Multilevel Memetic Hypergraph Partitioning with Greedy Recombination

Utku Umur Acikalin
Bugra Caskurlu
u.acikalin@etu.edu.tr
b.caskurlu@etu.edu.tr
TOBB University of Economics and Technology
Ankara, TURKEY

ABSTRACT
The Hypergraph Partitioning (HGP) problem is a well-studied problem that finds applications in a variety of domains. In several application domains, such as the VLSI design and database migration planning, the quality of the solution is more of a concern than the running time of the algorithm. In this work, we introduce novel problem-specific recombination and mutation operators and develop a new multilevel memetic algorithm by combining these operators with kKaHyPar-E. The performance of our algorithm is compared with the state-of-the-art HGP algorithms on 150 real-life instances taken from the benchmark sets used in the literature. The experiments reveal that our algorithm outperforms all others, and finds the best solutions in 112, 115, and 125 instances in 2, 4, and 8 hours, respectively.

CCS CONCEPTS
• Mathematics of computing → Hypergraphs; • Mathematics of computation → Evolutionary algorithms.

KEYWORDS
multilevel hypergraph partitioning, memetic algorithms

ACM Reference Format:
Utku Umur Acikalin and Bugra Caskurlu. 2022. Multilevel Memetic Hypergraph Partitioning with Greedy Recombination. In Genetic and Evolutionary Computation Conference Companion (GECCO ’22 Companion), July 9–13, 2022, Boston, MA, USA. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3520304.3529050

1 INTRODUCTION AND PRELIMINARIES
An undirected hypergraph \( H \) is a four-tuple \( (V, N, w, c) \), where \( V \) is the set of nodes, \( N \subseteq 2^V \setminus \emptyset \) is the set of nets, \( w : V \to \mathbb{R}_{\geq 0} \) is the node weight function, and, \( c : N \to \mathbb{R}_{\geq 0} \) is the net weight function. The set of nodes connected by a net \( e \) are called the pins of \( e \) and denoted by \( pins(e) \). We denote the set of nets incident to a node \( v \) with \( N(v) \subseteq N \). A \( k \)-way partition \( \Pi = \{V_1, V_2, \ldots, V_k\} \) of a hypergraph \( H \) is a partition of \( V \) into \( k \) nonempty disjoint blocks. \( \Pi \) is \( \epsilon \)-balanced if the size of each block is no more than \( \epsilon \) times the average block size. The set of blocks that contains a pin of a net \( e \) is called the connectivity set of \( e \), the size of which is denoted by \( \lambda(e) \).

The \( k \)-way hypergraph partitioning (HGP) problem is to find an \( \epsilon \)-balanced \( k \)-way partition \( \Pi \) of a given hypergraph \( H \) that minimizes \( \sum_{e \in N} (\lambda(e) - 1) \cdot c(e) \).

In some real-life applications, such as parallel matrix multiplication [5], the HGP problem is used to accelerate the main computation, thus it needs to be solved fast. In applications, such as VLSI design [11] and database migration planning [1, 17, 18], however; not only that the HGP problem is solved offline, but also the quality of the solution is directly related to the cost of the operation. Ergo even a small improvement in solution quality is critical [20]. In this paper, we target the applications of the HGP problem for which the quality of the solution is the main concern.

The literature on the HGP problem has focused on developing fast heuristics [5, 10, 11, 15] and almost all of the state-of-the-art algorithms use the multilevel (ML) paradigm [4] that consists of the following three stages: coarsening, initial partitioning, and uncoarsening. At the coarsening stage, nodes of the hypergraph are contracted at each level to obtain a series of smaller hypergraphs until the size of the hypergraph reaches a predetermined size. The coarsest hypergraph is partitioned at the initial partitioning stage. At each level of the uncoarsening stage, coarsening is reverted and the partition is refined using local search methods.

kKaHyPar-E, the first ML memetic algorithm for the HGP problem, outperforms all other ML algorithms on a large benchmark set, where each algorithm is given 8 hours [2]. None of the non-multilevel evolutionary algorithms is regarded as competitive as the state-of-the-art multilevel HGP tools [6]. In contrast to other evolutionary algorithms, which use mutation and recombination operators to obtain partitions directly, operators of kKaHyPar-E guides the coarsening stage of an ML algorithm.

Our main contribution is problem-specific mutation and recombination operators for the HGP problem that effectively explores the search space. We incorporate these operators to the KaHyPar framework to obtain a new memetic algorithm. Our algorithm outperforms kKaHyPar-E and the three most successful ML HGP algorithms kKaHyPar, PaToH, and hMetis on a benchmark set, where each algorithm is given 2, 4, 8 hours. Out of 150 instances, our algorithm finds the best solutions in 112, 115, and 125 instances in 2, 4, and 8 hours, respectively. Our experimental study indicates that the solutions found by our algorithm in 2 hours are better than the solutions found by kKaHyPar, PaToH, and hMetis in 8 hours. Furthermore, the solutions found by our algorithm in 4 hours are better than the solutions found by kKaHyPar-E in 8 hours.
THE KAHYPAR FRAMEWORK

KahyPar is a general-purpose hypergraph partitioning framework developed over the series of papers [2, 8–10, 15, 16]. The latest version, referred to as kKaHyPar, is considered to be the best constructive heuristic [16]. We first present a high-level description of the kKaHyPar algorithm since both our algorithm and kKaHyPar-E use it as a subroutine.

kKaHyPar employs two preprocessing steps. It first contracts some pins of large nets so that the average pin size is reduced. It then detects the community structure of the hypergraph [10] via Louvain algorithm [3] to guide the coarsening process.

At each level of the coarsening stage, a node is randomly chosen and its contraction partner is selected from the eligible nodes for which the heavy-edge function [11] is maximized. Nodes \( u \) and \( v \) are eligible if they belong to the same community, and \( w(u) + w(v) \leq \kappa \), where \( \kappa \) is a predefined parameter. The coarsening stage ends when the hypergraph is small enough, or there is no pair of eligible nodes.

At the initial partitioning stage, kKaHyPar recursively bipartitions the coarsest hypergraph until a \( k \)-way partition is found [15] by using a pool of 9 randomized algorithms [16]. Each partition is refined using local search algorithms based on FM heuristics [7, 14], and the best partition is used as the initial partition. At the uncoarsening stage, kKaHyPar uses \( k \)-way FM heuristic along with a local search algorithm based on maximum flows (MF) [8, 9] to refine the partition.

kKaHyPar-E: It takes a time limit as input and spends 15% of the time limit to create the initial population using kKaHyPar. The initial population is evolved over generations with the steady-state paradigm [13] where only one offspring is created at each generation. It uses two mutation operators and two recombination operators, each of which is picked with a probability of 0.25. The individual that will be replaced by the offspring is chosen by a replacement strategy that considers both solution quality (fitness) and the similarity of individuals. Contrary to other evolutionary algorithms developed for the HGP problem, it uses problem-specific mutation and combination operators based on the V-cycles [19]. V-cycle method starts with a partition and coarsens only the nodes in the same blocks. This ensures that the starting partition is still valid and the partition is refined at the uncoarsening stage.

Recombination Operators: The first recombination operator \( C_1 \) works similar to the V-cycle method but it is more restrictive. First, it selects two parents using two-way tournaments. \( C_1 \) contracts nodes \( u \) and \( v \) if and only if they are placed to the same block in both parents. Then, the partition of the fitter parent is refined at the uncoarsening stage. \( C_1 \) ensures that the solution quality is maintained even if no improvements are found. But, this also means that the created offspring will be the same as the first parent if no improvements are found. The second recombination operator \( C_2 \) counts how many times each net is a cut net in the best \( \sqrt{p} \) individuals, where \( p \) is the population size. Then it changes the heavy-edge function such that it favors the node pairs that share a large number of small nets with low cut frequencies.

Mutation Operators: kKaHyPar-E uses two mutation operators \( M_1 \) and \( M_2 \). \( M_1 \) is the V-cycle method. \( M_2 \) is similar, but finds a new initial partition before the uncoarsening stage. \( M_1 \) ensures that the solution quality is maintained while \( M_2 \) does not.

MULTILEVEL MEMETIC ALGORITHM

Our memetic algorithm is composed of incorporating a greedy recombination operator \( C_3 \), and two mutation operators \( M_3 \) and \( M_4 \) to the kKaHyPar-E algorithm. As opposed to \( C_1 \), \( C_3 \) contracts a subset of nodes in the same block of one of the parents.

Greedy Recombination Operator: \( C_3 \) selects two parents by two-way tournaments. It sorts the blocks of each parent with respect to a quality measure. It selects the block with the highest quality and assigns all nodes in this block to a new cluster. It removes these nodes from the blocks of the other parent and recomputes the qualities of these blocks. This procedure is repeated until either each node is assigned to a cluster, or \( \frac{3k}{4} \) blocks are selected. The nodes that are not assigned to a cluster are assigned to a special cluster called 0. These clusters are used to guide the coarsening stage of the kKaHyPar algorithm as follows. At the coarsening stage, two nodes \( u \) and \( v \) are only allowed to be contracted if they belong to the same cluster, and this cluster is not cluster 0. The coarsening stage ends if there are no possible contractions. This operator uses kKaHyPar algorithm with this modified coarsening stage to create an offspring. \( C_3 \) selects no more than \( \frac{3k}{4} \) blocks since clustering the nodes in the remaining \( \frac{k}{4} \) blocks may lead to less ideal contractions. In addition to that, allowing all possible contractions may create a hypergraph with a low number of nodes with uneven weights, which may complicate the initial partitioning stage [12].

A rating function supporting the intuition behind the design of \( C_3 \) needs to satisfy the following conditions.

- The quality of a block should not depend on its size.
- It should rank a good block higher compared to a bad block.

A block is said to be good if it contains either a significantly high or low fraction of pins of incident nets. We use the rating function given below which satisfies these conditions. Let \( E_{i} = \bigcup_{v \in V_{i}} N(v) \).

\[
    r(V_{i}) = \frac{1}{|E_{i}|} \sum_{e \in E_{i}} \left( \frac{|\text{pins}(e) \cap V_{i}|}{|\text{pins}(e)|} \right)^{2}
\]

Notice that for \( k = 2 \), \( C_3 \) finds one of the parent solutions.

Mutation Operators: The mutation operators \( M_3 \) and \( M_4 \) cluster the nodes using an already computed partition. They use these clusters to create a new coarsening scheme by blocking some contractions. Operators first find the connected components in each block. If a connected component does not contain all pins of at least one net, then nodes on this connected component are assigned to the cluster 0. Then, the remaining nodes in each block are assigned to a new cluster.

\( M_3 \) only allows the contraction of the nodes in the same cluster except the nodes in the cluster 0. Since the starting partition is still valid, it is refined at the uncoarsening stage as in \( M_1 \). \( M_4 \) only allows the contraction of nodes \( u \) and \( v \) if they are assigned to the same cluster, or either \( u \) or \( v \) is assigned to the cluster 0. The coarsening stage is followed by the initial partitioning and uncoarsening stages as in \( M_2 \). Notice that if each block only contains a single connected component, and there is at least one non-cut net incident to a node in every block, \( M_3 \) and \( M_4 \) are identical to \( M_1 \) and \( M_2 \), respectively.
4 EXPERIMENTAL EVALUATION

Experimental Setup: We implemented C3, M3 and M4 using the KaHyPar framework. Our implementation is available at \(^{1}\). We performed all the experiments on a cluster with 3 machines each of which has 2 Intel Xeon 6148 12-core processors clocked at 2.4GHz and 384 GB RAM. The instances are solved in parallel such that one core and 8GB RAM are reserved for each instance.

Instances: We use Benchmark Set D of Andre et al. \(^{2}\) with 25 unweighted hypergraphs from three different domains. Each hypergraph is partitioned into \(k\) blocks where \(k \in \{4, 8, 16, 32, 64, 128\}\), and we set \(\epsilon\) to 0.03 as in \([2]\). Thus, we have 150 instances in total.

4.1 Impact of Algorithmic Components
To compare different configurations of our algorithm, we run each configuration 3 times with different random seeds and a 2-hour time limit. To save time, we only used the instances with 32 blocks. We used the same random seeds for each configuration so that they start with the same initial population. We use convergence plots to show how the mean connectivity of the best solution of each instance evolved over time for each configuration.

Convergence plots: We first compute the best solution found until time \(t\) for each instance and configuration. To combine the results obtained by different seeds for a single instance, we use arithmetic mean. To combine the results of different instances, we use geometric mean so that each instance can contribute equally. The x-axis shows the time, and the y-axis shows the mean connectivity of all instances.

The details of the configurations are given in Table 1. The first column lists the configuration names, and the last 2 columns give the selection probabilities of recombination and mutation operators. Columns 2 – 4 (Columns 5 – 8) show the selection probability of each recombination (mutation) operator when the recombination (mutation) operation is selected.

| Configuration Name | Operator Selection Probability | C | M |
|--------------------|--------------------------------|---|---|
| kKaHyPar-E         | 0.5 0.5 0 0.5 0.5 0.5 0.5 | 0.5 0.5 |
| MMA-M-0.5          | 0.4 0.2 0.4 0.25 0.25 0.25 0.25 | 0.5 0.5 |
| MMA                | 0.4 0.2 0.4 0.25 0.25 0.25 0.25 | 0.8 0.2 |
| MMA-G              | 0 0 1 0.25 0.25 0.25 0.25 | 0.8 0.2 |
| MMA-EQ-C           | 0.33 0.33 0.33 0.25 0.25 0.25 0.25 | 0.8 0.2 |

Figure 1 shows how the mean best solution of each configuration evolved over time. MMA-G, which uses a single recombination operator (C3) performs the worst. MMA-M-0.5, which uses all recombination and mutation operators performs slightly better than kKaHyPar-E, which shows the effectiveness of the new operators. MMA uses less mutation operation compared to MMA-M-0.5 and performs better. This shows that C3 contributes to the diversity of the population since lowering mutation probability decreases the performance of kKaHyPar-E \([2]\). The performance of MMA-EQ-C is similar to that of MMA. We choose MMA configuration as our final configuration since it performs the best.

\(^{1}\)Link to our code

4.2 Comparison with Other Heuristics
We compare MMA with kKaHyPar-E, kKaHyPar, PaToH 3.2 \([5]\) and hMetis-R \([11, 12]\). While MMA and kKaHyPar-E spend all of their time evolving a population, the others compute new partitions using different random seeds until the time limit is reached. We use kKaHyPar with the configuration in \([16]\) with at most 100 V-cycle iterations as in \([2]\). We used PaToH and hMetis-R with their default parameters and chose \(\epsilon\) for hMetis-R as described in \([2]\).

We run each algorithm 5 times with different random seeds for each instance with 2, 4 and 8 hour time limits. We run our algorithm and kKaHyPar-E with the same random seeds to mitigate the impact of randomness. We use performance plots \([2]\) to compare the performances of algorithms with the best found solution on a per-instance basis.

Performance plots: For each instance, the connectivity of the solution found by each algorithm is divided by that of the best solution for that instance. This is repeated for each instance and results are aggregated such that the plot shows the fraction of instances where each algorithm produces solutions that are \(r\) times worse compared to the best. The x-axis shows the quality relative to the best solution, and the y-axis shows the fraction of instances. For instance, a point \((r, f)\) for an algorithm \(A\) in a performance plot shows that the algorithm \(A\) finds solutions that are at most \(r\) times higher than the best solution in \(f\) fraction of the instances. Notice \(r\) is monotone non-decreasing. At \(r = 1\), performance plots show the fraction of instances where each algorithm found the best solution.

Table 2 shows the average improvements over kKaHyPar-E for different time limits and number of blocks. For instances with \(k \geq 32\), MMA performs 0.46\%, 0.57\%, 0.8\% better on average than kKaHyPar-E, in 2, 4, 8 hours, respectively. A Wilcoxon matched pairs signed rank test \([21]\) shows that the average improvements by MMA over kKaHyPar-E are statistically significant for all time limits \((p \leq 10^{-10})\).

Figure 2 shows that MMA outperforms all other state-of-the-art HGP tools. In all plots, kKaHyPar, PaToH, and hMetis-R are run with 8 hour time limit. MMA and kKaHyPar-E are run with 2, 4, and 8 hour time limits. MMA and kKaHyPar-E outperform all non-evolutionary algorithms even when they use a quarter of the
Table 2: Average improvement over kKaHyPar-E

| k | 2 Hour | 4 Hour | 8 Hour |
|---|--------|--------|--------|
| 4 | -0.11% | 0.04%  | -0.05% |
| 8 | 0.28%  | 0.46%  | 0.38%  |
| 16| 0.73%  | 0.71%  | 0.73%  |
| 32| 0.56%  | 0.71%  | 0.89%  |
| 64| 0.54%  | 0.54%  | 0.86%  |
| 128| 0.29% | 0.46% | 0.63% |
| all | 0.37% | 0.49% | 0.57% |

Figure 2: Performance plots comparing all algorithms

time non-evolutionary algorithms use. MMA finds a best solution in 107, 113, 125 instances, and kKaHyPar-E finds a best solution in 33, 36, 31 instances for 2, 4, 8 hours, respectively. kKaHyPar finds a best solution in 23, 17, 14 instances 2, 4, 8 hours, respectively. While PaToH finds the best solution for one instance, hMetis-R does not find the best solution in any of the instances. For each algorithm, the maximum performance gap with the best solution in any instance is as follows: 1% for MMA, 8% for kKaHyPar-E, 12% for kKaHyPar, 82% for PaToH, 284% for hMetis-R. The results show that MMA is the best choice for applications where the solution quality is the most important concern.

REFERENCES

[1] Utku Umur Acikalin, Bugra Caskurlu, Piotr Wojciechowski, and K Subramani. 2021. New Results on Test-Cost Minimization in Database Migration. In International Symposium on Algorithmic Aspects of Cloud Computing. Springer, 58–55.

[2] Robin Andre, Sebastion Schlag, and Christian Schulz. 2018. Memetic multilevel hypergraph partitioning. In Proceedings of the Genetic and Evolutionary Computation Conference. 347–354.

[3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. Journal of statistical mechanics: theory and experiment 2008, 10 (2008), P10008.

[4] Aydin Buluç, Henning Meyerhenke, Ilya Safro, and Christian Schulz. 2016. Recent advances in graph partitioning. Algorithm engineering (2016), 117–158.

[5] Umit V Catalyurek and Cevdet Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Transactions on parallel and distributed systems 10, 7 (1999), 673–693.

[6] James Cohoon, John Kairo, and Jens Lienig. 2003. Evolutionary algorithms for the physical design of VLSI circuits. In Advances in Evolutionary Computing. Springer, 683–711.

[7] Charles M. Fiduccia and Robert M Mattheyes. 1982. A linear-time heuristic for improving network partitions. In 19th design automation conference. IEEE, 175–181.

[8] Lars Gottesbüren, Michael Hamann, Sebastion Schlag, and Dorothea Wagner. 2020. Advanced Flow-Based Multilevel Hypergraph Partitioning. In 18th International Symposium on Experimental Algorithms.

[9] Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2019. Network Flow-Based Refinement for Multilevel Hypergraph Partitioning. ACM J Exp. Algorithmics 24, Article 2.3 (sep 2019), 36 pages. https://doi.org/10.1145/3329872.

[10] Tobias Heuer and Sebastian Schlag. 2017. Improving coarsening schemes for hypergraph partitioning by exploiting community structure. In 16th International Symposium on Experimental Algorithms (SEA 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[11] George Karypis, Rajat Aggarwal, Vinip Kumar, and Shashi Shekhar. 1999. Multilevel hypergraph partitioning: Applications in VLSI domain. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 7, 1 (1999), 69–79.

[12] George Karypis and Vinip Kumar. 2000. Multilevel k-way hypergraph partitioning. VLSI design 11, 3 (2000), 285–300.

[13] John McCall. 2005. Genetic algorithms for modelling and optimisation. Journal of computational and Applied Mathematics 184, 1 (2005), 205–222.

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

[15] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2016. K-way hypergraph partitioning via n-level recursive bisection. In 2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX). SIAM, 53–67.

[16] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. 2021. High-Quality Hypergraph Partitioning. arXiv preprint arXiv:2106.08696 (2021).

[17] K Subramani, Bugra Caskurlu, and Utku Umur Acikalin. 2019. Security-aware database migration planning. In International Symposium on Algorithmic Aspects of Cloud Computing. Springer, 103–121.

[18] K Subramani, Bugra Caskurlu, and Alvaro Velasquez. 2018. Minimization of test-costs in capacity-constrained database migration. In International Symposium on Algorithmic Aspects of Cloud Computing. Springer, 683–711.

[19] Chris Walshaw. 2008. Multilevel refinement for combinatorial optimisation: Boosting metaheuristic performance. In Hybrid Metaheuristics. Springer, 261–289.

[20] Sverre Wichlund. 1998. On multilevel circuit partitioning. In Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design. 505–511.

[21] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In Breakthroughs in statistics. Springer, 196–202.