Karlsruher Institut für Technologie

Bachelor thesis

Evolutionary Hypergraph Partitioning

Robin Andre

Date: 9. November 2017

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

Institute of Theoretical Informatics, Algorithmics
Department of Informatics

Abstract

In this thesis we augment the existing Hypergraph Partitioner KaHyPar with an evolutionary framework with the goal to improve the solution quality.

Acknowledgments

I’d like to thank Timo for the supply of Club-Mate

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

A Hypergraph is a generalization from a regular graph in the regard that the edges may contain more than 2 vertices. Many real world scenarios like warehouse article limits, social networks and structures like computer chips can be modeled accurately as a hypergraph but only imprecise as a graph. As a result there are many applications to partition a hypergraph into k blocks in order to analyze or optimize the circumstances translated towards their respective mathematical model. The objective is to minimize the cut between the blocks. Primarily VLSI design is benefitting from hypergraph partitioning in a sense that the amount of interconnected components is bottlenecking the performance and chip size[**?**]. The size of the components also should not be severely disproportionate as this requires longer connections between the components resulting in longer transmission times. [1] Therefore the component sizes should be balanced. When trying to maintain balance for the block size hypergraph partitioning is a problem that, similar to balanced graph partitioning, is proven to be NP-hard[**?**]. Since the exact calculation for NP-hard problems is not feasible using the current academic knowledge and state-of-the-art algorithms these problems are often approached using heuristics to improve the corresponding objective while having a practicable runtime. One of the most effective heuristic in regard to graph/hypergraph partitioning is the multilevel paradigm [**?**]. By reducing the hypergraph whilst maintaining the hypergraph structure the original problem is translated to an easier problem by a large margin. Therefore a strong initial solution can be created and additionally be improved on during the recreation of the original problem using local search algorithms. While this approach is capable of generating a strong solution within one execution it falls short on further improving this solution due to the limited solution space evaluated. This problem is can be counteracted using meta heuristics such as evolutionary algorithms specifically designed to allow a more efficient traversal of the solution space and avoidance of local optima.

[1] https://www.nobelprize.org/educational/physics/integrated_circuit/history/

1.2. Contribution

This thesis will augment the existing hypergraph partitioner KaHyPar by adding an evolutionary framework designed to improve on the solution quality generated by KaHyPar. Using ideas presented in the evolutionary graph partitioner KaffPaE, we add new operators to improve the solution quality by evaluating and modifying the structures of solutions generated by KaHyPar in a more effective manner.

1.3. Structure of Thesis

In the beginning the main focus is to describe precisely how KaHyPar is utilizing the multilevel paradigm to generate the partitions. Afterwards the evolutionary framework is introduced, described and motivated and finally evaluated by comparing the solution quality of the evolutionary algorithm against KaHyPar by partitioning multiple instances using both variants.

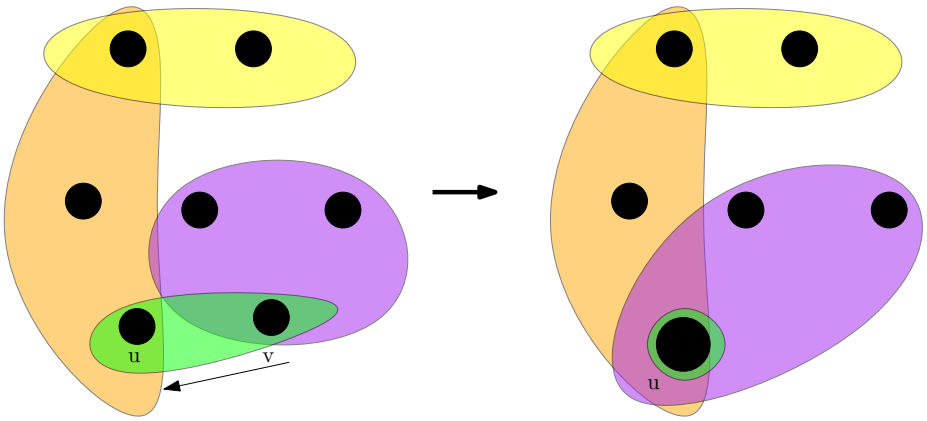


Figure 2.1.: An example of a contraction. Note that the Hyperedges of the reduced node v are rearranged to contain u.

2. Preliminaries

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 V $e \subseteq V$.

The weight of a vertex is measured by $c : V \to \mathbb{R}_{\geq 0}$. Similarly the weight of a hyperedge is defined by $w : E \to \mathbb{R}_{\geq 0}$. The set extension of c and w are defined as $c(\text{V'}) = \sum_{v \in V'} c(v)$ and $w(\text{E'}) = \sum_{e \in E'} w(e)$. Two verticies u, v are adjacent if $\exists\ e \in E \mid u, v \in e$. 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 vertices contained in e. A k-way partition of a hypergraph H is a partition of V into k disjoint blocks $V_1, .., V_k$. $part(u) : V \to [1, k]$ is a function mapping the vertex u into the corresponding partition block. A k-way partition is balanced when $\forall\ 1 \leq i \leq k \mid c(V_i) \leq (1 + \epsilon)\lceil \frac{c(V)}{k} \rceil$ for a balance constraint 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 P the connectivity set of a hyperedge e is $\Phi(e,\text{P}) := \{V_i \mid \Psi(e, V_i) > 0\}$. A hyperedge e is a cut edge in a partition P when $cut(e,\text{P}) := \Phi(e,\text{P}) > 1$. Let $\mathcal{E}$ be the set of cut edges from a partition P. The cut metric of P is defined as $cut(\text{P}) := w(\mathcal{E})$. The connectivity metric $(\lambda - 1)$ is defined as $(\lambda - 1)(\text{P}) := \sum_{e \in \mathcal{E}} (\Psi(e) - 1)w(e)$. 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 [**?**], circuit partitioning [**?**] or database storage sharding [**?**].

Two approaches are used for hypergraph partitioning. The first approach is the bisectioning of the hypergraph, where the number of blocks is fixed to $k = 2$ as in hMetis[**?**] and MLPart[**?**]. By recursively bisectioning the resulting partitions, k can assume values of the powers of 2. This is implemented in tools like PatoH [**?**], Mondrian[**?**] and Zoltan[**?**](allowing imbalanced bipartitions to create partitions of arbitrary k). 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 kMetis[**?**], kPatoH [**?**] and SHP[**?**](also implementing a recursive bisection). Note that except SHP [**?**] all tools are utilizing the multilevel paradigm.

Of course this is only a collection of the most common hypergraph partitioners, which is why we would like to refer to the survey works[**?**][**?**][**?**][**?**].

Saab and Rao [**?**] present one of the first evolutionary approaches to hypergraph partitioning by iteratively performing vertex moves comparing the gain to a random threshold using bin pacikgin and iterative improvement. Hulin [**?**] presents a genetic algorithm maintaining multiple solutions using a two dimensional representation of circuits and introduce 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 [**?**], in which solutions are preprocessed and optimized using the Fiduccia-Mattheyses [**?**] local search algorithm as well as a new replacement strategy considering the bit-wise difference of the child to the parent partitions for replacement as well as solution quality.

Chan and Mazmudner[**?**] 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 [**?**] gives another memetic algorithm for the k-way hypergraph partitioning problem. Using a variation of FM designed for k-way optimization [**?**] local search as well as a 4-point crossover operation, which splits the input partitions at 4 points and alternates between the blocks. [**?**] Kim et al. translate the lock gain local search [**?**] 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. [**?**] compare the metaheuristics 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.[**?**] are analyzing 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 [**?**] for the existing graph partitioner KaFFPa, [**?**] introducing different combination and mutation operations for graph partitioning. Their approach uses the mulitlevel paradigm in combination with evolutionary operations.

2.3. KaHyPar

The hypergraph partitioner KaHyPar additionally improves on solution quality optimizing on the connectivity metric using direct k-way partitioning [**?**] as well as the cut metric using recursive bisection[**?**]. 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 amount 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 chosen to generate an initial partitioning for the coarsened hypergraph. Afterwards contraction operations will be 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 [**?**].

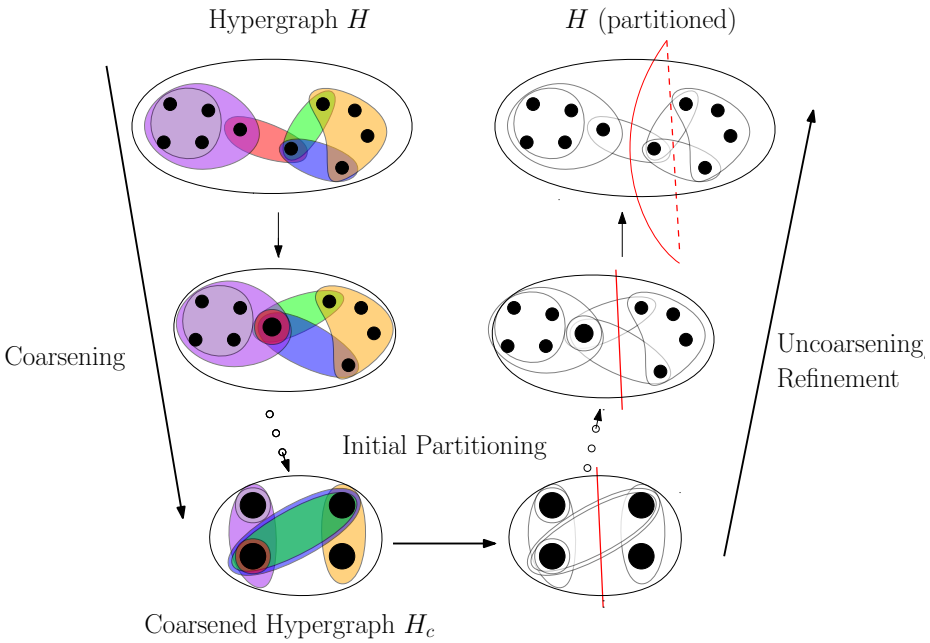


Figure 2.2.: An example of an iteration. Coarsening, Initial Partitioning and Refinement

In 2017 Heuer and Schlag improved KaHyPar by analyzing and exploiting community structures in Hypergraphs [**?**], showing that KaHyPar-CA generates solutions of superior quality compared to other established hypergraph partitioners.

3. KaHyPar-E

In this chapter we first outline the general procedure of an evolutionary algorithm. Then we transfer the hypergraph partitioning problem onto an evolutionary framework supporting the theoretical foundation. Then we introduce operators for combination and mutation as well as strategies for selection and replacement.

3.1. Overview

Evolutionary algorithms are inspired by the theory of evolution. Much like 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 timespan of multiple generations. The evolution theory states that individuals having more helpful traits like special beaks to assist in aquiring food are more likely to survive longer and thus more likely to pass these helpful traits onto the next generation. Addiditionaly some traits occur randomly through changing the genetic information erratically. These are called mutations and are even present in humans, and can be harmful like the sickle-cell desease or beneficial like the ability to consume lactose. In the 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 converting the process described above onto a mathematical problem. Therefore some initial solutions are generated and then the same steps are repeated until a stopping criterion has been met. Evolutionary algorithms are usually repeating 4 steps that try 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.

3.2. Population

The algorithm 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 [**?**]. We use KaHyPar to fill the initial population. This means that unlike most evolutionary algorithms we use high quality solutions instead of random solutions as initial population. 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 alloted time towards creating the initial population and consequently the population size is determined by $0.15 * \frac{t_{total}}{t_1}$. However we introduce lower and upper bounds for the population size to ensure a proper size for evolutionary operators and 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 healty, because the variation between the individuals ensures that no characteristic is carried by every individual and a high perturbation 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 also a cut edge in every solution.

Maintaining diversity is highly recommended[**?**], as it ensures a strong pertubation of the solutions and therefore allows for a greater exploration of the solution scope. Additionally it prolongs characteristics from manifesting throughout 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 [**?**] determines the difference of two partitions P_1, P_2 by counting the edges that are cut edges in exclusively one of the partitions $cutdiff(P_1,P_2) := \sum_{e \in E} |cut(e,P_1) - cut(e,P_2)|$. 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 $conndiff(\mathrm{P}_1,\mathrm{P}_2) := \sum_{e \in E} |\Phi(e,\mathrm{P}_1) - \Phi(e,\mathrm{P}_2)|$. This is a more natural representation for the connectivity metric as cut edges.

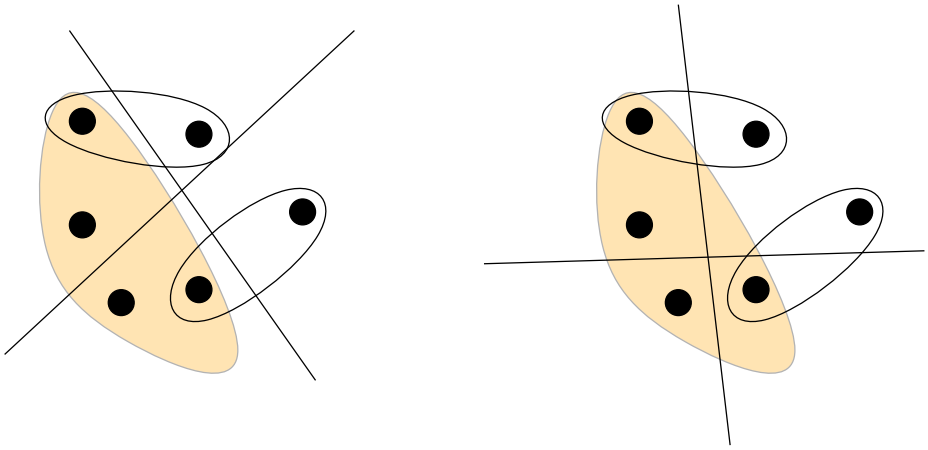


Figure 3.1.: Two different partitions of the same hypergraph

Figure 3.1 shows two different partitions of the same example hypergraph. Since all edges are a cut edge 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. close to the original individual and as result similarly good. We select our individuals using tournament selection [**?**], meaning that the individuals are competing for their chance of recombination by their fitness. In a long term perspective this ensures that good individuals get a more frequent chance of reproduction as stated by the evolutionary theory. By first selecting two random individuals and using the one with the better solution quality we can extract one individual I_1. For operators requiring two seperate Individuals we can simply repeat this step to get 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.

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 section3.7 to insert the newly generated individuals.

3.5.1. Basic Combine (C1)

The basic combine uses two parent partitions P_1 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 in both parents. Contractions performed in the same block do not modify the quality of a partition, because the block assignments cannot change, and as a result there is no fluctuation in connectivity for any hyperedge. This ensures that the solution quality for the coarsened Hypergraph H_c is the same for both parent partitions. After the coarsening we do not perform an initial partitioning. Instead we consider the coarsened hypergraph H_c and see which of the parents gives the better objective on H_c. The important part to note is that this operation is different than a V-cycle, since the coarsening condition is more strict due to the consideration of both parents partitions. The uncoarsening and application of local search algorithms which do not worsen solution quality. In combination with using the better partition of the two parents it is ensured that the child solution is at least as good as the best parent solution. The basic combine is benefitting from highly diverse parent partitions since it passes on more characteristic cut edges, resulting in a more efficient solution exploration. In terms of solution scope the operation is highly convergent due to the quality assurance and strong limitation during the coarsening.

3.5.2. Edge Frequency Multicombine (C2)

We also introduce a multi-combine operator, 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 [**?**] of e by $f(e) := \sum_{i=1}^{n} cut(e, I_i)$. We use the best $n = \sqrt{|P|}$ individuals from P for determining edge frequency as a standard parameter [**?**]. Assuming that the frequent cut edges of the best solutions are most likely a good characteristic, these edges should remain cut edges. By prolonging the contraction of nodes in these frequent cut edges, the other contractions may attach more hyperedges to those nodes. As a result the hyperedges are more probable to become cut edges Therefore we penalize contractions of nodes incident to a high frequency edge during the multilevel partition approach by using this formula

$$r(u,v) = \sum_{e \in I(u) \cap I(v)} \frac{e^{-\gamma * f(e)}}{(w(u)w(v))^{1.2}}$$

to disincentivize early contractions of nodes in edges of high probability. The power of 1.2 on the weight functions is a tuning parameter taken from [**?**] 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 during the uncoarsening refinement with local search.

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 improve current solutions to avoid population convergence towards a local optimum. We propose three different mutation operations. The first operator M1 is intended to increase 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 the 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 the refinement during the uncoarsening. Due to the randomization factor in the coarsening the structure of the coarsened hypergraph can vary and allow previously unfound improvements during local search. Using an individual 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 the quality of the solution I_{new} will have a quality at least equal to I. This is a weak mutation and the difference of I and I_{new} is small. This operation will also cause convergence, as multiple applications of a vcycle will eventually no longer improve the solution.

3.6.2. V-Cycle + New Initial Partitioning (M2)

Similar to a V-cycle we can coarsen H with a partition limiting the coarsening, but instead of immediately starting the refinement we drop the partition and perform a new initial partitioning on the coarsened hypergraph. This operation pertubes 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 introducing diversity regardless of current convergence in the population.

3.6.3. Stable Nets (M3)

Lim et al. [**?**] introduce the concept of stable net removal, stating that hyperedges that remain cutedges throughout the iterations are trapping the FM-algorithm in a local minima. Their solution is to force those edges into one block and thus from the cut. We use this approach to similarly force recurring cut edges into one block. Opposed to the 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 egde 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 number of nodes in order to maintain the balance criterion. This is done by assigning all nodes $v \in e$ the block id of the smallest block. Forcefully moved nodes may not be reassigned to another block by another stable net. These solutions have most likely siginficantly worse quality. Due to the nature of our selection strategy these solutions are very unlikely to be used in any combine operator. In order to keep these solutions competitive we also perform a vcycle 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 generated individual has 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 environment regarding the fitness of the individuals. 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 amount of the solution space. Another approach is to replace one of the elements used in the operator. This strategy was originally used for mutations, as the theory behind evolutionary algorithm mutations is to pertubate an existing solution. 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 element, instead we

replace the most similar element with a worse quality using diversity as measurement for similarity. By only replacing elements of worse quality the population is slowly increasing in quality and converging towards optima. However this approach ensures a more diverse population which boosts the combine operator effectiveness and avoids rapid convergence. In fact when considering the population slots rather than the single individual it results in the multiple small convergence lines towards several different local optima while improvements can be made and a convergence towards the best optimum as soon as all slots found a local optimum.

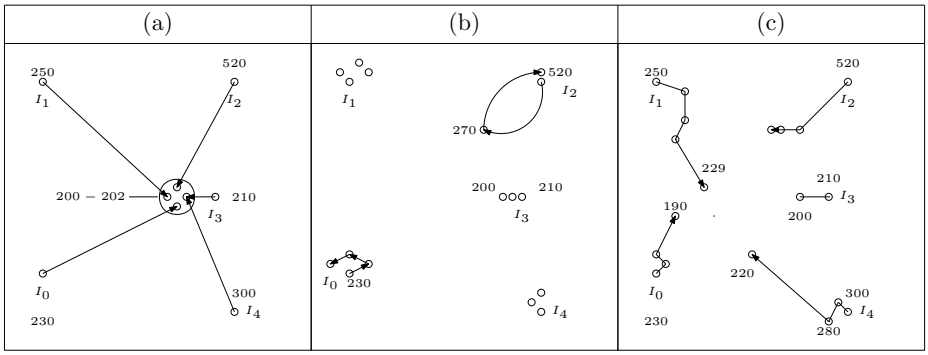


Figure 3.2.: Visual representation of different replacement strategies (a) replacing the worst element, (b) replacing the most similar element & (c) replacing similar/worse

In 3.2 the diversity of the elements is displayed as distance in the 2D plane. The arrows represent whenever an individual is replaced. In (a) a fast convergence towards the local optimum near I_3 is happening. Due to the premature convergence and strictness of the replacement strategy there it is highly unlikely that any solution will ever leave this local optimum. In (b) each individual solution is replaced more thoroughly, but good solutions are found at random and may be discarded at any moment. In (c) each of the five slots in the population is slowly converging towards its own respective local optimum, but eventually the best local optimum forces convergence. Note that this graphic is only displaying replacements. The individuals involved in creating these solutions are not linked to the replacement. i.e. I_3 at 210 might have been replaced with a solution of quality 200, but the elements creating the solution have been I_1 and I_2.

4. Experimental Evaluation

All following plots are using KaHyPar for generating an initial population and perform some of the mentioned operators.

4.1. Experimental Setup

We use two benchmark sets for evaluation. Both sets are using instances from the benchmark set of Heuer and Schlag {cite KaHyPar E, which consists of various hypergraphs from the ISPD98 benchmark [**?**], Sparse Matrices [**?**], the DAC benchmark [**?**] and Sat instances [**?**]. The sparse matrices are converted by row-net model [**?**], whereas the SAT instances are converted to hypergraphs using three approaches [] 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 were 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 were partitioned for $\epsilon = 0.03$ and 7 different values for $k = \{2, 4, 8, 16, 32, 64, 128\}$. 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 per data set produced on the benchmark subset. The reason for repeating the partitions with different starting seeds is to balance out possible outliers of the randomization. This process is described more detailed in section**??**. 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. 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 [**?**] indirectly with other hypergraph partitioning tools) We compare against KaHyPar-CA, which is an abbrevation for community aware KaHyPar [**?**], which uses communities during coarsening to increase the quality of KaHyPar. Additionally KaHyPar-CA can be improved using v-cycles as described in section **??**. Similar to [**?**] applying local search until

no improvement has been found, KaHyPar-CA can perform v-cycles until no improvement can be found. Such a stopping criterion is implemented in KaHyPar. We set the number of v-cycles to be performed in KaHyPar-CA high enough that the stopping criterion is always met $v - cycles = 100$. This algorithm configuration is called KaHyPar-CA+V. Neither KaHyPar-CA nor KaHyPar-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 KaHyPar-CA and KaHyPar-CA+V are restarted repeadeatly to ensure a proper usage of the running time.

4.2. Instances Methodology

4.3. First Evaluation

4.4. Tuning Parameters

4.5. Final Evaluation

4.6. Implementation

4.7. Experimental Setup

4.7.1. Environment

We evaluate our algorithm on two Hypergraph sets. One time for different $k = \{2, 4, 8, 16, 32, 64, 128\}$ and 174 Hypergraph Instances, repeating each run 5 times with different seeds and a algorithm runtime of 8 hours. This is referenced as benchmark subset. The other evaluation is a selection of 25 Hypergraphs using only one $k = 32$ and a runtime of 2 hours. This is referenced as tuning subset. As such our datasets contain multiple tuples of (H, k, s, t, λ) where H is the instance, s the seed, t the required time and λ the solution quality. An Instance I is the subset of (H, k, s, t, λ) where H and k are fixed. The main interest is the best solution over time compared to other partitioning algorithms. Unfortunately most graphs vary drastically in solution quality and time required to perform one iteration. We solve the time differences by not using the measured time, but rather the normalized time. In order to do so we choose one of the partitioning algorithms p as baseline and determine the average duration t_I of an iteration for each I. Afterwards for each I the normalized time t_n is calculated by $t_n = \frac{t}{t_I}$. We then generate a list for each instance containing (s, t_n, λ) sorted by t_n. Now we generate the averaged improvements for I. For each seed s a value a_s is reserved and the list is read in. When (s, t_n, λ) is better than the reserved

value for s the average over all a_s is the new value appended to the averaged improvements with$((avg(a), t_n)$. Similarly we now reserve a value a_I for each Instance of averaged improvements and calculate a new list of general improvements by calculating the geometric mean over all a_I.

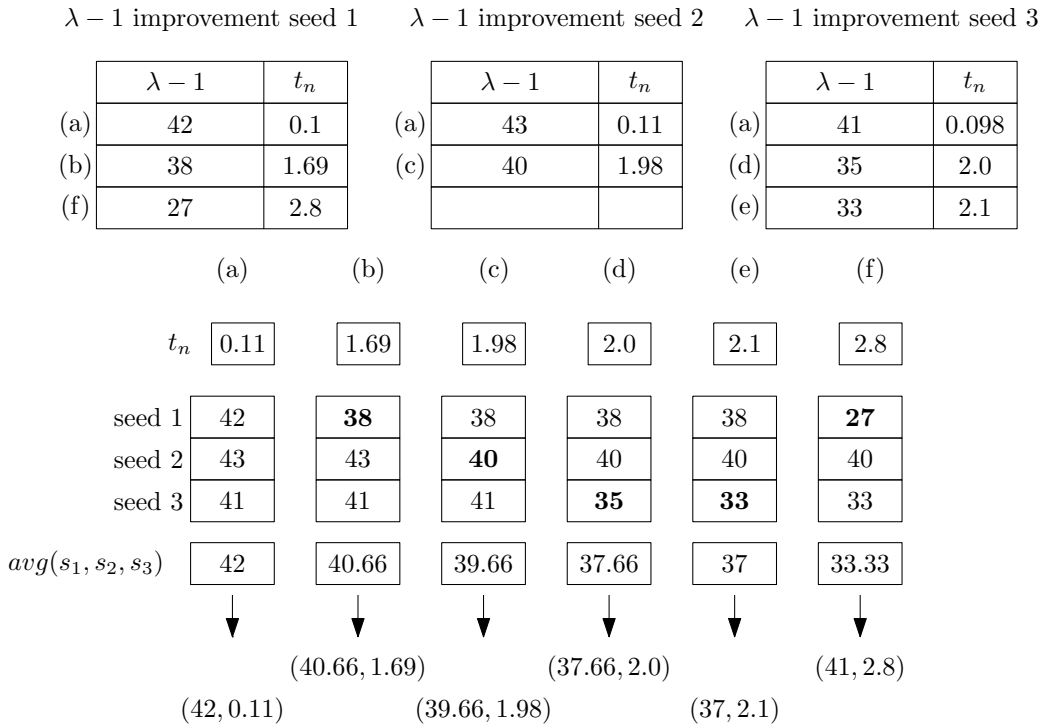


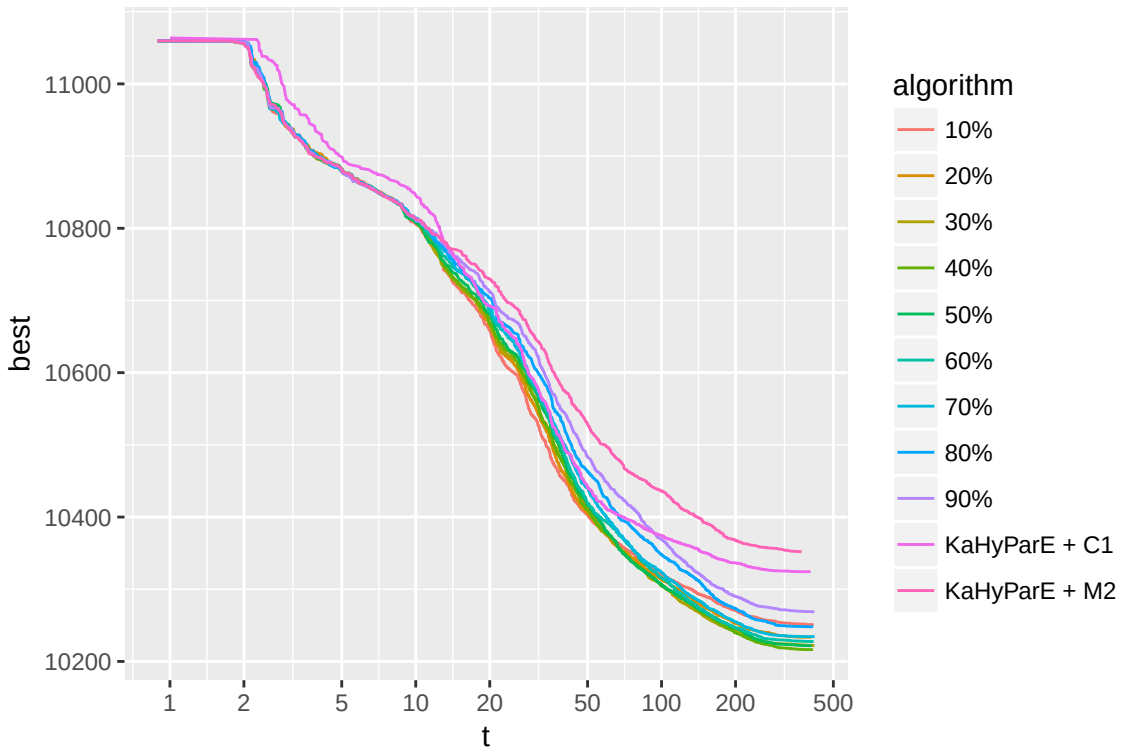
Figure 4.1.: An example for averaging the seeds of an Instance I

The solution improvements of the different seeds are run through by ascending 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.

4.7.2. Tuning Parameters

The parameter spectrum of the evolutionary algorithm is rather voluminous. Beginning with the different chances of the combine and mutation operations to be chosen. As such we first evaluate the operations themselves before tuning the respective chances. The first chance is the amount of mutations performed in contrast to combinations. As such the best value for mutation chance using new initial partitioning, the most powerful mutation operation results in a value of 40% to 50% for mutation chance. This is drastically diverging from the chance used in evolutionary graph partitioning which is around 10% [**?**].

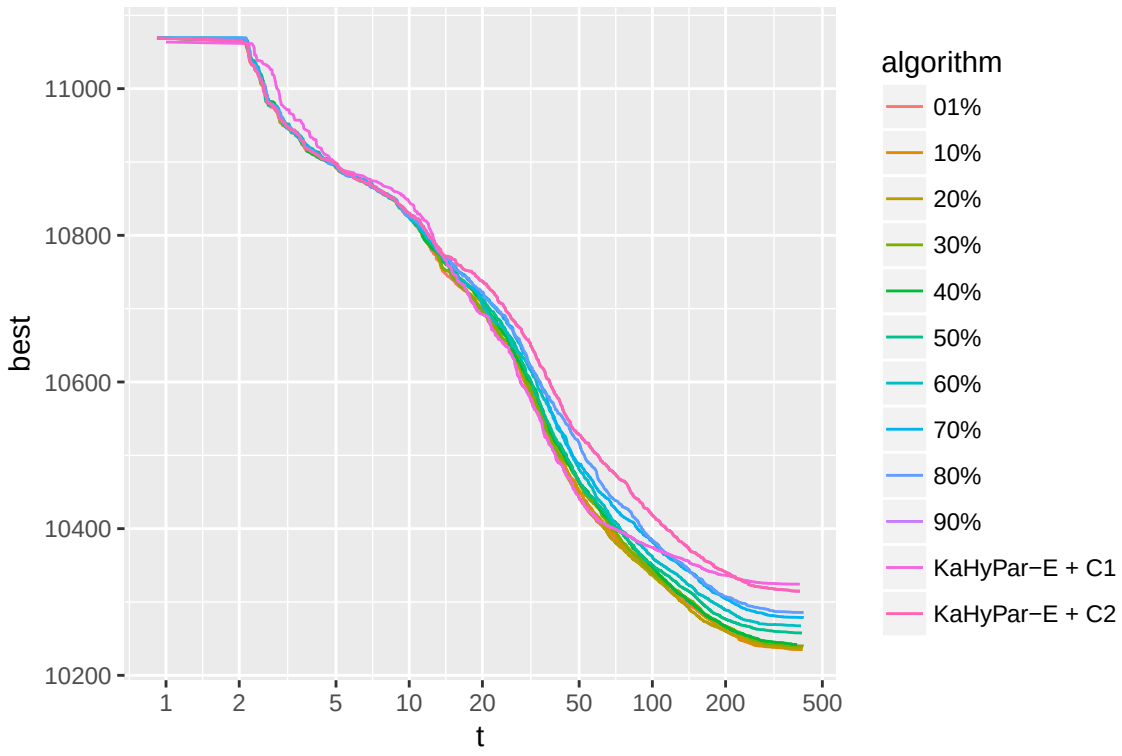
Figure 4.2.: New initial partitioning mutation chance



In this plot the different mutation chances are represented. The percentages represent the chance of performing a new initial partitioning vcycle. 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 40% for new initial partitioning is generating the best solution. This experiment was performed on the tuning subset to generate values for the run on the benchmark subset.

Additionally for tuning the chances of selecting a mutation the chance of performing an edge frequency combine can be tuned as well.

Figure 4.3.: Edge frequency chance



In this plot the chances of performing an edge frequency instead of a basic combine 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 to the mutation chance tuning no significant difference can be observed between the tuned values.

Additional parameters for that were used as default configuration. The dampening factor for edge frequency (C1) is set to $\gamma = 0.5$ as in [**?**]. The time allocation for creating the population is set to 15% of the total time. The time limit was set to 8 hours on the benchmark subset and 2 hours on the tuning subset.

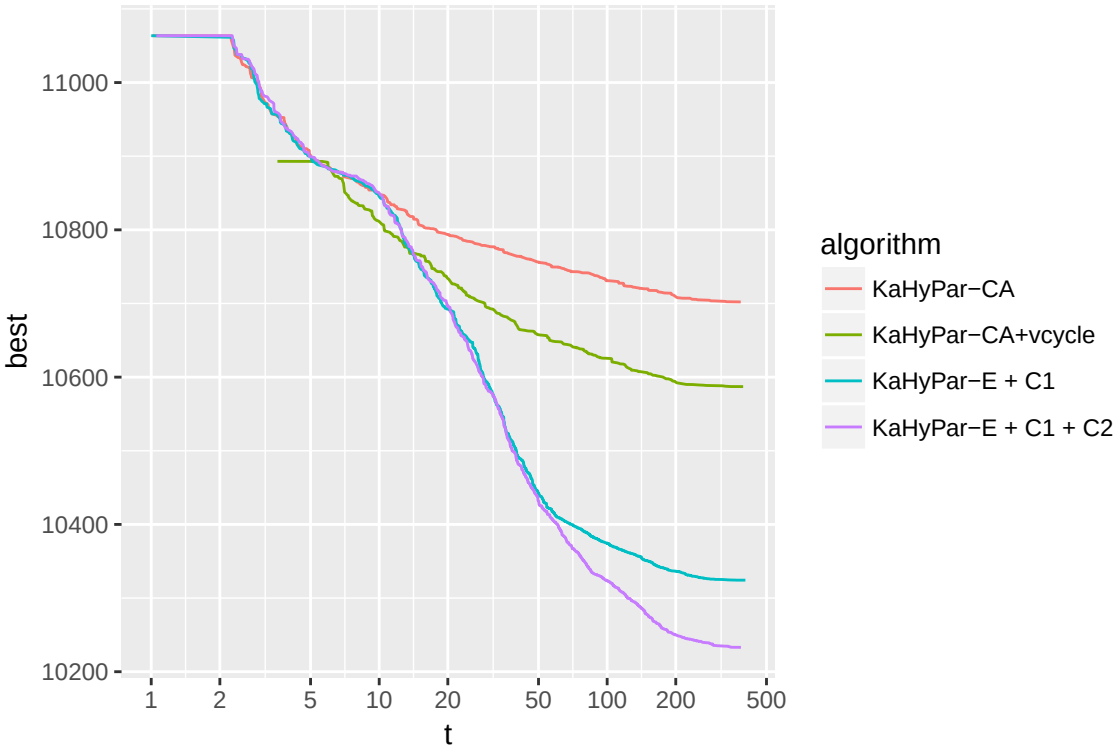
4.7.3. Instances

4.8. Algorithm Evaluation

The first evaluation is to compare the simple version of KaHyPar-E against the best configuration KaHyPar-CA + vcycle. To allow this comparison both algorithms have been

alloted the same time to create a solution. KaHyPar-CA + vcycle is performing repeated repetitions. All algorithm configurations are run 3 times with different starting seeds. This has been performed on the tuning subset.

Figure 4.4.: Comparing KaHyParE to KaHyPar

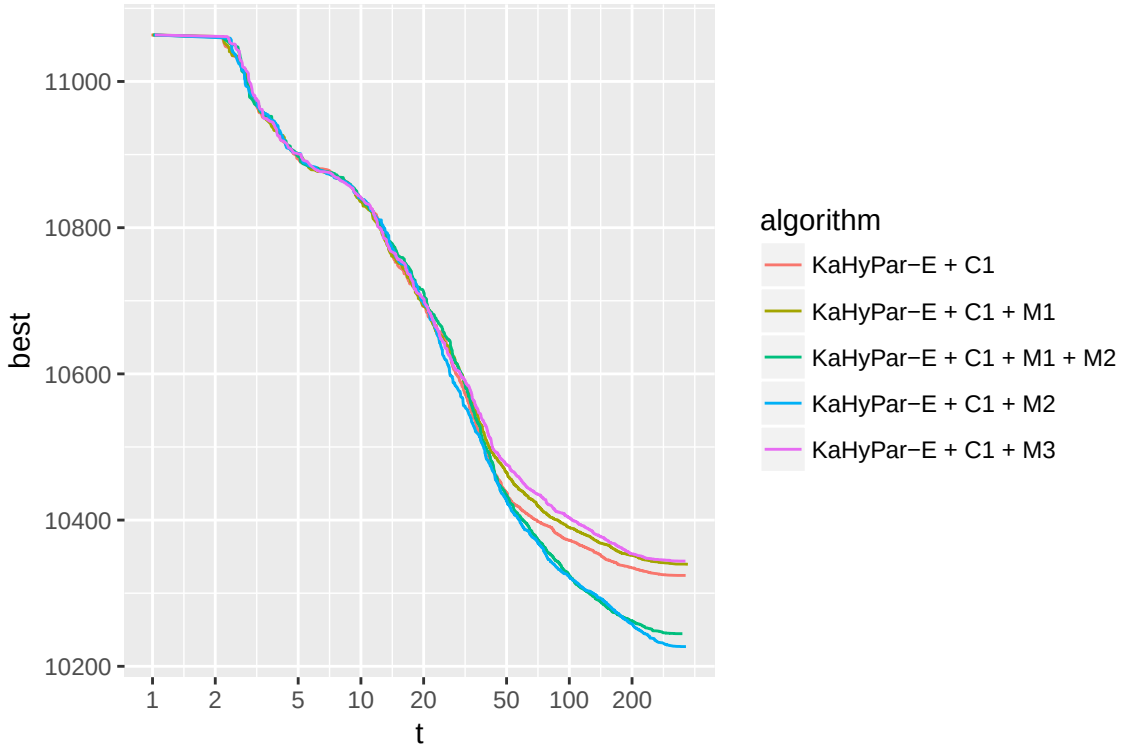


In this plot we evaluate the results of KaHyParE using only basic combines as evolutionary operation against repeated repetitions of KaHyPar-CA and KaHyPar-CA + vcycles which is the strongest configuration for KaHyPar-CA. This evaluation is performed on the tuning subset. The plot lines of KaHyPar-CA, KaHyParE + C1 and KaHyParE + C1 + C2 are nearly identical up to the time point of 10 t_n normalized time. This is due to the fact that KaHyParE is using KaHyPar-CA to generate the initial population. The variations are caused by hardware limitations causing minor fluctuation in the time for performing an iteration and thus slight variations in the normalized time. However the values generated are the same since KaHyPar-CA is configured with the same seed during each of those experiments. KaHyPar-CA + vycles is not sharing the same starting curve. Since v-cycles are time consuming the algorithm steps of KaHyPar-CA + vcycles are slower than the steps of KaHyPar-CA, resulting in an offset of the starting point for the plot line. As expected

KaHyPar-CA + vcycles produces better solutions than KaHyPar-CA, which also extends to repeated repetitions. Both variations of KaHyParE gain a significant amount of improvement after generating the initial population. This is due to the combine schemes being able to exploit structural improvements by comparing different partitions as described in 3.5.1 and continually improving the solution quality by the assurance of nondecreasing solution quality 3.5.1. However this operation is eventually converging into a local optima since all individuals are going to be replaced with the best solution in relatively close proximity due to the replacement strategy and neither replacement strategy nor operator are capable to decrease the solution quality of a given individual. the multicombine operation C2 however is not restricted to this assurance. This operator is profiting from a stable population in a sense that the best solutions in the population are most likely good solutions. This is visible in the plot since KaHyParE + C1 + C2 is not drastically different from KaHyParE + C1 while the basic combine is still effective but allows for an improvement of solution quality whereas KaHyParE + C1 is plateauing. Comparing KaHyPar-E + C1 against KaHyPar-CA + vcycles using the Wilcoxon-Pratt Test generates a Z-Value of 4.37 indicating that KaHyPar-E + C1 is computing better solutions with a confidence of 99.9998%.

After evaluating the combine operators we now consider which mutation operators are improving the solution quality.

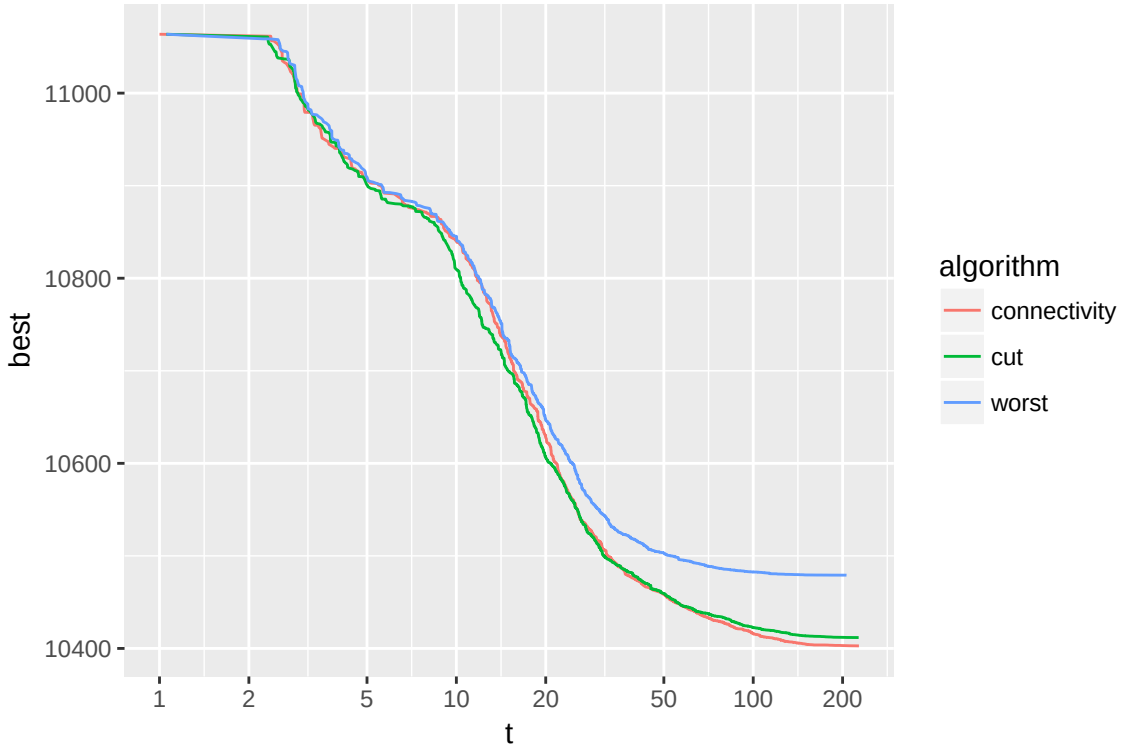
Figure 4.5.: Different Mutation Operations



In this figure the effectiveness of the different mutation operators are evaluated. Each data set was created using a 50% chance of C1 and a 50% chance of the respective mutation operations. In the specific case M1 + M2 the mutation operation performed was selected uniformly at random. This evaluation was performed on the tuning subset. Adding simple v-cycles 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 unable to create worse solutions which are necessary to broaden the solution scope of the population. 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 scope can be explored more effectively. Stable net detection M3 is generating worse solutions or using up more time to generate individuals and therefore dropped from the algorithm. 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.

Mutation operations 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.

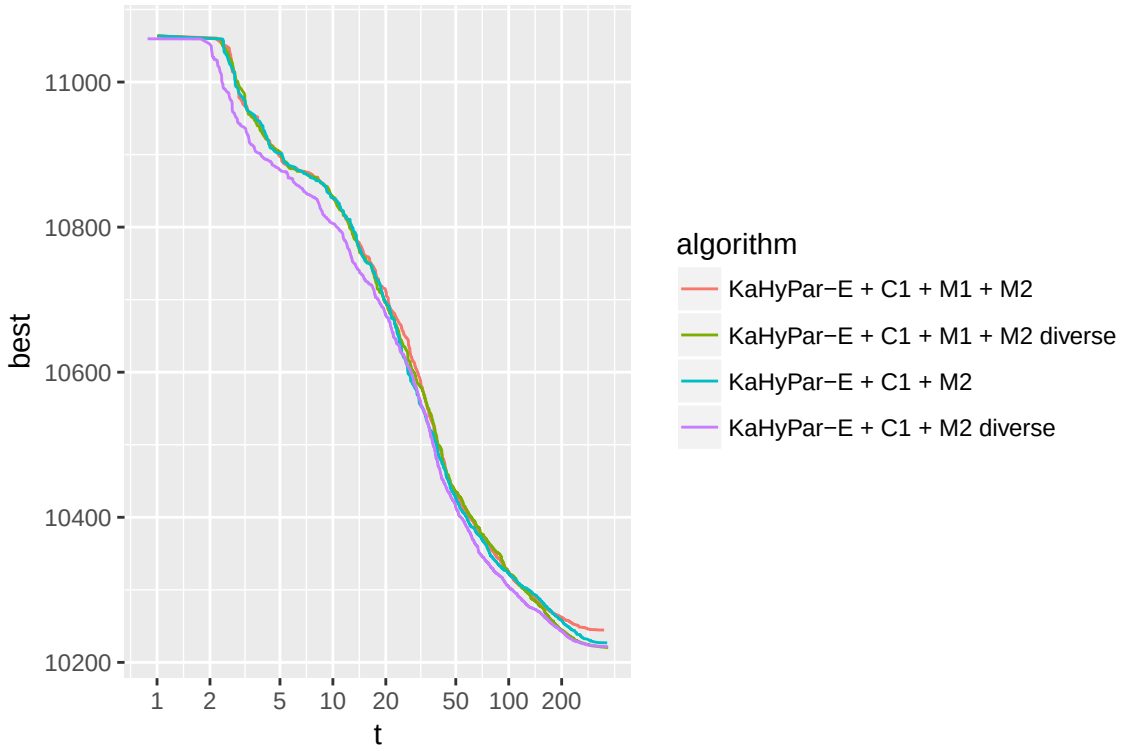
Figure 4.7.: Using different replacement strategies



In 4.7 we compare the effectiveness of the three different replacement strategies. Replacing the worst element in the population leads to premature convergence, which is the primary reason for introducing the diverse replacement strategy. However the most natural approach for hypergraphs using the connectivity difference instead of the cut difference results in a slight improvement of the quality. This leads to the conclusion that the connectivity approach is generating sufficiently different results from the cut approach to influence the replacement target.

Translating the results onto the big data set.

Figure 4.6.: Different replacement strategies for mutations



As seen in this plot the different replacement strategies are displayed for M2 and a combination of M1 and M2. While the difference in KaHyPar-E + C1 + M2 and KaHyPar-E + C1 + M2 diverse is only a miniscule improvement in convergence time and solution quality, the difference of KaHyPar-E + C1 + M1 + M2 and KaHyPar-E + C1 + M1 + M2 + diverse is more prominent in terms of solution quality.

Table 4.1.: Connectivity improvement of the strongest configurations for KaHyPar-E

	KaHyPar-E+C1+C2		KaHyParE+C1+M1+M2	
k	KaHyPar-CA-V	KaHyPar-CA	KaHyPar-CA-V	KaHyPar-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	3.4%	5.1%	3.3%	5.0%

Figure 4.8.: Benchmark subset results

The results for the big dataset should be described HERE

Figure 4.9.: Benchmark subset split by k

The results for the specific k values should be described HERE

5. Discussion

5.1. Conclusion

Creating an evolutionary framework for KaHyPar resulted in a quality improvement for hypergraph partitions of 3.2%. 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 applicable to hypergraphs for replacement. 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 authors best 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

Currently KaHyPar-E is providing solution quality but lacks performance. Adding a layer of parallelization would allow a significant speedup by creating multiple individuals during an iteration. This should be realizable without greater effort due to the independence of the child and parent partitions. Another interesting approach is a time cost analysis for the different operators. There is a strong suggestion that the basic combine operator is significantly faster than a regular iteration in KaHyPar. If this suggestion 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 sophistiacted 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.

A. Implementation Details

A.1. Software

The source code was written in C++, using the C++11 version. The software was compiled by gcc-5.2+. The combine operators are implemented as a policy in the rater. These policies are meant to avoid overhead and control flow complexity during the coarsening, which is the most time consuming part of an iteration in KaHyPar. An action element is attached to each iteration, containing the necessary requirements for each of the different operation. These requirements activate or deactivate initial partitioning or the use of parent information during the coarsening, depending on the action performed. Edge frequency and stable net removal use the same baseline vector containing the amount of cut edges from X partitions.

A.2. Hardware

The experiments on the tuning subset were performed on i128. The experiments on the benchmark subset were performed on bwUnicluster.

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, 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.

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

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

[5] 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.

[6] 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.

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

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

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

[10] BOESE, KENNETH D, JASON CONG, ANDREW B KAHNG, KWOK-SHING LEUNG DIAN ZHOU: *On high-speed VLSI interconnects: Analysis and design*. *Proc. Asia-Pacific Conf. on Circuits and Systems*, 35–40, 1992.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.

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

[30] 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.

[31] 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.

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

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

[34] 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.

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

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

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