27	29 194	7 729	77 124
countbitssrl032	55 724	18 607	130 020
6s184	97 516	33 365	227 536
6s9	100 384	34 317	234 228
6s133	140 968	48 215	328 924
6s153	245 440	85 646	572 692
atco_enc1_opt2_10_16	152 744	9 643	641 139
aaai10-planning-ipc5	308 235	53 919	690 466
hwmcc10-timeframe	488 120	163 622	1 138 944
itox_vc1130	441 729	152 256	1 143 974
dated-10-11-u	629 461	141 860	1 429 872
manol-pipe-g10bid_i	792 175	266 405	1 848 407
manol-pipe-c8nidw	799 867	269 048	1 866 355
atco_enc2_opt1_05_21	526 872	56 533	2 097 393
dated-10-17-u	1 070 757	229 544	2 471 122
ACG-20-5p0	1 390 931	324 716	3 269 132
ACG-20-5p1	1 416 850	331 196	3 333 531
SAT14Literal			
AProVE07-27	15 458	29 194	77 124
countbitssrl032	37 213	55 724	130 020
6s184	66 730	97 516	227 536
6s9	68 634	100 384	234 228
6s133	96 430	140 968	328 924
6s153	171 292	245 440	572 692
atco_enc1_opt2_10_16	18 930	152 744	641 139
aaai10-planning-ipc5	107 838	308 235	690 466
hwmcc10-timeframe	327 243	488 120	1 138 944
itox_vc1130	294 326	441 729	1 143 974
dated-10-11-u	283 720	629 461	1 429 872
atco_enc1_opt2_05_4	28 738	386 163	1 652 800
manol-pipe-g10bid_i	532 810	792 175	1 848 407
manol-pipe-c8nidw	538 096	799 867	1 866 355
atco_enc2_opt1_05_21	112 732	526 872	2 097 393
dated-10-17-u	459 088	1 070 757	2 471 122
ACG-20-5p0	649 432	1 390 931	3 269 132
ACG-20-5p1	662 392	1 416 850	3 333 531

Hypergraph	n	m	p
SAT14Primal			
AProVE07-27	7 729	29 194	77 124
countbitssrl032	18 607	55 724	130 020
6s184	33 365	97 516	227 536
6s9	34 317	100 384	234 228
6s133	48 215	140 968	328 924
6s153	85 646	245 440	572 692
atco_enc1_opt2_10_16	9 643	152 744	641 139
aaai10-planning-ipc5	53 919	308 235	690 466
hwmcc10-timeframe	163 622	488 120	1 138 944
itox_vc1130	152 256	441 729	1 143 974
dated-10-11-u	141 860	629 461	1 429 872
atco_enc1_opt2_05_4	14 636	386 163	1 652 800
manol-pipe-c8nidw	269 048	799 867	1 866 355
atco_enc2_opt1_05_21	56 533	526 872	2 097 393
dated-10-17-u	229 544	1 070 757	2 471 122
ACG-20-5p0	324 716	1 390 931	3 269 132
ACG-20-5p1	331 196	1 416 850	3 333 531
SPM			
powersim	15 838	15 838	67 562
as-caida	31 379	26 475	106 762
hvdc1	24 842	24 842	159 981
Ill_Stokes	20 896	20 896	191 368
mult_dcop_01	25 187	25 187	193 276
lp_pds_20	108 175	33 798	232 647
lhr14	14 270	14 270	307 858
c-61	43 618	43 618	310 016
ckt11752_dc_1	49 702	49 702	333 029
RFdevice	74 104	74 104	365 580
light_in_tissue	29 282	29 282	406 084
Andrews	60 000	60 000	760 154
2D_54019_highK	54 019	54 019	996 414
case39	40 216	40 216	1 042 160
denormal	89 400	89 400	1 156 224
2cubes_sphere	101492	101492	1647264
av41092	41 092	41 092	1 683 902
Lin	256 000	256 000	1 766 400
cfd1	70 656	70 656	1 828 364
mc2depi	525 825	525 825	2 100 225
poisson3Db	85 623	85 623	2 374 949
rgg_n_2_18_s0	262 144	262 141	3 094 566
cnr-2000	325 557	247 501	3 216 152

Similar to Sanders and Schulz [36], we use convergence plots to compare the improvement of solution quality ofver time. We choose one of the partitioning algorithms as baseline and determine the average duration t_I to partition a given instance I. We then can calculate for each absolute time stamp t of an instance I the normalized time by $t_n = \frac{t}{t_I}$. By doing so we can compare instances requiring different partition times, as well as time differences in algorithmic components for the same instance. The seperate runs on each instance are performed to generate an average solution.

We determine the average solution for an instance I at the time point t_n as follows. For each seed we create a variable a_s storing the best solution so far. These variables a_s are filled with the first solution created by the corresponding seed. Then the average over all a_s is calculated, determining the first average solution. Each time a seed s is improving its best solution at a time point t_n the corresponding value a_s is updated, and a new average is calculated with the time point t_n. This process is illustrated in Figure 4.1. The improvements are processed by ascending order of t_n. The result is a list of averaged improvements for instance I, containing tuples of the following structure $(avg(a_s), t_n)$. By ordering the list of averaged improvements by t_n we can calculate the average solution at a time point t_n.

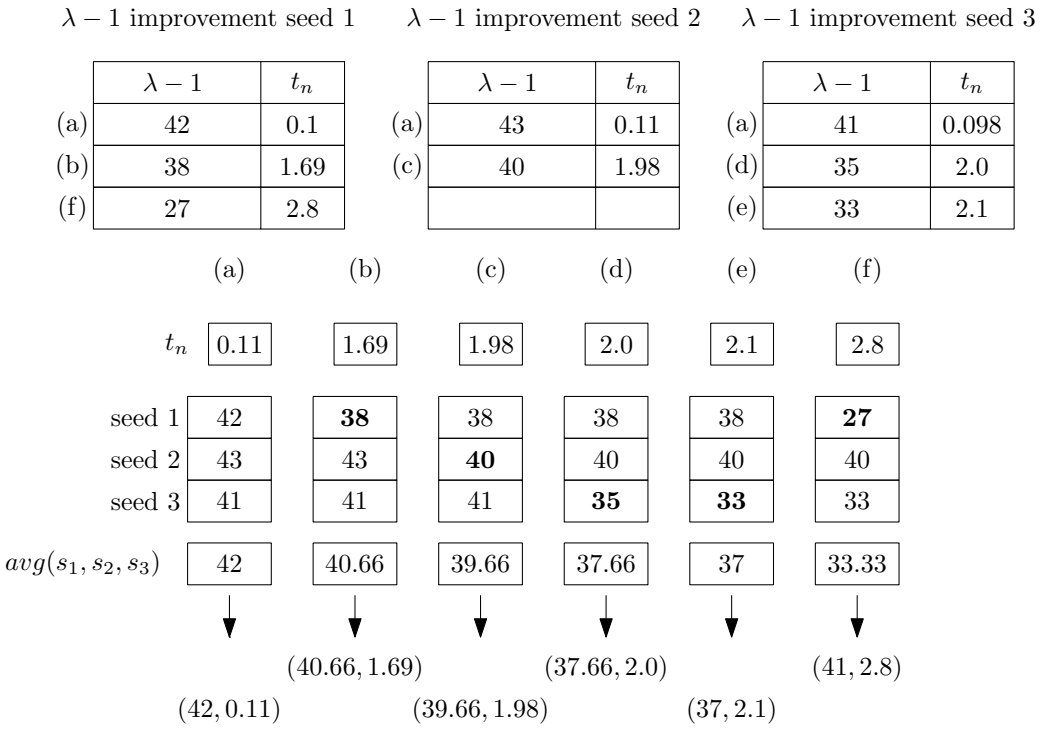

Figure 4.1: An example for averaging the seeds of an instance I

The solution improvements of the different seeds are sorted by normalized time t_n and

afterwards scanned (a)-(f). A pair of solution quality and current time is appended to the result every time an improvement is found (b)-(f). The only exception is (a) since there are no existing values to replace. In this case the first values of all seeds are used and the maximum normalized time is used for the first pair.

Next we need to average over different instances. Different instances have highly varying connectivity. To give each instance a comparable influence on the final result we use the geometric mean. Other than that the process is similar to averaging the seeds. For each instance I we create a variable a_I containing the best solution so far. We use the previously generated lists of averaged improvement l_I as input data. Following the same procedure, each time an improvement is found in one of the l_I the corresponding value a_I is updated and a tuple $(geoMean(a_I), t_n)$ is appended to the final result. An example for this procedure can be found in Figure 4.2

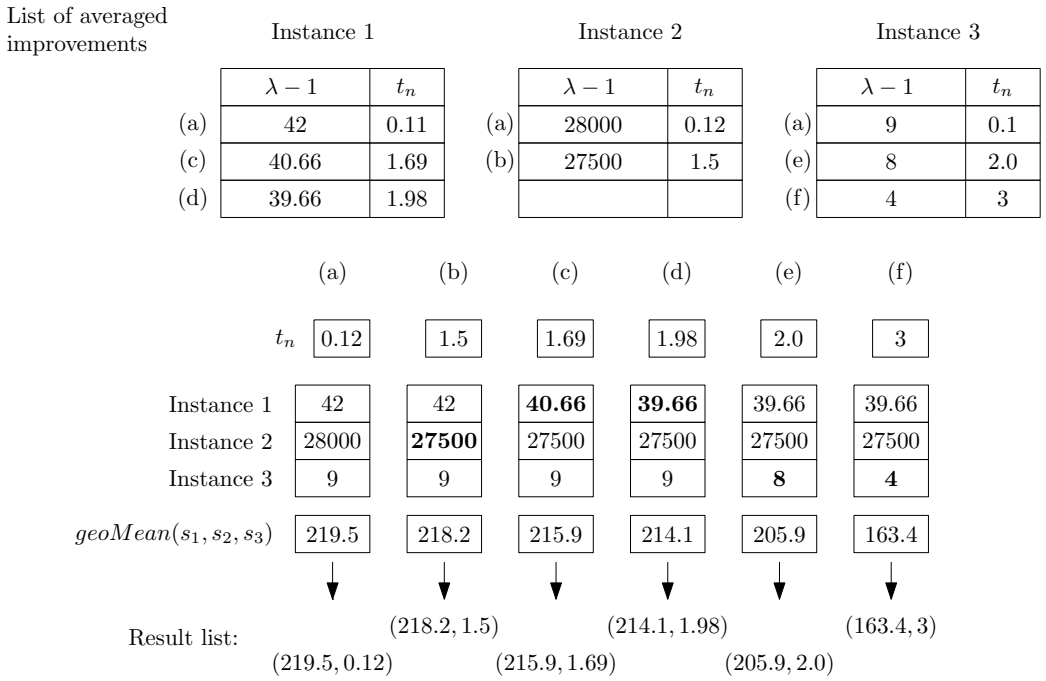

Figure 4.2: An example for averaging multiple instances I

I Similar to seed averaging the lists of averaged improvement are processed in order of increasing normalized time t_n (a)-(f). Then a pair of solution quality and current time is appended to the result every time an improvement is found (b)-(f). The only exception is (a) since there are no existing values to replace. In this case the first values of all seeds are used and the maximum normalized time is used for the first pair.

4.3 Evaluation of Algorithmic components

All experiments in this section are based on the tuning subset. The experiments were run on an Ubuntu 14.04 machine with four Intel Xeon E5-4640 Octa-Core processors with 2.4 GHz, 512 GB main memory, 20 MB L3- and 8x256 KB L2-Cache. The following parameters are used as default: Dynamic population size using $\delta = 0.15$ and $[3, 50]$ as lower/upper bound of the population size, $\gamma = 0.5$ as dampening factor for C2, C2 and M3 are using $\sqrt{|P|}$ as default number of individuals. The threshold of M3 is set to 0.75. The parameters of KaHyPar are set to the default configuration for direct k-way partitioning. [1]

4.3.1 Replacement Strategy

We begin by evaluating the different replacement strategies presented in Section 3.3. KaHyPar-E uses operator C1 as the only evolutionary action.

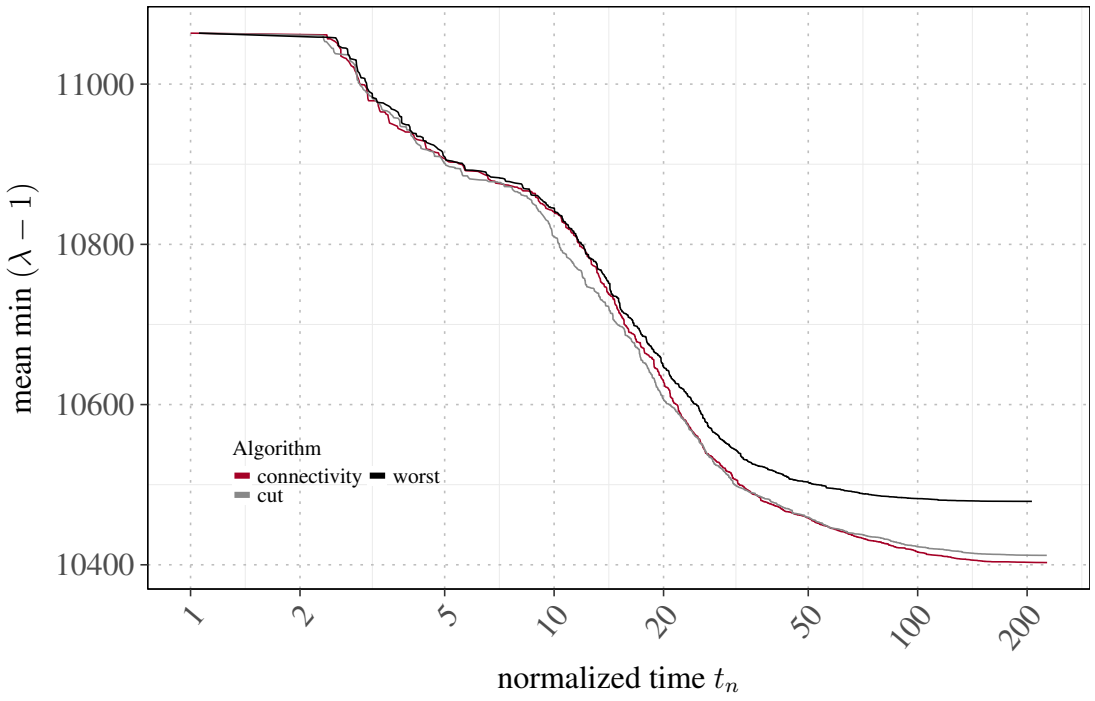

Figure 4.3: Using different replacement strategies

In Figure 4.3 we compare the effectiveness of the three different replacement strategies. As expected in Section 3.7 replacing the worst individual in the population leads to premature

[1] https://github.com/SebastianSchlag/kahypar/blob/master/config/km1_direct_kway_sea17.ini

convergence and should be avoided. Using the diversity replacement for graph partitioning [36] from Section 3.3 shows that trying to maintain diversity prevents early plateauing and is thus capable of generating better solutions. However the connectivity approach for diversity results in a slightly better solution quality. This confirms the assumption formulated in Section 3.3 that connectivity difference is a more appropriate approach in expressing different characteristics of two partitions. Comparing connectivity replacement against worst replacement results in $Z = 3.14$ with error margin $p = 0.0017$. Based on these results we will use connectivity difference as default replacement strategy in all upcoming evaluations.

4.3.2 Combine Operator

Next we evaluate the different combine operators and compare the solution quality with the solution quality of K_N-CA and K_N-CA-V. We evaluate two configurations of the evolutionary algorithm. The specification K_E+C$_x$ means that the configuration uses the combine operator C$_x$. The configuration K_E+C$_1$+C$_2$ chooses one of the two combine strategies uniformly at random for each iteration.

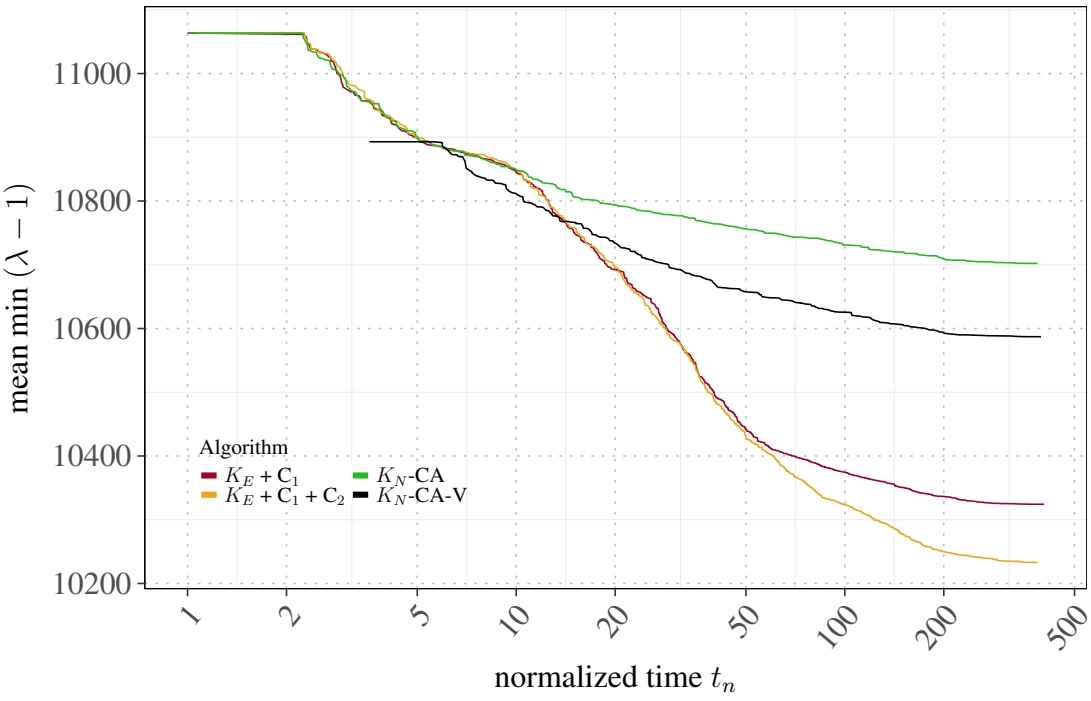

Figure 4.4: Comparing KaHyPar-E to KaHyPar

In Figure 4.4 we evaluate the results of KaHyPar-E using only basic combines as evolutionary operation against repeated repetitions of the nonevolutionary algorithms K_N-CA and

K_N-CA-V. The plot lines of K_N-CA, K_E+C_1 and K_E+C_1+C_2 are nearly identical up to the time point of 10 t_n normalized time. This is due to the fact that KaHyPar-E is using K_N-CA to generate the initial population. The minor fluctuations are caused by the shared memory parallelization explained in the experimental setup 4.1. However the values generated are the same since K_N-CA is configured with the same seed during each of those experiments. K_N-CA-V is not sharing the same starting curve. Since V-cycles are time consuming the algorithm steps of K_N-CA-V are slower than the steps of K_N-CA, resulting in an offset of the starting point for the plot line. As expected K_N-CA-V produces better solutions than K_N-CA, which also extends to repeated repetitions. Both variations of KaHyPar-E gain a significant amount of improvement after generating the initial population. This is due to the combine schemes being able to exploit structural improvements of different partitions as described in 3.5.1, allowing for a more efficient traversal of the solution space. However K_E + C_1 is eventually converging into a local optima since the combine operation is causing convergence and reduces diversity. The multicombine operation C2 however is not causing premature convergence. This operator is profiting from a stable population generated by the operator C1 in a sense that the best solutions in the population can most likely be considered good solutions. This is visible in the plot since K_E+C_1+C_2 is not drastically different from K_E+C_1 in the beginning but allows for an improvement of solution quality when K_E+C_1 is already plateauing. Comparing K_E+C_1 with K_N-CA-V generates a Z-Value of 4.37 ($p = 0.000012$) indicating that K_E+C_1 is computing better solutions. Comparing K_E+C_1 with K_E+C_1+C_2 results in $Z = 2.58$ and $p = 0.0098$.

4.3.3 Mutation Operator

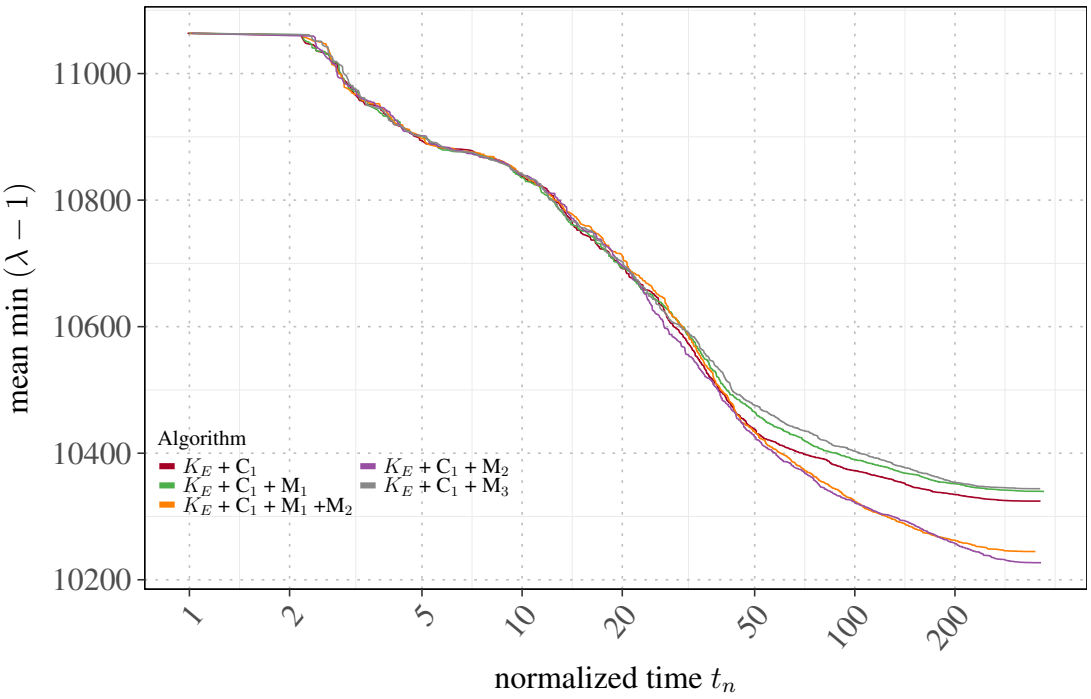

Figure 4.5: Different Mutation Operations

In Figure 4.5 the effectiveness of the different mutation operators is evaluated. KaHyPar-E uses a 50% chance of C1 and a 50% chance of the respective mutation operations. In the specific case M1 + M2 the mutation operation performed is selected uniformly at random. Adding mutation operator M1 to the already existing combine operator is in fact performing worse. This is due to the fact that V-cycles share the same quality assurance as C1 and are thus incapable of preventing premature convergence and introducing diversity. Individuals that have been optimized during the execution of the algorithm also often have been improved by V-cycles or already have a good enough quality so that V-cycles cannot find improvements. In conclusion this means that V-cycles alone are unable to prevent premature convergence. In contrast using V-cycles with new initial partitioning M2 or a combination of both mutation operators will generate better solutions. Since M2 is not limited by the quality assurance of C1 and M1, worse solutions can be created and the solution space can be explored more effectively. Stable net detection M3 is generating worse solutions or using up more time to generate individuals and therefore not further inspected in upcoming experiments. Interestingly when considering how often the different mutation operations have been able to generate a new best solution M3 has only been able to do so for 2 instances out of 25 whereas any combination of M1 and M2 have created a new best

solution in all 25 instances. This suggests that M3 is an operation depending on the hypergraph structure and not easily applicable to a generic hypergraph. Due to the replacement strategy newly generated solutions are only considered for insertion if the solution quality is better than the worst individual in the population and will lead to convergence regardless of which mutation operators are applied.

4.3.4 Mutation Replacement

Mutated individuals can be inserted into the population in two different ways. Replacing the element used for the mutation in the classical sense of evolutionary algorithms, or using the diversity replacement approach. We evaluate whether the different replacement strategies are influencing the solution quality.

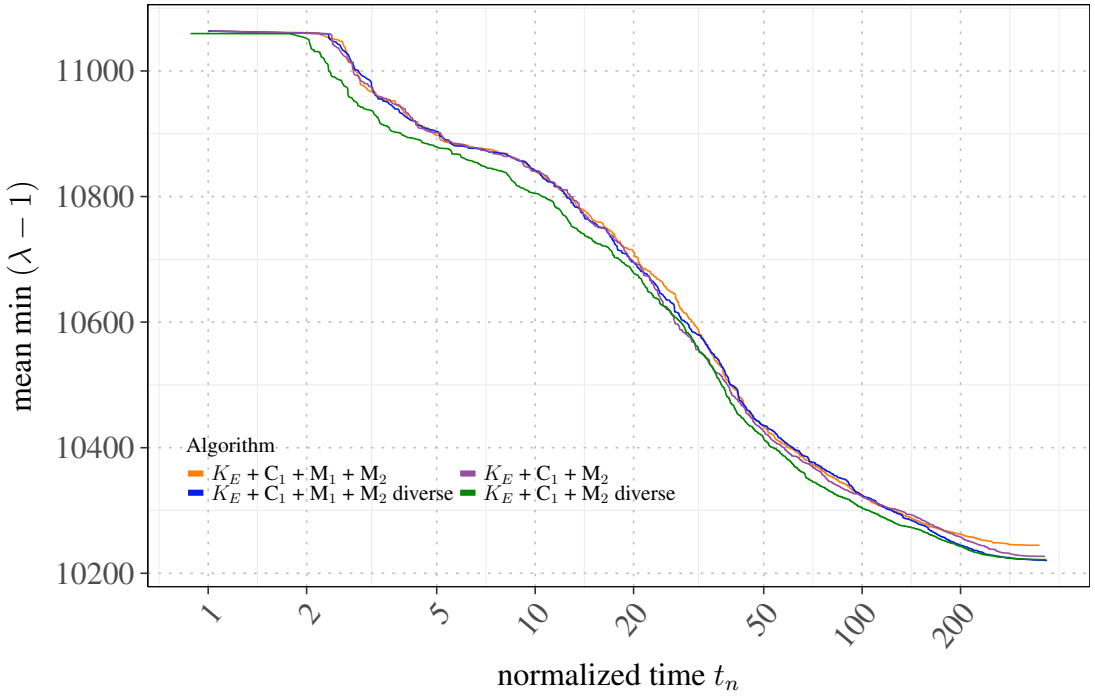

Figure 4.6: Different replacement strategies for mutations

As seen in Figure 4.6 the different replacement strategies are displayed for M2 and a combination of M1 and M2. While the difference in K_E + C$_1$ + M$_2$ and K_E + C$_1$ + M$_2$ diverse is only a minuscule improvement in convergence time and solution quality, the difference of K_E + C$_1$ + M$_1$ + M$_2$ and K_E + C$_1$ + M$_1$ + M$_2$ + diverse is more prominent in terms of solution quality. Comparing the diverse approach with the basic approach results in $Z = 2.18, p = 0.029$ for K_E + C$_1$ + M$_1$ + M$_2$ and $Z = 2.47, p = 0.013$ for K_E + C$_1$ + M$_2$.

4.4 Tuning Parameters

KaHyPar-E is adding the following parameters to KaHyPar: dynamic population size $\delta = 0.15$, upper and lower bound limits for the population size $[3, 50]$, the edge frequency dampening parameter $\gamma = 0.5$ as well as the amount of individuals used by C2 & M3 $\mathcal{N} = \sqrt{|P|}$. Additional parameters are the replacement strategy, as well as the distribution of combine operators and mutation operators. Most importantly, KaHyPar-E adds the time limit t.

4.4.1 Configuring Combine Operators

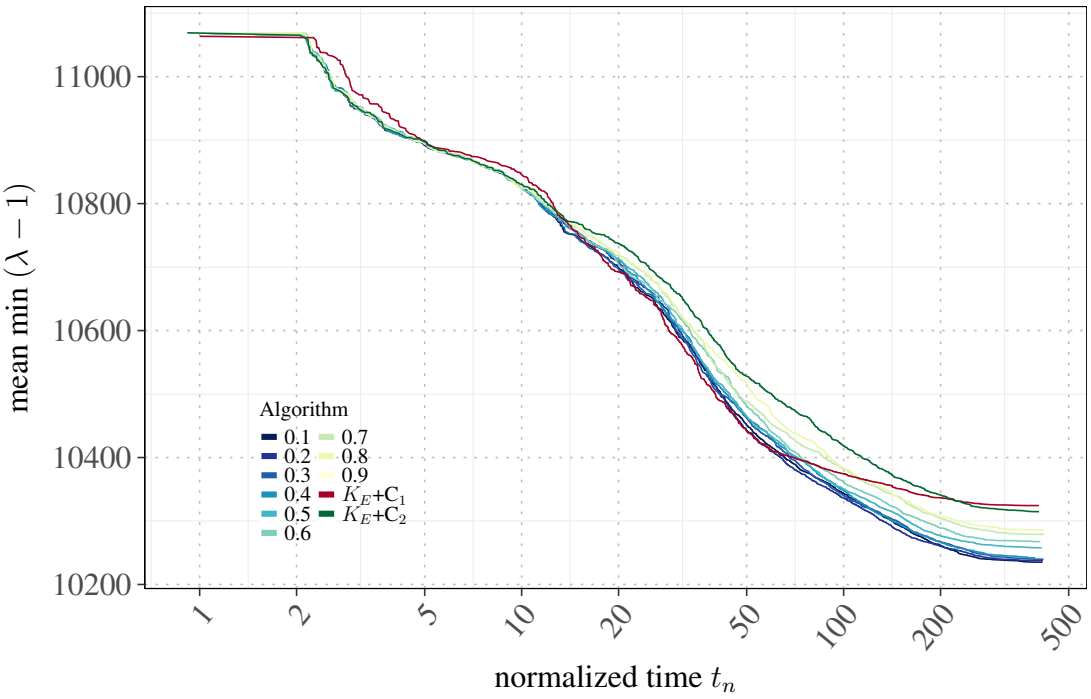

Figure 4.7: Chances of edge frequency combines

As seen in Section 4.3.2 combining operators C1 + C2 produces better results than using each operator alone. We therefore analyze how the fraction of operator distribution affects solution qualiy. In Figure 4.7 the chances of performing an edge frequency combine C2 instead of a basic combine C1 are displayed. Clearly recognizable is that a proper application of both operators will lead to improvements. The optimal values are in a range from 20% to 50%, however similar no significant difference can be observed between the tuned values. We choose 50% as distribution parameter when using both combine operators in an algorithm configuration.

4.4.2 Configuring Mutation Operators

Next we determine an appropriate ratio for choosing combine and mutation operations. We use K_E + C_1 + M_2 for tuning this parameter.

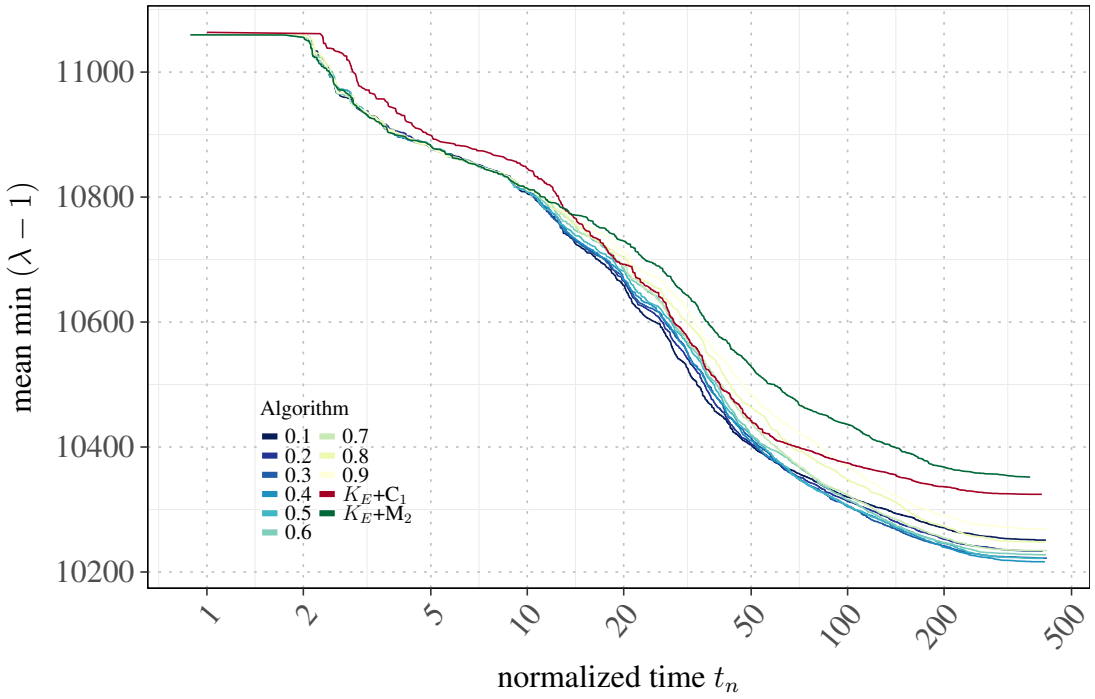

Figure 4.8: New initial partitioning mutation chance

Figure 4.8 shows the different mutation ratios. The percentages represent the chance of performing a new initial partitioning V-cycle. If not performing a new initial partitioning a basic combine is performed. It is visible that choosing either 0% mutation chance or 100% mutation chance are both not viable for generating good solutions. A combination of both operators is increasing solution quality. As seen in this experiment a mutation chance of 30% -50% for new initial partitioning is generating the best solutions. This is drastically diverging from the chance used in evolutionary graph partitioning which is around 10% [36]. We choose a ratio of 50% between combine operators and mutation operators.

4.5 Final Evaluation

Now we use the results form the tuning subset and transfer them on the benchmark subset. The following data sets are evaluated on the benchmark subset: K_N-CA, K_N-CA-V, K_E +

C_1, K_E + C_1 + C_2 and K_E + C_1 + M_1 + M_2. The experiments were performed on a cluster consisting of 512 16-way Intel Xeon compute nodes. All nodes contain two Octa-core Intel Xeon processors E5-2670 (Sandy Bridge) @ 2.6 GHz and have 8x256 KB of level 2 cache and 20 MB level 3 cache. Each node has 64 GB of main memory. The nodes were allocated exclusively to avoid CPU time interference. K_E + C_1 + C_2 is using C1 or C2 with a probability of 50%. K_E + C_1 + M_1 + M_2 is choosing between a combine operation or a mutation operation with 50% chance. The mutation strategy is selected uniformly at random.

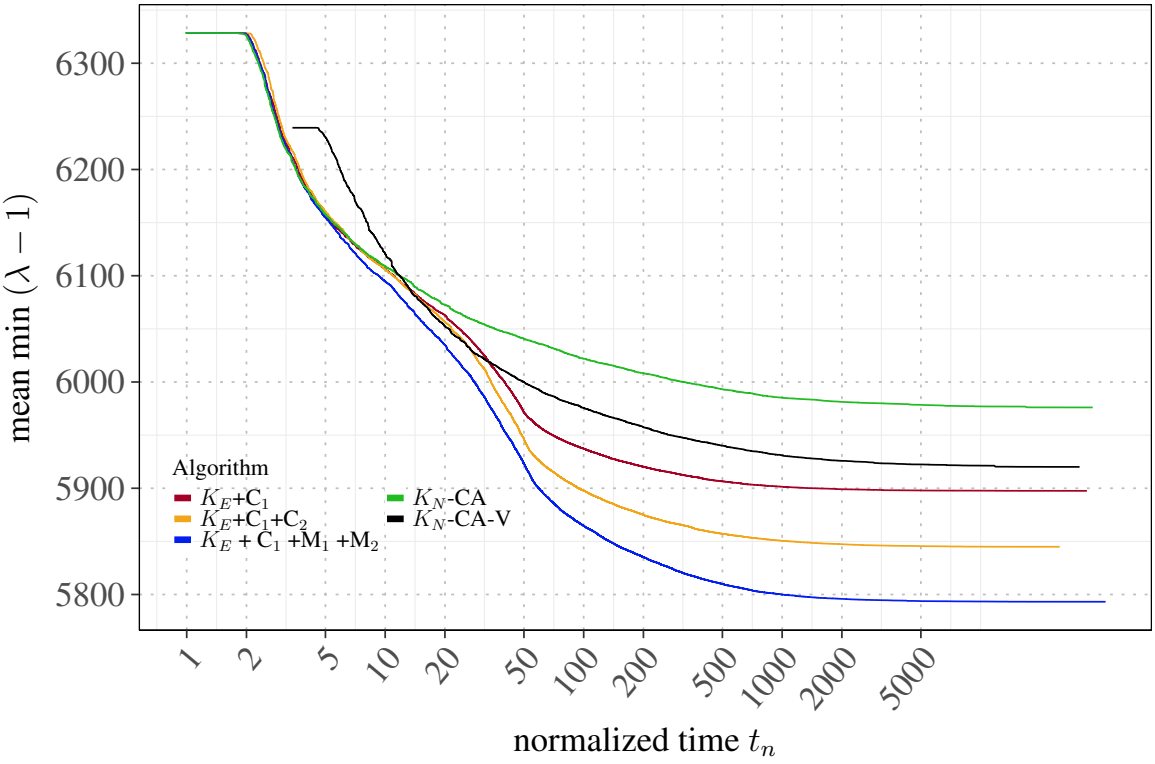

Figure 4.9: Benchmark subset results

Similarly to the results on the tuning subset, it is noticeable that the evolutionary algorithms K_E are performing better than the nonevolutionary counterparts K_N. K_N-CA-V is again requiring more time for an iteration, resulting in a difference of the starting points of the plot. Comparing the evolutionary algorithms with the nonevolutionary algorithms results in a Z-Value of 14.97 when comparing K_E + C_1 + C_2 against K_N-CA-V and a Z-Value of 20.11 when comparing K_E + C_1 + M_1 + M_2 against K_N-CA-V. The error margin is $p \approx 0$, being too small to be expressed by 32 bit floating point representation. The concluding statement is that both algorithm configurations are generating better solutions than the nonevolutionary algorithms.

Table 4.3: Connectivity improvement of the strongest configurations for KaHyPar-E

	K_E + C_1 + C_2		K_E + C_1 + M_1 + M_2	
k	K_N-CA-V	K_N-CA	K_N-CA-V	K_N-CA
all k	1.7%	2.7%	2.2%	3.2%
2	-0.2%	0.4%	0.2%	0.8%
4	-0.2%	0.3%	0.9%	1.3%
8	0.7%	1.6%	1.9%	2.7%
16	1.9%	2.8%	2.6%	3.5%
32	2.9%	3.9%	3.2%	4.2%
64	3.2%	4.7%	3.4%	4.8%
128	3.3%	5.0%	3.3%	5.0%

In Table 4.3 the average improvements of the two strongest configurations of KaHyPar-E are compared to KaHyPar-CA-V and KaHyPar-CA. The average best improvement is 2.2% when using K_E + C_1 + M_1 + M_2. As seen in Figure 4.10 the quality gains increase with growing k. The strongest configuration K_E + C_1 + M_1 + M_2 produces better solutions than K_N-CA-V in 597 out of the 630 instances.

As seen in Figure 4.10 the solution quality of the evolutionary algorithms K_E is quite similar to the nonevolutionary algorithms K_N for small values of k. In fact the evolutionary algorithms are sometimes performing worse. With increasing k the solution quality gains of the evolutionary algorithms are growing. Additionally K_E + C_1 + C_2 is approaching K_E + C_1 + M_1 + M_2 and K_N-CA-V is diverging from K_N-CA. This suggests that KaHyPar is generating good solutions for small values of k, but allows for improvement if k is large. The increasing number of blocks decrease the benefit of local search during refinement. However the evolutionary framework is still capable of improving the solution quality, so do V-cycles. The edge frequency operator additionally seems to thrive with increasing problem complexity.

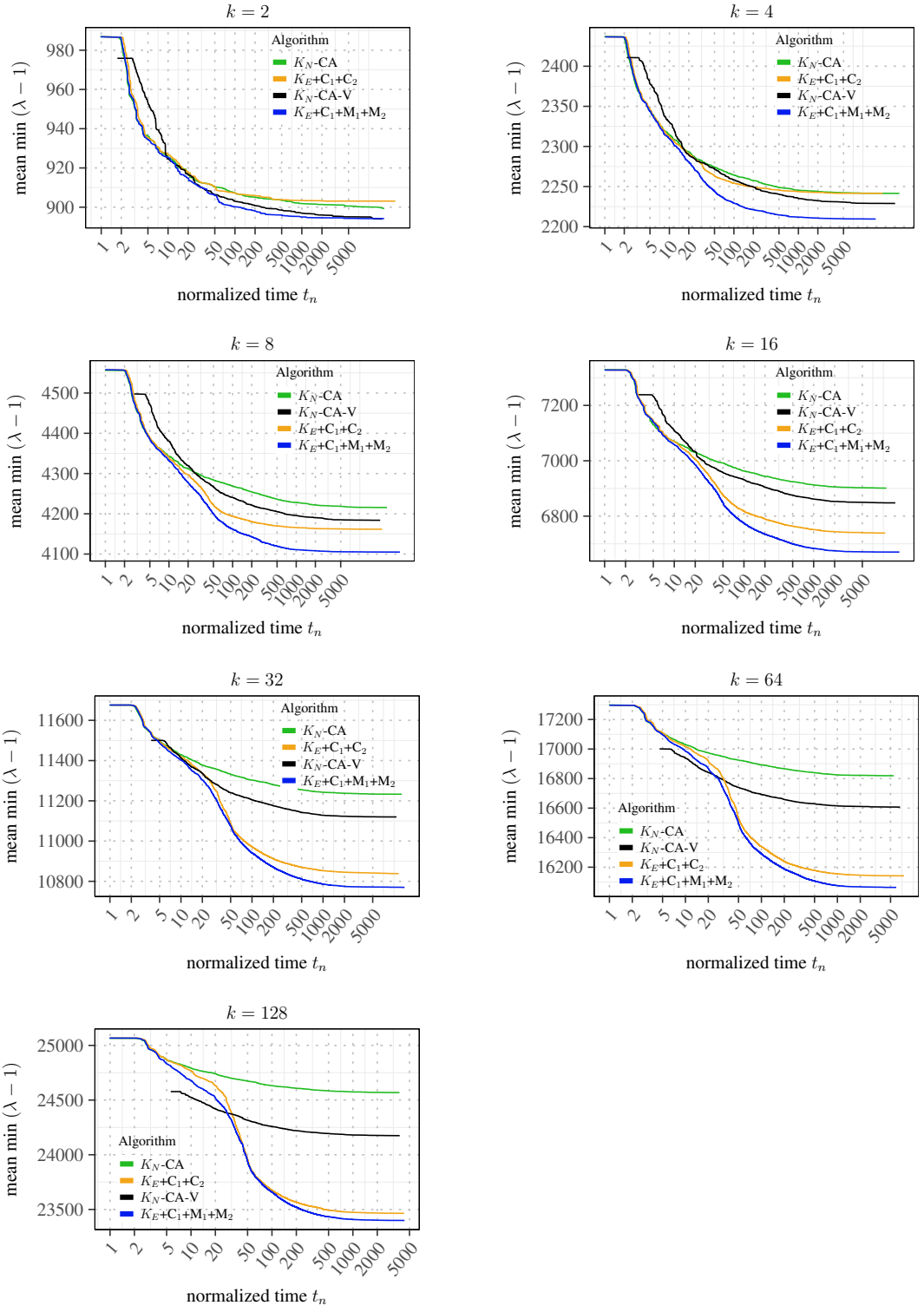

Figure 4.10: Benchmark subset split by k

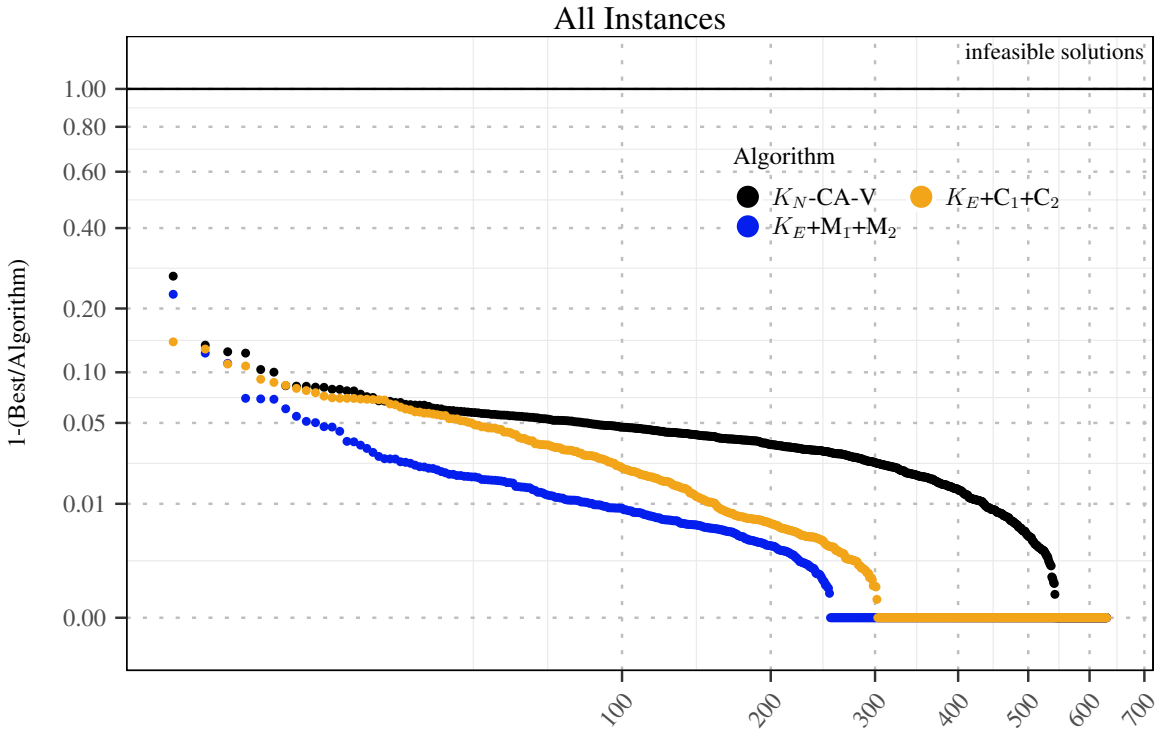

Figure 4.11: Performance Plot over all instances

In Figure 4.11 we analyze the performance of the algorithms using the method presented in [37]. For each separate instance we consider the best solution of all data sets $Best$ and display the resulting quotient of $1 - \frac{Best}{Algorithm}$ sorted by descending order. As expected K_E + C$_1$ + M$_1$ + M$_2$ is outperforming the other configurations and both evolutionary algorithms are closer to the best solution than K_N-CA-V. However when analyzing by graph class in Figure 4.12 K_E + C$_1$ +C$_2$ is outperforming K_E + C$_1$ + M$_1$ + M$_2$ on the ISPD98 instances whereas K_E + C$_1$ + M$_1$ + M$_2$ is performing better on the DAC2012 instances and SAT14dual instances. The analysis suggests that edge frequency is more useful on instances with few big hyperedges mixed with small hyperedges, but is underperforming in hypergraphs with a great amount of large hyperedges and hypergraphs with many hyperedes in correspondence to the amount of nodes.

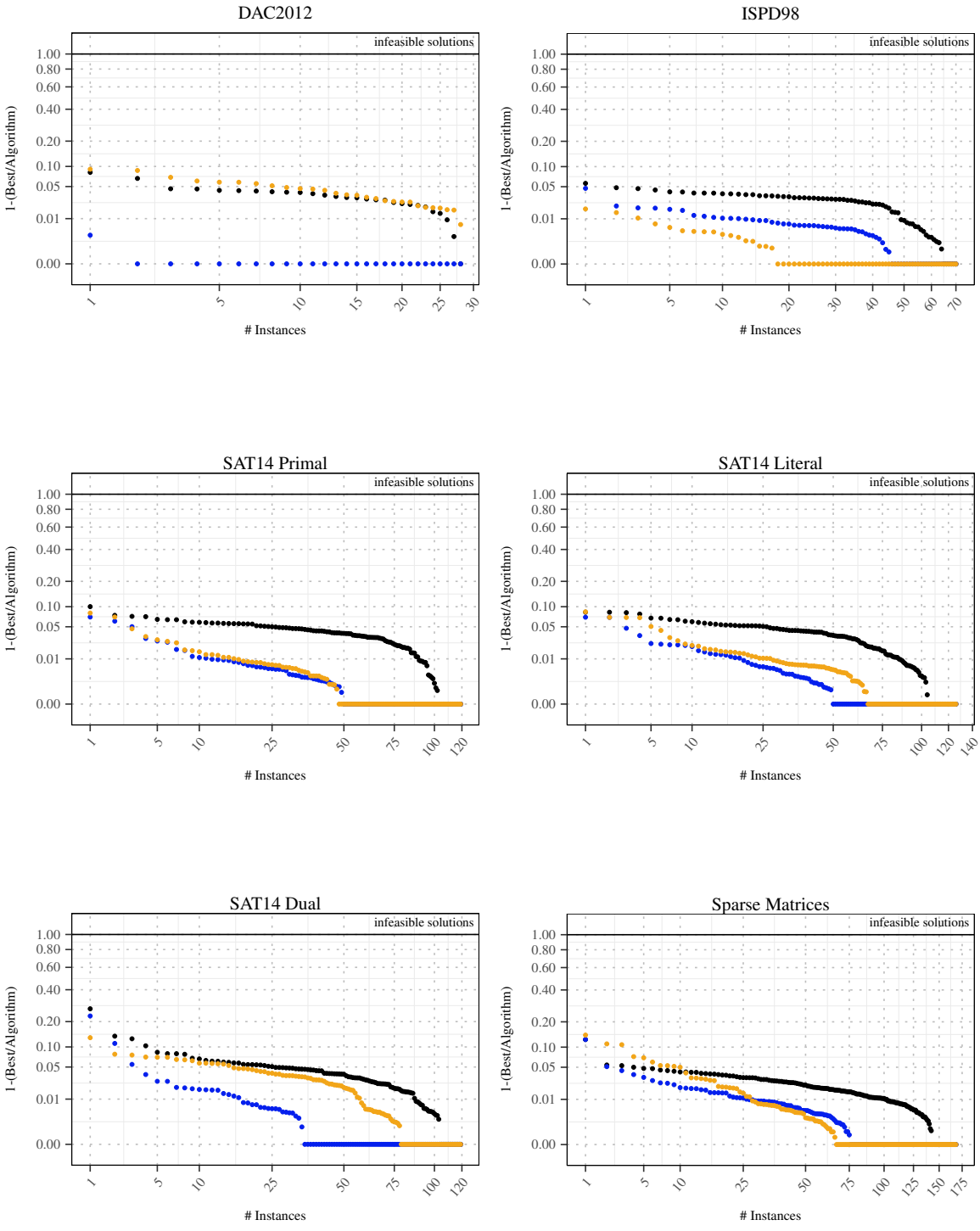

Figure 4.12: Performance Plot split by instance class

5 Discussion

5.1 Conclusion

This thesis presents an evolutionary framework for KaHyPar resulting in a quality improvement of up to 5%. We used combine operators different from usual crossover approaches to detect and exploit good solution qualities in partitions, as well as mutation operations to increase the solution scope compared to KaHyPar. Additionally we created a diversity strategy suited for hypergraph. Our operators are heavily integrated into the standard procedure of KaHyPar, to the point where all operators make use of the multilevel partition steps provided by KaHyPar. To the best of our knowledge this work is the first combination of multilevel and memetic algorithms in the field of hypergraph partitioning. As expected of an evolutionary algorithm, the quality improvement needs multiple iterations to show significance. KaHyPar-E was designed with that mentality to improve the best possible solution for the partition of a hypergraph where the time constraint is of secondary relevance.

5.2 Future Work

KaHyPar-E can be augmented using a distributed implementation similar to KaffPaE [36]. Adding a layer of parallelization would allow a significant speedup by creating multiple individuals during an iteration. Another interesting approach is a time cost analysis for the different operators. There is a strong indication that the basic combine operator is significantly faster than a regular iteration in KaHyPar. If this indication is true, a faster population generation would be a valid approach to increase performance. Also the V-cycle mutation operator might turn out to be more beneficial if the number of cycles is increased. Other than that more sophisticated selection strategies for parent selection as well as edge frequency might also be helpful. A more detailed analysis and exploitation of hypergraph structures may enhance the performance of the current evolutionary operators, or perhaps even inspire the addition of new operators.

Bibliography

[1] AKHREMTSEV, YAROSLAV, TOBIAS HEUER, PETER SANDERS SEBASTIAN SCHLAG: *Engineering a direct k-way Hypergraph Partitioning Algorithm. 2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, 28–42. SIAM, 2017.

[2] ALPERT, CHARLES J: *The ISPD98 circuit benchmark suite. Proceedings of the 1998 international symposium on Physical design*, 80–85. ACM, 1998.

[3] ALPERT, CHARLES J, JEN-HSIN HUANG ANDREW B KAHNG: *Multilevel circuit partitioning*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 17(8):655–667, 1998.

[4] ALPERT, CHARLES J ANDREW B KAHNG: *Recent directions in netlist partitioning: a survey*. Integration, the VLSI journal, 19(1-2):1–81, 1995.

[5] AREIBI, SHAWKI: *An integrated genetic algorithm with dynamic hill climbing for VLSI circuit partitioning. GECCO 2000*, 97–102, 2000.

[6] ARMSTRONG, ED, G GREWAL, SHAWKI AREIBI GERARDA DARLINGTON: *An investigation of parallel memetic algorithms for VLSI circuit partitioning on multi-core computers. Electrical and Computer Engineering (CCECE), 2010 23rd Canadian Conference on*, 1–6. IEEE, 2010.

[7] AYKANAT, CEVDET, B BARLA CAMBAZOGLU BORA UÇAR: *Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices*. Journal of Parallel and Distributed Computing, 68(5):609–625, 2008.

[8] BACK, THOMAS: *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[9] BADER, DAVID A, HENNING MEYERHENKE, PETER SANDERS DOROTHEA WAGNER: *Graph partitioning and graph clustering*, 588. American Mathematical Soc., 2013.

[10] BELOV, ANTON, DANIEL DIEPOLD, MARIJN JH HEULE MATTI JÄRVISALO: *Proceedings of SAT Competition 2014*. 2014.

[11] BLICKLE, TOBIAS LOTHAR THIELE: *A comparison of selection schemes used in evolutionary algorithms*. Evolutionary Computation, 4(4):361–394, 1996.

[12] BUI, THANG NGUYEN BYUNG RO MOON: *A fast and stable hybrid genetic algorithm for the ratio-cut partitioning problem on hypergraphs. Proceedings of the 31st annual Design Automation Conference*, 664–669. ACM, 1994.

[13] BULUÇ, AYDIN, HENNING MEYERHENKE, ILYA SAFRO, PETER SANDERS CHRISTIAN SCHULZ: *Recent advances in graph partitioning*. *Algorithm Engineering*, 117–158. Springer, 2016.

[14] CATALYUREK, UMIT V CEVDET AYKANAT: *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*. IEEE Transactions on parallel and distributed systems, 10(7):673–693, 1999.

[15] CHAN, HEMING PINAKI MAZUMDER: *A systolic architecture for high speed hypergraph partitioning using a genetic algorithm*. Progress in evolutionary computation, 109–126, 1995.

[16] CHEN, TIANSHI, KE TANG, GUOLIANG CHEN XIN YAO: *A large population size can be unhelpful in evolutionary algorithms*. Theoretical Computer Science, 436:54–70, 2012.

[17] DAVIS, TIMOTHY A YIFAN HU: *The University of Florida sparse matrix collection*. ACM Transactions on Mathematical Software (TOMS), 38(1):1, 2011.

[18] DELLING, DANIEL, ANDREW V GOLDBERG, ILYA RAZENSHTEYN RENATO F WERNECK: *Graph partitioning with natural cuts*. *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 1135–1146. IEEE, 2011.

[19] DEVINE, KAREN D, ERIK G BOMAN, ROBERT T HEAPHY, ROB H BISSELING UMIT V CATALYUREK: *Parallel hypergraph partitioning for scientific computing*. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 10–pp. IEEE, 2006.

[20] FIDUCCIA, CHARLES M ROBERT M MATTHEYSES: *A linear-time heuristic for improving network partitions*. *Papers on Twenty-five years of electronic design automation*, 241–247. ACM, 1988.

[21] GAREY, MICHAEL R DAVID S JOHNSON: *Computers and intractability*, 29. wh freeman New York, 2002.

[22] HEUER, TOBIAS SEBASTIAN SCHLAG: *Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure*. *LIPIcs-Leibniz International Proceedings in Informatics*, 75. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[23] HOLTGREWE, MANUEL, PETER SANDERS CHRISTIAN SCHULZ: *Engineering a scalable high quality graph partitioner*. *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 1–12. IEEE, 2010.

[24] HULIN, MARTIN: *Circuit partitioning with genetic algorithms using a coding scheme to preserve the structure of a circuit*. *International Conference on Parallel Problem Solving from Nature*, 75–79. Springer, 1990.

[25] KABILJO, IGOR, BRIAN KARRER, MAYANK PUNDIR, SERGEY PUPYREV ALON SHALITA: *Social hash partitioner: a scalable distributed hypergraph partitioner*. Proceedings of the VLDB Endowment, 10(11):1418–1429, 2017.

[26] KARYPIS, GEORGE, RAJAT AGGARWAL, VIPIN KUMAR SHASHI SHEKHAR: *Multilevel hypergraph partitioning: applications in VLSI domain*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 7(1):69–79, 1999.

[27] KARYPIS, GEORGE VIPIN KUMAR: *Multilevel k-way hypergraph partitioning*. VLSI design, 11(3):285–300, 2000.

[28] KIM, JONG-PIL, YONG-HYUK KIM BYUNG-RO MOON: *A hybrid genetic approach for circuit bipartitioning*. *Genetic and Evolutionary Computation–GECCO 2004*, 1054–1064. Springer, 2004.

[29] KIM, YONG-HYUK BYUNG-RO MOON: *Lock-gain based graph partitioning*. Journal of Heuristics, 10(1):37–57, 2004.

[30] LIM, SUNG KYU, DONGMIN XU .: *Large scale circuit partitioning with loose/stable net removal and signal flow based clustering*. *Computer-Aided Design, 1997. Digest of Technical Papers., 1997 IEEE/ACM International Conference on*, 441–446. IEEE, 1997.

[31] MOSCATO, PABLO .: *On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms*. Caltech concurrent computation program, C3P Report, 826:1989, 1989.

[32] PAPA, DAVID A IGOR L MARKOV: *Hypergraph Partitioning and Clustering.*, 2007.

[33] SAAB, YOUSSEF VASANT RAO: *An evolution-based approach to partitioning ASIC systems*. *Proceedings of the 26th ACM/IEEE design automation conference*, 767–770. ACM, 1989.

[34] SAIT, SADIQ M, AIMAN H EL-MALEH RASLAN H AL-ABAJI: *Evolutionary algorithms for VLSI multi-objective netlist partitioning*. Engineering applications of artificial intelligence, 19(3):257–268, 2006.

[35] SANCHIS, LAURA A: *Multiple-way network partitioning*. IEEE Transactions on Computers, 38(1):62–81, 1989.

[36] SANDERS, PETER CHRISTIAN SCHULZ: *Distributed evolutionary graph partitioning*. *2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, 16–29. SIAM, 2012.

[37] SCHLAG, SEBASTIAN, VITALI HENNE, TOBIAS HEUER, HENNING MEYERHENKE, PETER SANDERS CHRISTIAN SCHULZ: *k-way Hypergraph Partitioning via n-Level Recursive Bisection*. *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, 53–67. SIAM, 2016.

[38] TRIFUNOVIC, ALEKSANDAR: *Parallel algorithms for hypergraph partitioning*. , University of London, 2006.

[39] VASTENHOUW, BRENDAN ROB H BISSELING: *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*. SIAM review, 47(1):67–95, 2005.

[40] VISWANATHAN, NATARAJAN, CHARLES ALPERT, CLIFF SZE, ZHUO LI YAOGUANG WEI: *The DAC 2012 routability-driven placement contest and bench-*

mark suite. *Proceedings of the 49th Annual Design Automation Conference*, 774–782. ACM, 2012.

[41] WICHLUND, SVERRE: *On multilevel circuit partitioning*. *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, 505–511. ACM, 1998.

[42] WILCOXON, FRANK: *Individual comparisons by ranking methods*. Biometrics bulletin, 1(6):80–83, 1945.