Karlsruher Institut für Technologie

Bachelor thesis

Evolutionary Hypergraph Partitioning

Robin Andre

Date: 10. November 2017

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

Institute of Theoretical Informatics, Algorithmics
Department of Informatics

Abstract

The NP-hard hypergraph partitioning problem has many real life applications like processor load balance or VLSI design. As such meta heuristics are the design philosophy to increase solution quality. In this thesis we create an effective evolutionary algorithm improving the existing multilevel hypergraph partitioner KaHyPar. We introduce operations heavily integrated in both the evolutionary and multilevel aspect. We evaluate the benefits of the algorithm on 90 instances, showing that the evolutionary component increases 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. Wir evaluieren die Vorteile des evolutionären Algorithmus auf 90 Hypergraphinstanzen, wobei gezeigt werden kann dass eine durchschnittliche Verbesserung von durchschnittlich 2.2% erzielt werden kann.

Acknowledgments

I'd like to thank Timo Bingmann 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[12]. 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[22]. 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 [14]. 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 augments 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.

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 the vertices 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 $c(V') = \sum_{v \in V'} c(v)$ and $w(E') = \sum_{e \in E'} w(e)$. Two verticies u, v are adjacent if $\exists\ e \in E \mid u, v \in e$. 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 pins 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 $\mathcal{P}$ 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 $cut(e, \mathcal{P}) := \Phi(e, \mathcal{P}) > 1$. Let $\mathcal{E}$ be the set of cut edges from a partitionP The cut metric ofPis 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 [15], circuit partitioning [3] or database storage sharding [26].

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 hMetis[27] and MLPart[3]. By recursively bisectioning the resulting partitions, k can assume values other than 2. This is implemented in tools like PatoH [15], Mondrian[39] and Zoltan[20]. 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[28], kPatoH [7] and SHP[26](also

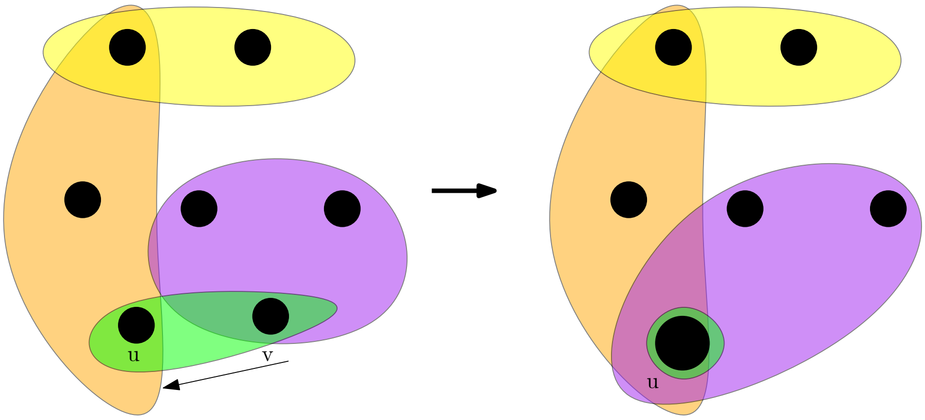


Figure 2.1.: An example of a contraction. Note that the Hyperedges of the reduced node v are rearranged to contain u.

implementing a recursive bisection). Note that except SHP [26] 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 surveys [4][9][32][38].

Saab and Rao [33] 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 [25] 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 [13], in which solutions are preprocessed and optimized using the Fiduccia-Mattheyses [21] 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[16] 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 designed for k-way optimization [35] local search as well as a 4-point crossover operation, which splits the input partitions at 4 points and alternates between the blocks. [29] Kim et al. translate the lock gain local search [30] 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 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.[6] 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 [36] for the existing graph partitioner KaFFPa, [24] 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 [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 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 [21].

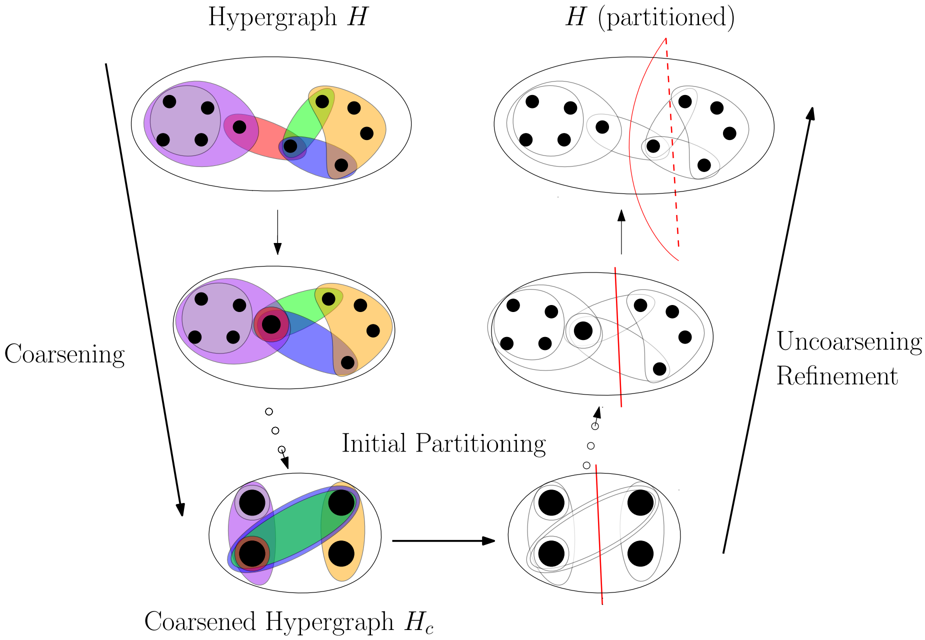


Figure 2.2.: An example of an iteration. Coarsening, Initial Partitioning and Refinement

In 2017 Heuer and Schlag improved KaHyPar by analyzing and exploiting community structures in Hypergraphs [23], 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 [17]. 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[8], 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 [36] determines the difference of two partitions P_1, P_2 by counting the edges that are cut edges in exclusively one of the partitions $cutdiff(\mathrm{P}_1,\mathrm{P}_2) := \sum_{e \in E} |cut(e,\mathrm{P}_1) - cut(e,\mathrm{P}_2)|$. 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.

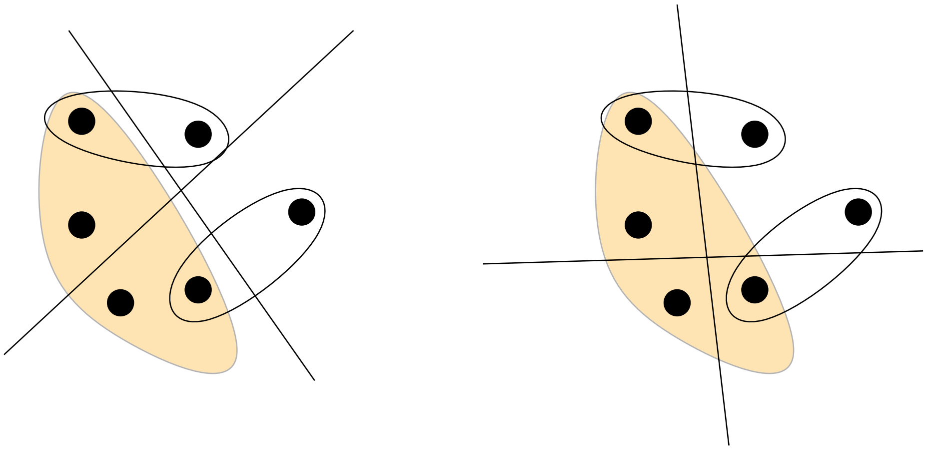


Figure 3.1.: Two different partitions of the same hypergraph

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 [11], 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 [41] 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 [19]. 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

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

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 [19] 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. [31] 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.

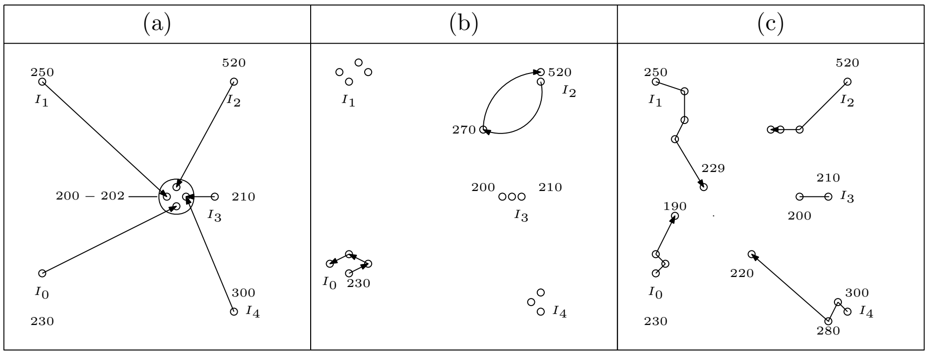


Figure 3.2.: Visual representation of different replacement strategies (a) replacing the worst element, (b) replacing the most similar element & (c) replacing similar/worse

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 [23], which consists of various hypergraphs from the ISPD98 benchmark [2], Sparse Matrices [18], the DAC benchmark [40] and SAT 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 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 [23], 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

Table 4.1.: Hypergraph properties of 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 4.1 displays the instances in the tuning subset, as well as their basic properties. n is the number of vertices, m ist 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 4.2.: Hypergraph properties of 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

The main interest is to show how the best solution of the population is increasing compared to the time used by the evolutionary algorithm. There are three inherent issues that need to be dealt with.

(i) The iteration times vary greatly between hypergraphs.

(ii) The different seeds need to be normalized.

(iii) The connectivity also varies greatly between hypergraphs.

To emphasize the first issue, lets assume two hypergraph instances s and l. For the small hypergraph s the evolutionary algorithm may perform thousands of iterations, and for the large hypergraph l only 10. When using the absolute time as comparison we have no guarantee how many iterations the evolutionary algorithm has performed in average at a given time point. Similarly when using the iterations of the evolutionary algorithm as comparsion we cannot display the time required by the iterations. The Operator C_x may produce slightly better results than operator C_y per iteration, but also require 1000 times the time per iteration. Sanders and Schulz [36] present an approach using normalized time as a comparison tool. We choose one of the partitioning algorithms as baseline and determine the average duration t_I to partition a given instance I one time. We then can calculate for each absolute timestamp t of an instance I the normalized time by $t_n = \frac{t}{t_I}$. By doing so we can adjust the variation of iteration times, without dropping the duration information.

To address the second issue, we used multiple seeds to balance out outlier performance due to randomization. We determine the average best 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 best solution. Each time a seed s is improving its best solution at a timepoint 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 elements of the following structure $(avg(a_s), t_n)$. By ordering the list of averaged improvements by t_n we can calculate the average best solution at a timepoint t_n.

$\lambda - 1$ improvement seed 1

	$\lambda - 1$	t_n
(a)	42	0.1
(b)	38	1.69
(f)	27	2.8

$\lambda - 1$ improvement seed 2

	$\lambda - 1$	t_n
(a)	43	0.11
(c)	40	1.98

$\lambda - 1$ improvement seed 3

	$\lambda - 1$	t_n
(a)	41	0.098
(d)	35	2.0
(e)	33	2.1

	(a)	(b)	(c)	(d)	(e)	(f)
t_n	0.11	1.69	1.98	2.0	2.1	2.8
seed 1	42	**38**	38	38	38	**27**
seed 2	43	43	**40**	40	40	40
seed 3	41	41	41	**35**	**33**	33
$avg(s_1, s_2, s_3)$	42	40.66	39.66	37.66	37	33.33
	$(42, 0.11)$	$(40.66, 1.69)$	$(39.66, 1.98)$	$(37.66, 2.0)$	$(37, 2.1)$	$(41, 2.8)$

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.

Next we need to average the different instances. Because different instances have higly varying connectivity, we cannot use the arithmetic mean as averaging tool. Instead we use the geometric Mean, to counteract a priorization of instances because of their size. 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

list of averaged improvements

instance 1

	$\lambda - 1$	t_n
(a)	42	0.11
(c)	40.66	1.69
(d)	39.66	1.98

instance 2

	$\lambda - 1$	t_n
(a)	28000	0.12
(b)	27500	1.5

instance 3

	$\lambda - 1$	t_n
(a)	9	0.1
(e)	8	2.0
(f)	4	3

	(a)	(b)	(c)	(d)	(e)	(f)
t_n	0.12	1.5	1.69	1.98	2.0	3
instance 1	42	42	**40.66**	**39.66**	39.66	39.66
instance 2	28000	**27500**	27500	27500	27500	27500
instance 3	9	9	9	9	**8**	**4**
$avg(s_1, s_2, s_3)$	219.5	218.2	215.9	214.1	205.9	163.4

Result list: $(219.5, 0.12)$ $(218.2, 1.5)$ $(215.9, 1.69)$ $(214.1, 1.98)$ $(205.9, 2.0)$ $(163.4, 3)$

Figure 4.2.: An example for averaging the seeds of an Instance I

Similar to seed averaging the lists of averaged improvement 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.

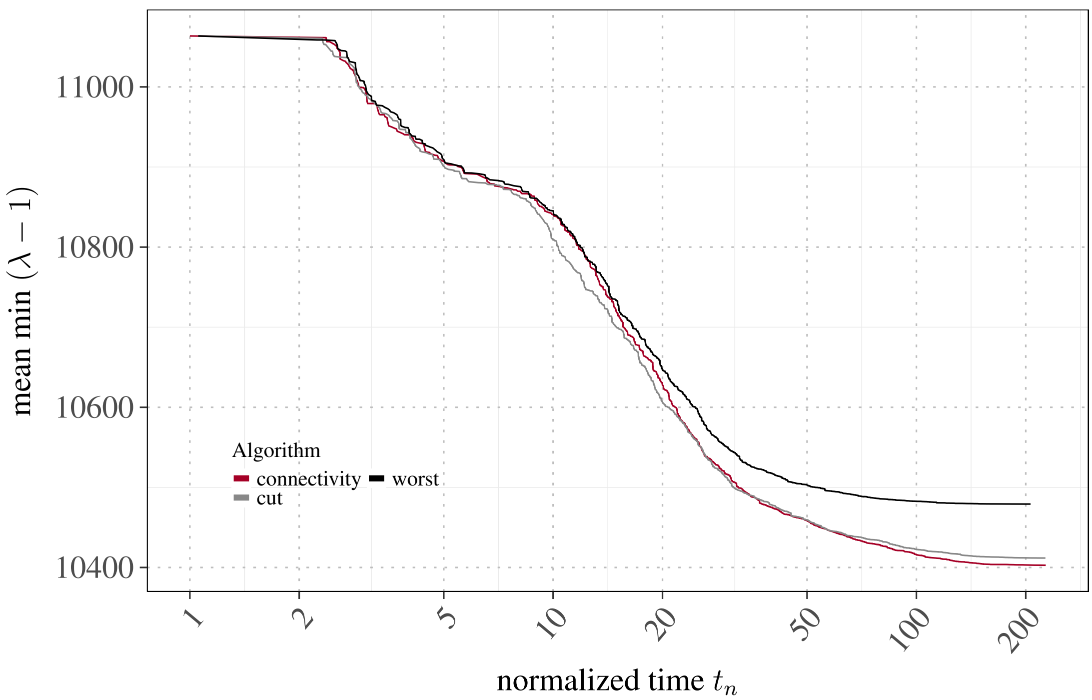
4.3. First Evaluation

All of the data sets presented in this section were created using the tuning subset. These data sets were computed on an Ubuntu 14.04 machine with four Intel Xeon E5-4640 Octa-Core processors @2.4 GHz with 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 for the population size, $\gamma = 0.5$ as dampening factor for C2, C2 and M3 are using $\sqrt{|P|}$ as default amount of individuals. The threshold of M3 is 0.75. As preluded the time limit t is set to 2 hours, $\epsilon = 0.03$ and $k = 32$. The used seeds are $1, 2$ and 3. The parameters of KaHyPar are set using the default configuration for direct k-way partitioning. [1]

4.3.1. Replacement strategy evaluation

We begin by evaluating the different replacement strategies presented in section **??**. All data sets use KaHyPar-E with the operator C1 as the only evolutionary action. The population size is determined dynamically using $\delta = 15\%$.

Figure 4.3.: Using different replacement strategies



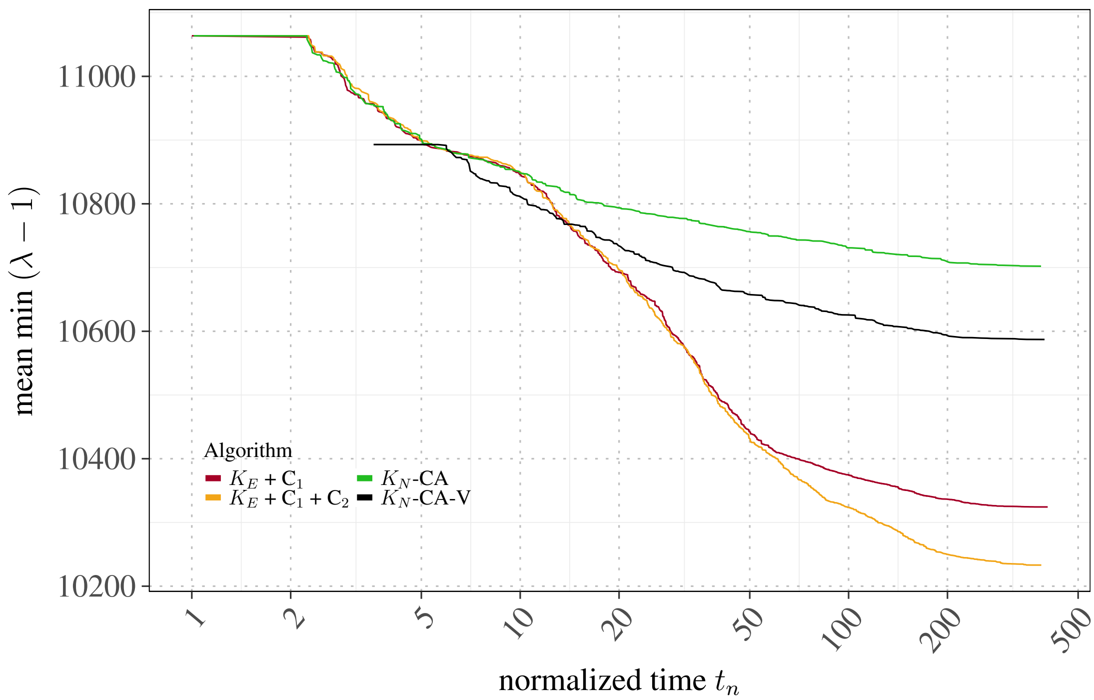
[1] https://github.com/SebastianSchlag/kahypar/blob/master/config/km1_direct_kway_sea17.ini

In figure4.3 we compare the effectiveness of the three different replacement strategies. As expected in section **??** replacing the worst element in the population leads to premature 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 is able to slightly increment the solution quality. This comfirms the assumption formulated in section 3.3 that connectivity difference is a more appropriate approach in expressing different characteristics of two partitions. Based on these results we will use connectivity difference as default replacement strategy in all upcoming evaluations.

4.3.2. Combine operator evaluation

Next up we evaluate the different combine operators and compare the solution quality against the solution quality of KaHyPar-CA and KaHyPar-CA+V. We evaluate three configurations of the evolutionary algorithm. The specification KaHyPar-E+C_x means that the configuration used the combine operator C_x. KaHyPar-E+C1+C2 chooses one of the two combine strategies randomly for each iteration.

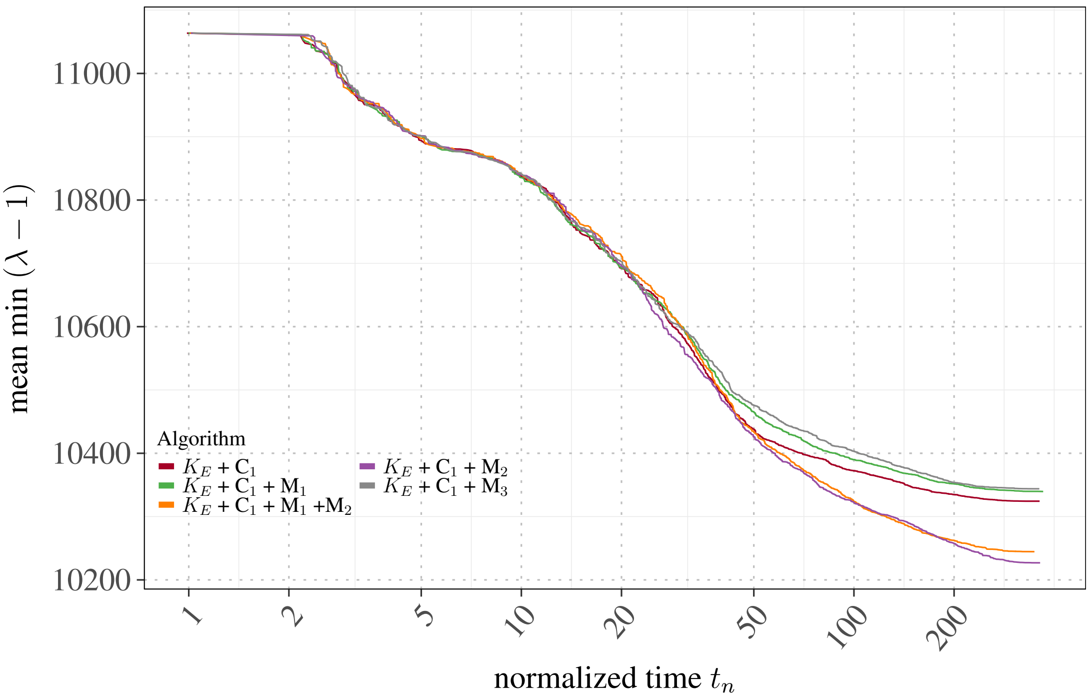
Figure 4.4.: Comparing KaHyPar-E to KaHyPar



In figure 4.4 we evaluate the results of KaHyPar-E using only basic combines as evolutionary operation against repeated repetitions of KaHyPar-CA and KaHyPar-CA+v which is the strongest configuration for KaHyPar-CA. This evaluation is performed on the tuning subset. The plot lines of KaHyPar-CA, KaHyPar-E + C1 and KaHyPar-E + C1 + C2 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 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+v is not sharing the same starting curve. Since v-cycles are time consuming the algorithm steps of KaHyPar-CA+v are slower than the steps of KaHyPar-CA, resulting in an offset of the starting point for the plot line. As expected KaHyPar-CA+v produces better solutions than KaHyPar-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 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 can most likely be considered good solutions. This is visible in the plot since KaHyPar-E + C1 + C2 is not drastically different from KaHyPar-E + C1 while the basic combine is still effective but allows for an improvement of solution quality whereas KaHyPar-E + C1 is plateauing. Comparing KaHyPar-E + C1 against KaHyPar-CA + v 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%.

4.3.3. Mutation operator evaluation

Figure 4.5.: Different Mutation Operations

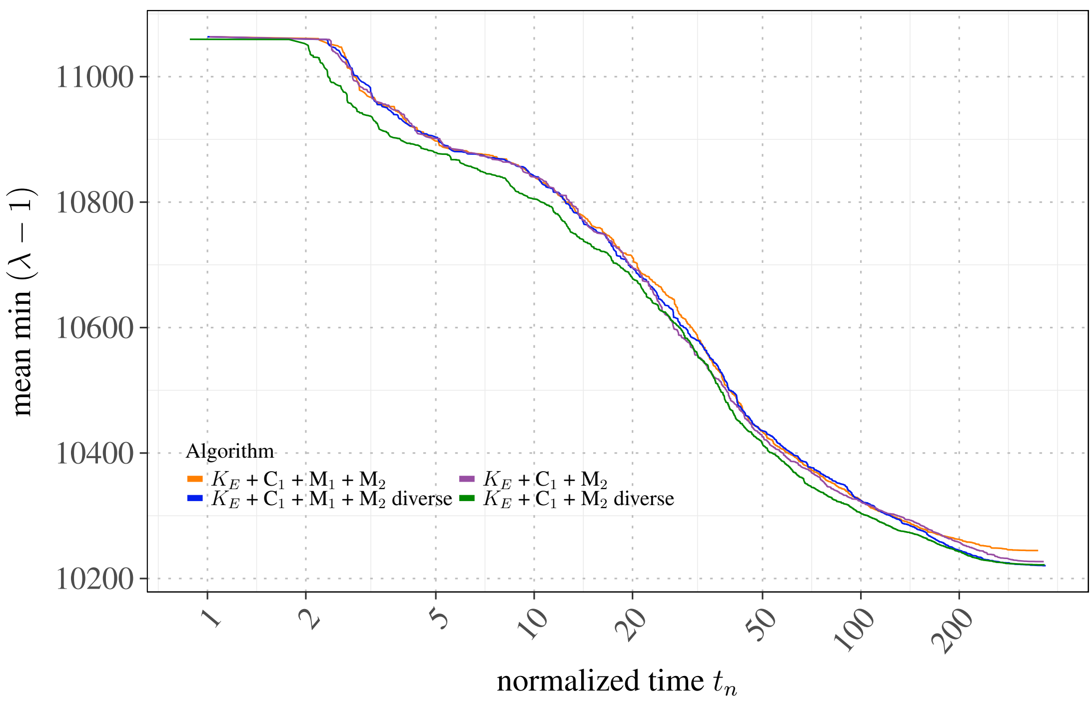


In this figure the effectiveness of the different mutation operators are evaluated. Each data set was created using a 50% chance of C1 and a 50% chance of the respective mutation operations. In the specific case M1 + M2 the mutation operation performed was selected uniformly at random. This evaluation 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.

4.3.4. Mutation replacement strategies

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 4.6.: 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.

4.4. Tuning Parameters

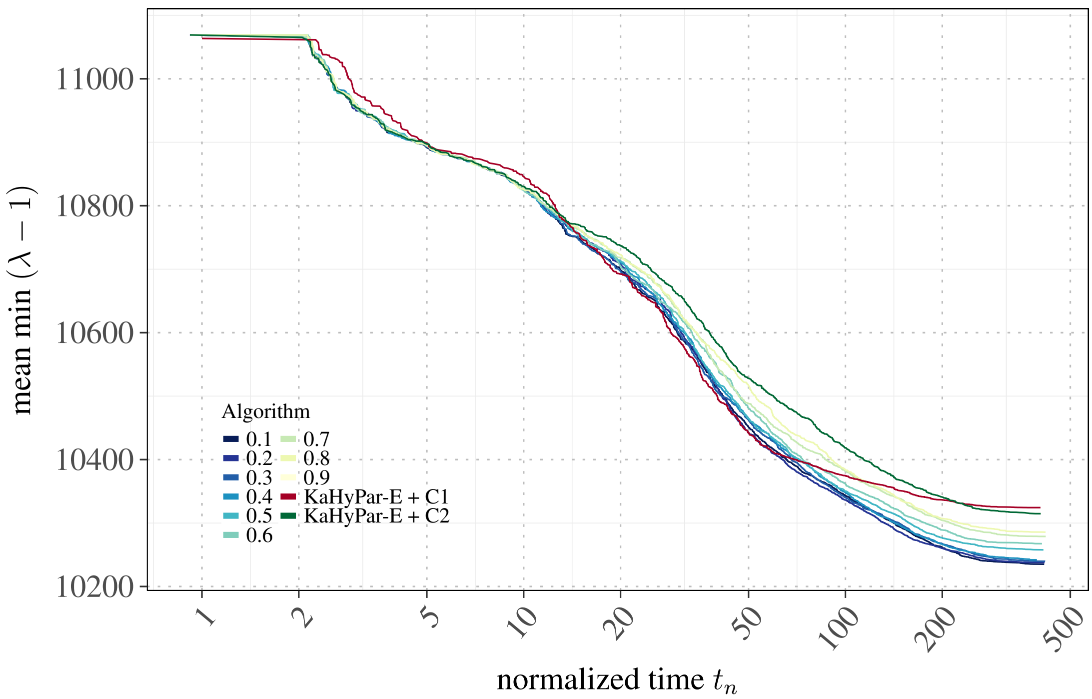
KaHyPar-E is adding the following parameters to KaHyPar: dynamic population size $\delta = 0.15$, upper and lower bound limits for the population size $[3, 50]$, the edge frequency dampening parameter $\gamma = 0.5$ as well as the amount of individuals used by C2 & M3 $n = \sqrt{|P|}$. Additional parameters are the replacement strategy, as well as the distribution of combine operators and mutation operators. Most importantly, KaHyPar-E adds the time limit t. Noteworthy parameters of KaHyPar include $\epsilon = 0.03$, k and the seed s. For other parameters of KaHyPar we refer to the default configurations [2].

4.4.1. Combine chance distribution

As seen in section 4.3.2 a combination of both C1 + C2 is generating better results when being used together. We want to analze different percentages regarding the distribution of the two operatos

[2]https://github.com/SebastianSchlag/kahypar/blob/master/config

Figure 4.7.: Edge frequency chance



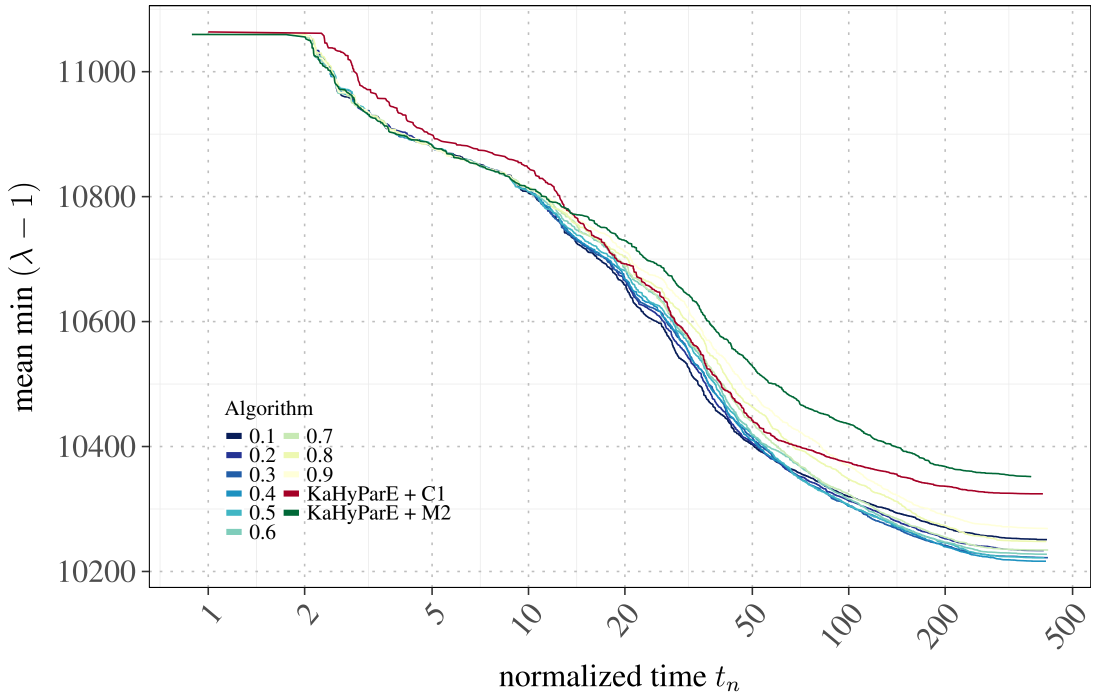
In this plot the chances of performing an edge frequency instead of a basic combine are displayed. Clearly recognizable is that a proper application of both operators will lead to improvements. The optimal values are in a range from 20% to 50%, however similar to the mutation chance tuning no significant 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 [41]. 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.

4.4.2. Mutation chance distribution

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% [36].

Figure 4.8.: New initial partitioning mutation chance

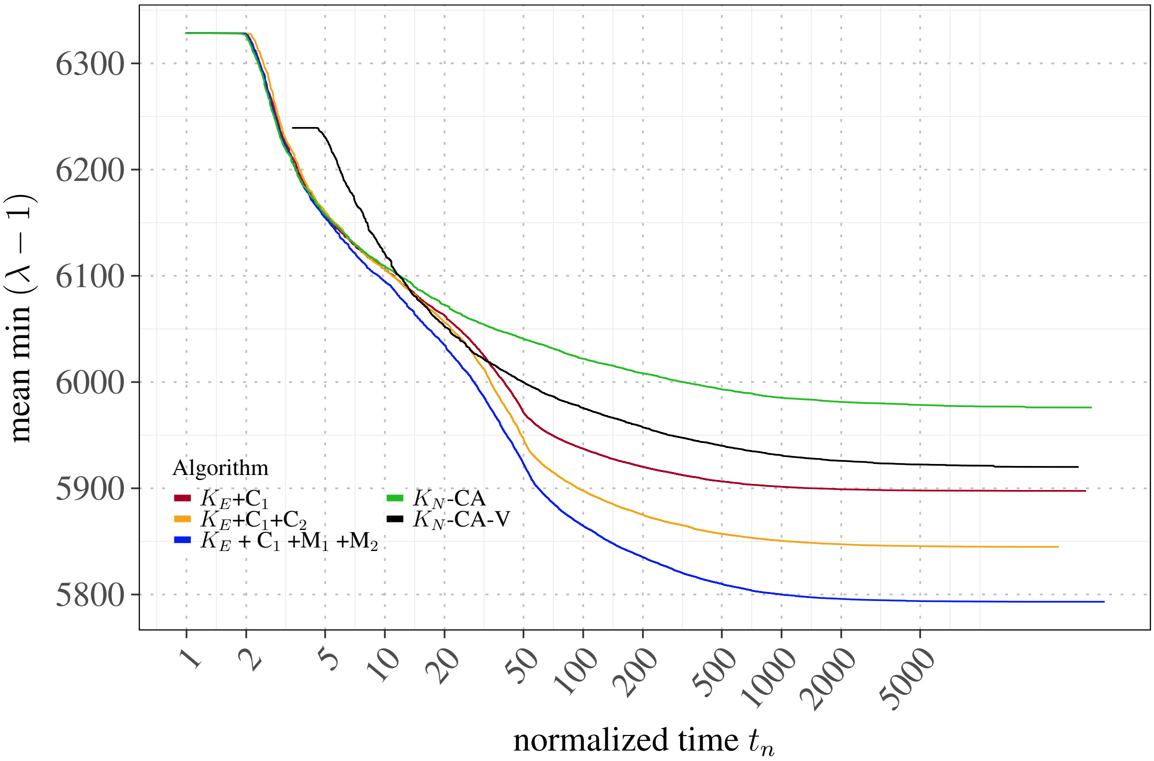


In this plot the different mutation chances are represented. The percentages represent the chance of performing a new initial partitioning vcycle. If not performing a new initial partitioning a basic combine is performed. It is visible that choosing either 0% mutation chance or 100% mutation chance are 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 30% -50% for new initial partitioning is generating the best solutions. This experiment was performed on the tuning subset to generate values for the run on the benchmark subset.

4.5. Final Evaluation

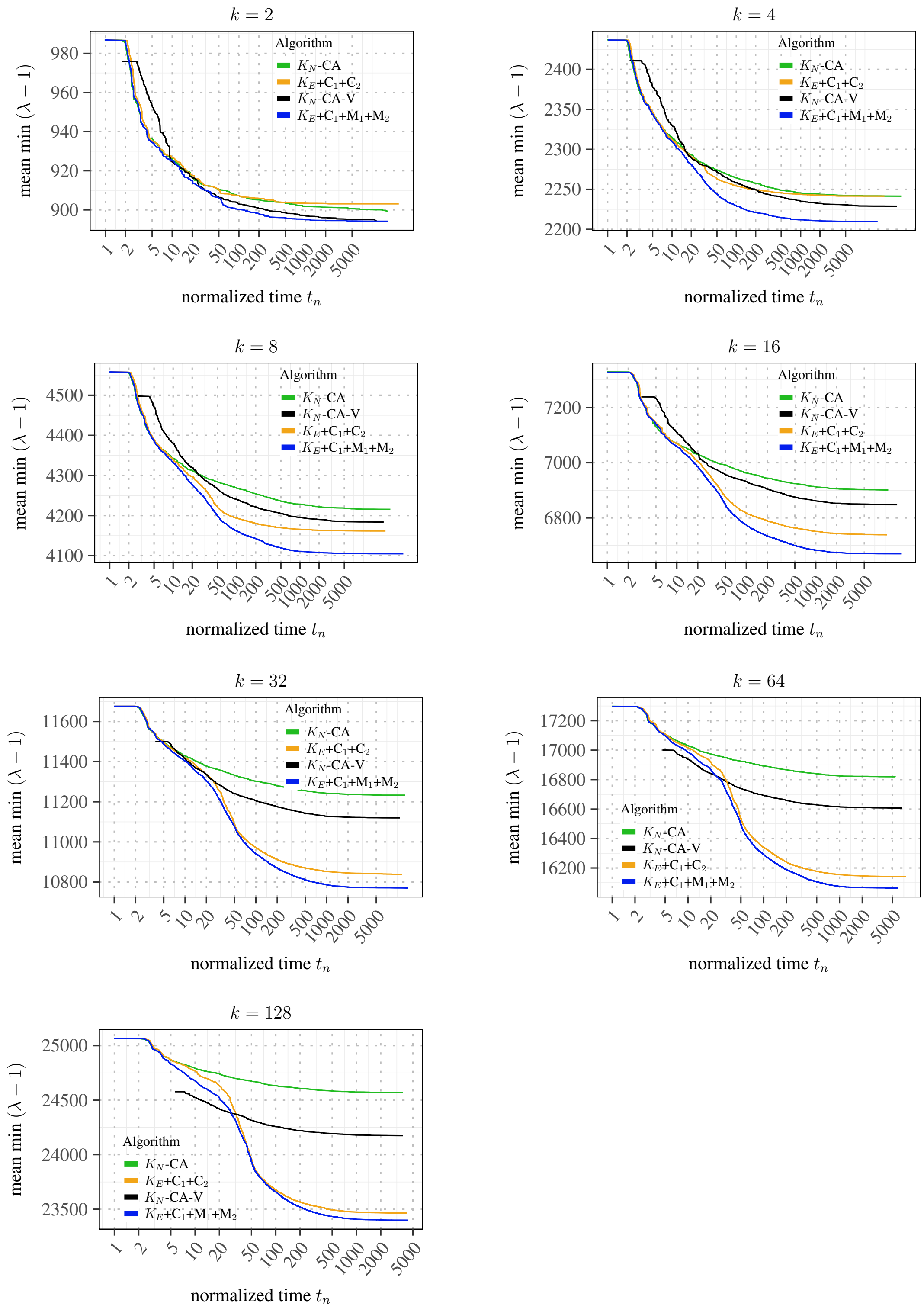
Now we use the results form the tuning subset and translate them onto the benchmark subset. As already mentioned the data sets are generated over more hypergraph instances, using t 8 hours for partitioning, $\epsilon = 0.03$, and $=\{2, 4, 8, 16, 32, 64, 128\}$. The following data sets are evaluated on the benchmark subset: KaHyPar-CA, KaHyPar-CA-V, KaHyPar-E+C1, KaHyPar-E+C1+C2 and KaHyPar-E+M1+M2. The data sets were created 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 and local disks with 2 TB capacity. The nodes were allocated exclusively to avoid cpu time interference.

Figure 4.9.: Benchmark subset results



In average all evolutionary operators are performing better. TODO TEXT

Figure 4.10.: Benchmark subset split by k

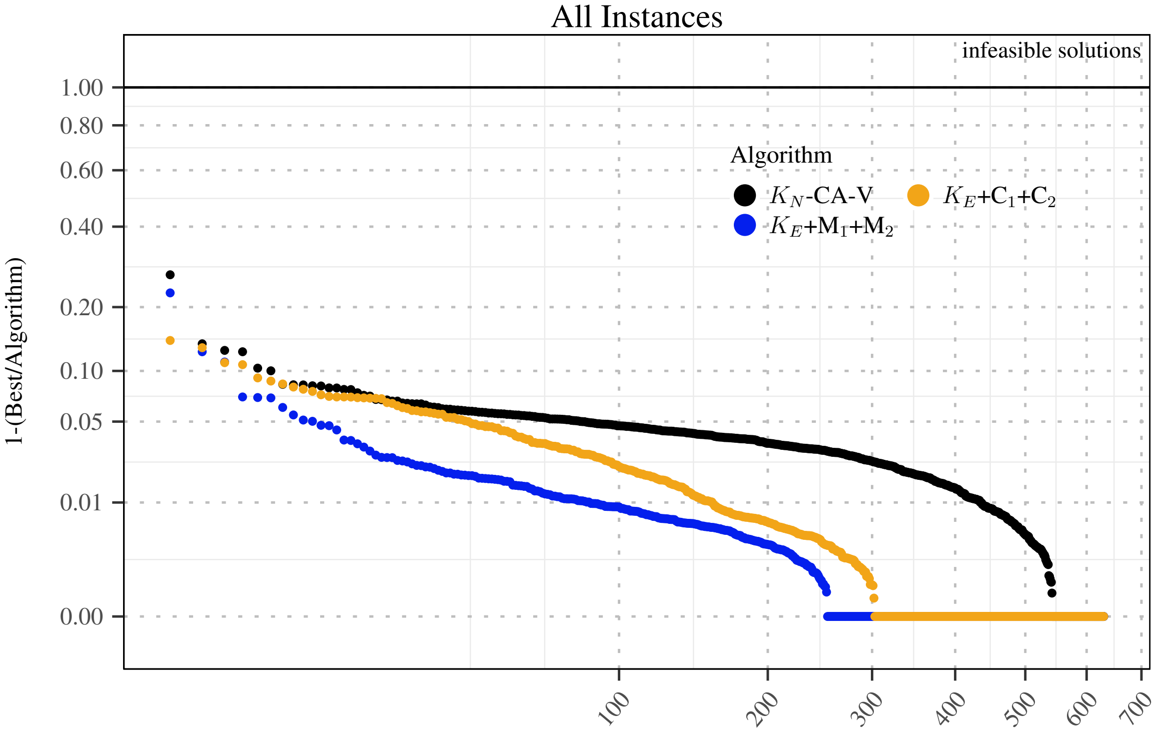


The results for the specific k values should be described HERE

Table 4.3.: 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%

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 improvements are then split by k.



5. Discussion

5.1. Conclusion

This thesis presents an evolutionary framework for KaHyPar resulting in a quality improvement for hypergraph partitions 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 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. 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 no speedup for a single partition. 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.

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] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

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

[31] 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.

[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.