Engineering Exact Reductions and the D-Core Heuristic for Hypergraph b-Matching

Jul Gundlach Valcárcel

November 5, 2026

4181590

Master Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Referee:
Prof. Dr. Felix Joos

Co-Supervisor:
Dr. Henrik Reinstädtler

Acknowledgments

Firstly, I would like to thank Prof. Christian Schulz for introducing to me to the topic of this thesis and the algorithm engineering world, as well as his valuable advices during the development of this thesis. Furthermore, a lot of thanks to Henrik Reinstädtler who discussed with me my research, also gave insightful advices and additionally provided a beautiful code framework. I thank Prof. Felix Joos for serving as my thesis refeere. Last but certainly not least, I would like to thank everyone who has supported me throughout my studies, with special thanks to my mother.

I hereby confirm, Jul Gundlach Valcárcel, Matr. No. 4181590, that this thesis has been written independently and without unauthorised external assistance. I have not used any sources or aids other than those specified, and I have identified all passages that have been taken verbatim or paraphrased from published or unpublished works, including those from digital or AI-based sources. I confirm that any use of AI tools was discussed in advance with the supervisor of the thesis and that the agreed rules were followed. I take full responsibility for the academic quality and content of this paper, the methodology chosen and the process of its creation, as well as the literature cited. I agree that my thesis will be checked for plagiarism and possible misuse of automated text and code generation as part of university procedures.

Heidelberg, November 5, 2026

Jul Gundlach Valcárcel

Abstract

Matching in graphs is a classical problem in computer science with application in fields like chemical computation and resource allocation. One variation to the classical problem is hypergraph b-matching. In this variation, the graph structure is generalized by allowing an edge to connect multiple vertices and the matching condition is relaxed by allowing some vertices to be incident to more than one matched edge, limited by a capacity function $b : V \to \mathbb{N}$. In contrast to the classical graph matching problem, hypergraph b-matching is not solvable in polynomial-time unless $P = NP$. To tackle this problem, this thesis presents exact reductions and adapts the D-Core heuristic to hypergraph b-matching. A technique that heuristically decreases the solution space and allows to concentrate the computational resources on a smaller instance. It determines a subgraph by computing multiple solutions and selecting the edges for which the solutions do not agree on. In addition, we propose an initialization and perturbation technique as well as parallel (sub)algorithms. We contribute seven exact data reductions (two new reductions, two adapted from independent set, three generalizations). Based on that, we propose parallel algorithms for the reductions. Finally, we present two improvements in the local search procedure. We show experimentally that for $b \equiv 1$ the newly implemented reductions decrease the kernel size on average by 25% and the parallel algorithms of the reductions are capable of speeding up the process on average by 32 times with 64 processing elements. Solving the integer linear program of hypergraph b-matching with reductions improves the runtime by an average factor of 2.26 for uniform capacities. Additionally, the experiments show that the local search procedure benefits from the improvements, finding solutions that are up to 7.61% better, with an average of 0.57%. For uniform capacities, the D-Core heuristic is capable of computing on average solutions with 99.8% optimal quality for the hypergraphs for which we could retrieve the optimal solution. Compared to the local search, it reduces the average gap to the optimal value by a factor of 3.5 with total improvements of up to 6.3%. Using 16 processing elements, the D-Core algorithm computes solutions with the same quality as those returned by the single-core local search on average 6 times faster. We observe similar results for other b-functions. For random capacities, Gurobi finds the best solutions on most instances.

Contents

Abstract v

1 Introduction 1
1.1 Motivation . . . 1
1.2 Our Contribution . . . 2
1.3 Structure . . . 3

2 Fundamentals 5
2.1 Structure Definition . . . 5
2.2 Problem Definition . . . 6
2.3 Data Reductions . . . 7
2.4 Linear Programming . . . 7

3 Related Work 9
3.1 Matchings . . . 9
3.2 Hypergraph Matching . . . 10
3.3 Independent Set . . . 10
3.4 Hypergraph b-Matching . . . 11

4 Reductions 13
4.1 Isolated Weight Transfer . . . 13
4.2 Weighted Edge Folding . . . 16
4.3 Tuple Folding . . . 18
4.4 General Tuple Folding . . . 19
4.5 Subset Vertices . . . 20
4.6 Subset Duo Vertices . . . 21
4.7 General Weighted Domination . . . 22
4.8 Parallelization . . . 24

5 Heuristic Approaches 27
5.1 D-Core Algorithm . . . 27
5.2 Local Search . . . 31

6 Experimental Evaluation **33**
6.1 Hardware 33
6.2 Instances 33
6.3 Methodology 34
6.4 Experiments 35
6.4.1 Reductions 35
6.4.2 Local Search 39
6.4.3 D-Core Algorithm 39

7 Discussion **49**
7.1 Conclusion 50
7.2 Future Work 51

A Further Results **53**

Abstract (German) **55**

Bibliography **57**

CHAPTER 1

Introduction

1.1 Motivation

Matching in general graphs is a classical and well-studied combinatorial problem in computer science. Given a graph, a matching is a subset of the set of edges such that no two edges in the subset share a vertex. A popular objective is to find the largest matching in a graph. This problem is solvable in polynomial-time, which was first shown for general (non-bipartite) graphs by Edmonds [19]. It has applications in a variety of fields such as in processor assignment problems [23], in chemical computations [44] and job assignment [53]. Matchings also showed to be useful for combinatorial problems such as in a $\frac{3}{2}$-approximation algorithm for the metric traveling salesman problem [9]. The matching problem exists in several variations such as matching in bipartite graphs, edge coloring or hypergraph matching. Unlike conventional graphs, in a hypergraph a (hyper)edge is allowed to connect more than two vertices. This generalization increases the computational effort for finding matchings. Hypergraph matching also known as set packing has been shown to be NP-complete [37] which leads the research in this topic to techniques such as kernelizations, approximations, and heuristics. Its close relation to the independent set problem allows to adapt concepts and ideas directly to hypergraph matching and vise versa. For example, reductions and local search developed for independent sets [42] were transformed to hypergraph b-matching by Großmann et al. [30]. This success is a motivational factor to adapt the D-Core heuristic [29] to hypergraph matching. Applications of hypergraph matching include computer vision [57], scheduling in surgical procedures [54], public transportation [31] and resource allocation [58]. The matching problem can be further relaxed by allowing some vertices to be next to multiple edges in the matching. This problem is known as b-matching. It is studied in general graphs and hypergraphs. The version for hypergraphs is used for example for auctioning goods [11], and hypergraph clustering [48].

1.2 Our Contribution

We summarize our main contributions of this thesis. The three main parts of this thesis are reductions, the D-Core heuristic, and an experimental evaluation.

1. We present seven novel contribution for exact reductions that can be applied to the hypergraph b-matching problem. This includes adapting a reduction from the independent set problem to hypergraph b-matching, namely Isolated Weight Transfer. We show that the Weighted Edge Folding reduction can be extended to cover more cases. Additionally, we adapt "Extended Twins" to hypergraph b-matching and generalize it twice. Once with respect to the number of folded edges and once with respect to dropping the independent property. We present two new vertex reductions, namely Subset Vertices and Subset Duo Vertices. Finally, we show a generalization of Weighted Domination and an improvement in runtime complexity for finding edges for Weighted Domination. We present algorithms for identifying the reductions in hypergraphs as well as parallelization techniques. For the implementation, we extend the existing code basis of the six reductions by Großmann et al. [30].

2. Our main contribution is the adaptation of the D-Core heuristic to hypergraph b-matching. This includes a detailed description of how the algorithm is implemented. Additionally, we present a randomized greedy algorithm that initializes the base solutions and a perturbation technique in case the D-Core is empty. We present a detailed explanation and analysis on how to parallelize the D-Core set calculation. Furthermore, we provide two improvements for the local search procedure, namely $(1, 1)$-Swaps and a copying algorithm. The implementation of these algorithms is also an extension of the framework by Großmann et al. [30].

3. We present comprehensive experimental evaluations of the implemented reductions and heuristic. For this, the quality of the algorithms is tested on 584 hypergraph instances. Specifically, we analyze the kernel size after exhaustively applying the reductions. For $b \equiv 1$, the new reductions shrink the average kernel size (in edges) by 24.6%; for $b \equiv 3$ by 5.1%; and for random b by 20.1%. The speedup of the parallel algorithm for finding the reductions with 64 threads is evaluated and shows for $b \equiv 1$ that on average the reduction can be computed 32 times faster. In addition, we analyze how the reductions improve the runtime for exactly solving the ILP of hypergraph b-matching, seeing average speedup of 2.26 for $b \equiv 1$; 2.21 for $b \equiv 3$; 1.94 for random b. For the D-Core heuristic, we test for a variety of parameters the quality and runtime and we compare the D-Core algorithm to an optimization tool, local search and a state-of-the-art algorithm. For all tested b-functions, the D-Core outperforms the local search algorithm with respect to runtime and quality on almost all instances. We furthermore evaluate how close the solutions computed by the D-Core are to the optimal values.

1.3 Structure

The remainder of this thesis is organized as follows. In Chapter 2, we introduce the notions and basic definitions that revolve around hypergraph b-matching. Chapter 3 references scientific work that is closely related to hypergraph b-matching which includes a brief explanation of algorithms and reductions that are part of the implementation but not topic of this thesis. The theoretical approaches are divided into reductions and heuristic approaches. In Chapter 4, we present seven contributions to reductions, their proofs, and algorithms to find them. Following up with Chapter 5, we introduce the D-Core algorithm as well as some improvements of the local search procedure. We test the implementation of the algorithms in Chapter 6. Additionally, we provide a comparison to data received by a state-of-the-art algorithm in the appendix, Chapter A. We end the thesis in Chapter 7 with a discussion of the results and an outlook for future work.

CHAPTER 2

Fundamentals

In this chapter, we introduce the general definitions and notations for the thesis. This includes structure definitions in Section 2.1 and problem definitions in Section 2.2, as well as the notion of data reduction in Section 2.3 and linear programming in Section 2.4.

2.1 Structure Definition

In the following section, we introduce hypergraphs and graphs as basis for the problems considered in this work.

Hypergraph. An undirected weighted hypergraph H is a triple (V, E, w) where V is a set of n vertices, E is a set of m hyperedges, and w is a function $E \to \mathbb{R}_{>0}$. A hyperedge is identified by a unique natural number. Every hyperedge is mapped to a subset of V. This setup allows us to have multiple hyperedges with the same vertex set but different properties such as hyperedge weight. If not otherwise stated, the symbol of a hyperedge directly refers to its vertex set. For simplicity, we refer to hyperedges as edges. Given a hypergraph H, we refer to $V(H)$, $E(H)$ and w_H as the vertices of the hypergraph, the edges of the hypergraph and the weight function of the hypergraph, respectively. We say a vertex v is incident to an edge e if v is in e. For a vertex v, we define the set of incident edges of v as $E_H(v) = \{e \in E(H) \mid v \in e\}$. When the underlying hypergraph is clear from the context, we write $E(v)$ instead of $E_H(v)$. The degree of v, $deg(v)$, is $|E(v)|$. We denote Δ_V the maximum degree in the hypergraph. Two vertices v, w are adjacent if v and w are distinct and there is an edge e such that v and w are incident to e. For an edge e, we define the edge size as $|e|$. We denote Δ_E the maximum edge size in the hypergraph. Two edges u, v are adjacent if their intersection is non-empty and u and v are distinct. For an edge e, we define $N(e) = \bigcup_{v \in e} E(v)$ as the closed neighborhood of e. For a subset E' of E, we define $N(E') = \bigcup_{e \in E'} N(e)$. For a subset E' of E, the weight of E', $w(E')$, is $\sum_{e \in E'} w(e)$. A set $N \subseteq E$ is an independent set if every pair of distinct edges $f, g \in N$

is non-adjacent. A hypergraph is d-uniform if all edges are of size d. A hypergraph is d-partite if there is a partition of V into d subsets $V_1, ..., V_d$ and every edge has at least one vertex in every V_i. Given an edge set $E' \subseteq E$, the hypergraph induced on E' is defined as a hypergraph (V, E', w). For a set $V' \subseteq V$, let E' be the edges in E modified such that for every edge e in E', the vertices in $V \backslash V'$ are removed from e. The induced hypergraph on V' is (V', E', w). An undirected graph is the special case of a hypergraph in which every edge has exactly two vertices.

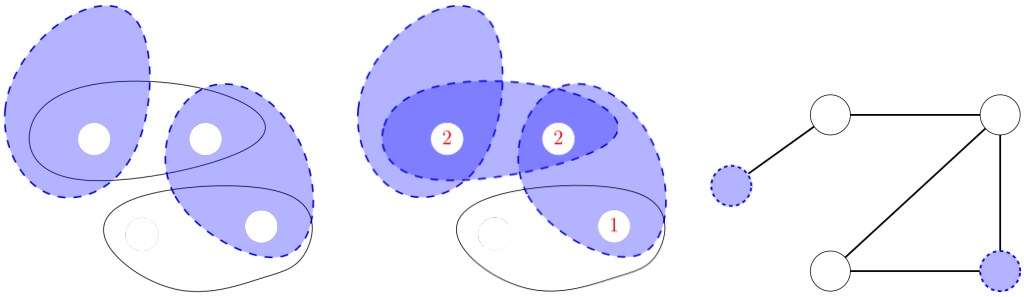
2.2 Problem Definition

In this section, we introduce the problems of this thesis, hypergraph matching and hypergraph b-matching, as well as the closely related independent set problem.

Matching. For a hypergraph $H = (V, E, w)$, a matching M is a subset of E such that no two edges in M are adjacent. We refer to $\mathcal{M}_H$ as the set of all matchings in H, the solution space. An edge is matched iff. it is in the matching. An edge is free if all adjacent edges are not matched. An edge is blocked if it is not free. We say a matching M is maximal if there is no free edge. A maximum cardinality matching M is a matching such that for all $M_2 \in \mathcal{M}_H$ M_2 has less or equal number of elements compared to M. A maximum-weight matching M is a matching such that for all $M_2 \in \mathcal{M}_H$ the weight of M_2 is smaller or equal to the weight of M. The decision problem of hypergraph matching inputs a hypergraph H and a natural number k, and asks whether there is a matching of size k in H.

b-Matching. For a hypergraph $H = (V, E, w)$ and function $b : V \to \mathbb{N}_0$, a b-matching M is a subset of E such that for every vertex v at most $b(v)$ edges in $E(v)$ are in the matching. The b-value of a vertex is also referred to as its capacity. In case $b \equiv 1$, a b-matching satisfies the same conditions as a matching. We use the notion of matchings also for b-matchings with the exception: An edge e is free if for every vertex v in e, $|M \cap E(v)| < b(v)$. A vertex v is blocking (or at full capacity) if $|E(v) \cap M| = b(v)$. The residual capacity of a vertex v is $b(v) - |M \cap E(v)|$.

Independent Set. For a graph $G = (V, E)$, an independent set I is a subset of V such that no two vertices in I are adjacent. The maximum cardinality independent set asks for an independent set with the highest cardinality. For a function $w : V \to \mathbb{R}$, the maximum-weight independent set problem asks for an independent set that has the highest combined weight defined as $\sum_{v \in I} w(v)$. There is a direct polynomial-time reduction [26] from the independent set problem to the set packing/hypergraph matching problem where n vertices are transferred to n (hyper)edges. Figure 2.1 shows visual examples for the three presented problems.



(a) Hypergraph Matching **(b)** Hypergraph b-Matching **(c)** Independent Set

Figure 2.2: Visual examples for the presented problems. A blue edge and vertex is in a matching and an independent set, respectively.

2.3 Data Reductions

Data reduction is a technique commonly used in the preprocessing of NP-hard problems. For an optimization problem p, it maps in polynomial-time an instance x of p to an instance x' of p such that an optimal solution for x' can be transformed in polynomial time to an optimal solution for x. The objective is to find data reductions that return instances that are smaller in size, thus improving the runtime of the algorithms that solve the problem. The reduced instance is called kernel. In hypergraph b-matching, data reductions can be classified into three categories. In the first category, a reduction determines whether an edge is necessarily in an optimal matching or in no optimal matching, which allows to remove this edge. The second category consists of foldings. A set of edges N is folded if there is a disjoint set of edges N', possibly containing newly introduced edges, such that given an optimal solution for the hypergraph without N, the solution values of N' admit the optimal values for N in the original hypergraph. In the third category, an edge is removed from the hypergraph if it is known that an optimal matching with the same size/weight as in the original hypergraph can be found in the kernel.

2.4 Linear Programming

All previously discussed problems can be formalized as (integer) linear programs. A linear program is defined by a vector of variables $(x_1, .., x_n) \in \mathbb{R}^n$, a linear function $f : \mathbb{R}^n \to \mathbb{R}$, that is to be minimized or maximized, and linear equality and inequality constraints. Solving a linear program is finding a feasible vector x such that $f(x)$ is the optimal value among all vectors that satisfy the constraints. If the domain of the variables are real values, then it is referred to as a linear program (LP) and it is solvable in polynomial-time [36]. If the

domain of the variables are integer values, then the problem is called integer linear program (ILP). Polynomial-time reductions from NP-hard problems to integer linear programs such as the SAT reduction [46] show that solving an integer linear program is NP-hard. Großmann et al. [30] present an integer linear program for the maximum weighted hypergraph b-matching problem.

CHAPTER 3

Related Work

In this chapter, we reference and summarize scientific work that cover hypergraph matching and other closely related topics.

3.1 Matchings

Matching on general graphs is a well studied problem in combinatorial optimization. Both the maximum cardinality and maximum-weight matching problem are proven to be solvable in polynomial-time [19] [20].

Exact Algorithms. Micali and Vazirani [45] introduced an algorithm that solves the maximum matching problem in $O(\sqrt{n}m)$ by iteratively finding paths in the graph that can be used to improve the matching. One way to solve the weighted case is by the scaling method, which limits the edge weights by an iteratively increasing upper bound. This leads to an algorithm [17] that solves the maximum-weight matching problem in $O(\sqrt{n}m \log(nN))$ with N being the largest edge weight.

Approximations. Preis [50] introduced a $\frac{1}{2}$-approximation algorithm for the maximum weighted matching problem that runs in linear time in the number of edges. Drake and Hougardy [15] provide a simpler algorithm with the same approximation guaranty, that does not require an amortized analysis of the runtime. This was improved in runtime and generalized to the maximum weighted b-matching problem by Khan et al. [39]. Duan et al. [16] present an $(1-\epsilon)$-approximation algorithm for maximum weighted matching which runs in $O(m\epsilon^{-1} \log \epsilon^{-1})$ for any $\epsilon > 0$. Birn et al. [7] provide a parallel algorithm that finds a maximal matching in $O(log^2 n)$ expected running time in the CREW PRAM model. Maue and Sanders [43] show an approximation algorithm that performs well in practice with respect to solution quality and runtime even though it does not have the best theoretical performance.

3.2 Hypergraph Matching

The decision problem of hypergraph matching is NP-complete. It is a graphical representation of the set packing problem, which is one of Karp's 21 NP-complete problems [37].

Approximations. In contrast to matching in general graphs, the d-uniform maximum hypergraph matching problem was shown to be MAX-SNP-hard by Kann [35], which implies that it does not admit a polynomial-time approximation scheme (PTAS) unless $P = NP$. A PTAS algorithm has a $(1 - \epsilon)$-approximation ratio for any ϵ. Håstad [34] proved that (for n vertices) there is no polynomial-time $n^{1-\epsilon}$ approximation algorithm for maximum clique for any constant $\epsilon > 0$ unless $NP \subseteq ZPP$. Thus, this also holds for maximum hypergraph matching (transferred to m edges). According to Hazan et al. [33], the d-set packing problem cannot be efficiently approximated within a factor of $\Omega(d/\ln d)$ unless $P = NP$. Thus, this also holds for d-uniform hypergraph matching. Füredi et al. [24] showed that a linear program for unweighted hypergraph matching returns a solution that is bounded by $\sum_{e \in M_{opt}} (|e| - 1 + \frac{1}{|e|})$. In case of a LP for d-uniform weighted hypergraphs, the bound is $(d - 1 + \frac{1}{d})w(M_{opt})$. If the hypergraph is weighted, d-partite and d-uniform, then the bound for a LP is $(d - 1)w(M_{opt})$. Anegg et al. [4] show a bound for LP of the maximum (non-uniform, non-partite) weighted hypergraph matching problem. Cygan [10] developed a local search technique which bounds the maximum matching problem in d-uniform hypergraphs by a ratio of $(d + 1 + \epsilon)/3$. Fürer and Yu [25] improve this result with respect to run-time. Chan and Lau [8] investigate strengthening of the linear program by local constraints. Besser and Poloczek [6] proved that no deterministic priority algorithm achieves a better ratio than $\frac{1}{d}$. For hypergraphs that are so large that they do not fit into memory entirely, Reinstädtler et al. [51] propose two semi-streaming algorithms, one having an approximation guarantee of $\frac{1}{d(1+\epsilon)}$ for d-uniform hypergraphs with $\epsilon > 0$.

Heuristic Approaches. Dufossé et al. [18] presented five heuristics for the unweighted matching problem on d-partite, d-uniform hypergraphs, including an adaptation of the Karp-Sipser heuristic [38] to hypergraphs. Reinstädtler et al. [52] introduce a parallel algorithm for maximal matchings on hypergraphs which runs in $O((\log \Delta_V + \log d) \log m)$ in a CREW PRAM model. In the CRCW PRAM model, the parallel algorithms run in $O(\log \log \Delta_V \log m)$ and work in $O(\kappa \log m)$ with κ the sum of all vertex degrees.

3.3 Independent Set

Exact Approaches. One approach to finding an optimal solution to an NP-hard problem is to systematically explore the solution space, for example, using branch-and-bound algorithms. A branch is a sub-solution space defined by including or excluding vertices in or from the solution, respectively. This kind of algorithm aims to reduce the search space

with upper and lower bounds on the branches, as shown in [56]. This was further improved by including reductions in the branching steps, the so called branch-and-reduce procedure, as presented by Lamm et al. [42]. Branch-and-transform [27] is an approach that changes the graph structure by adding more vertices allowing more reductions to be applied later on in the branching steps.

Heuristic Approaches. Local Search is a common technique for improving sub-optimal solutions of NP-hard problems. In combination with the local search technique, $(1, 2)$-Swap, originally proposed by Andrade et al. [3], Großmann et al. [29] introduce a new meta-heuristic for maximum weighted independent sets. This meta-heuristic is called Difference-Core, short D-Core, and is the main basis for this paper. We aim to apply the same technique to Hypergraph Matching. Dong et al. [14] use a greedy randomized adaptive search procedure (GRASP) and path-relinking as meta-heuristics to find maximum-weight independents sets. The hybrid iterative local search by Nogueira et al. [47] makes use of $(w, 1)$-Swaps and $(1, 2)$-Swaps.

3.4 Hypergraph b-Matching

Approximations. El Ouali and Jäger [21] generalize the result by Kann [35] that d-uniform hypergraph matching problem is MAX-SNP-hard to the d-uniform hypergraph b-matching problem. For the d-uniform maximum cardinality hypergraph b-matching, Parekh and Pritchard [49] gave a $(d - 1 + \frac{1}{d})$-algorithm. El Ouali et al. [22] showed that for any $2 \leq b' \leq \frac{d}{\log d}$ there is no polynomial-time approximations within any ratio smaller than $\Omega(\frac{d}{b' \log d})$ for the b-matching problem on d-uniform hypergraphs where $b \equiv b'$. For weighted b-matching on d-uniform hypergraphs, Krysta [41] gave a greedy $(d + 1)$-approximation algorithm. For the weighted d-uniform hypergraph matching problem Koufogiannakis and Young [40] present distributed d-approximation algorithms, which for general d run in $O(\log^2 m)$ expected rounds.

Heuristic Approach. Großmann et al. [30] present reductions, foldings and local search techniques for weighted b-matching on general hypergraphs, which are used in this thesis as tools for the D-Core algorithm. We briefly summarize the local search algorithm. Table 3.1 presents a short summary of the existing reductions.

The local search procedure is a repetition of iteration steps until some condition is met. This condition is a time limit and a maximum number of consecutive rounds without improvement. An iteration step consists of executing a perturbation after which $(1, 2)$-Swaps are exhaustively performed. The perturbation is a technique to overcome local optima and is implemented as follows: a random edge is added to the solution and the adjacent edges are adjusted such that the b-matching remains correct. A $(1, 2)$-Swap is an algorithm that removes an edge e from the b-matching in order to add two edges to the b-matching that

became free due to the removal of e. For three edges that can perform a $(1, 2)$-Swap, in the unweighted case, it improves the solution by one. In the weighted case, a $(1, 2)$-Swap is feasible if the weight of e is strictly smaller than the combined weight of the two edges that became free. The candidate version of the local search procedure improves the runtime of searching (1,2)-Swaps. The goal is to minimize the number of matched edges that are checked for a $(1, 2)$-Swap while not missing a possible $(1, 2)$-Swap. Its intuitive idea is that after exhaustively applying $(1, 2)$-Swaps a new $(1, 2)$-Swap only occurs close to a change like a perturbation. So it is not necessary to check all edges.

Reduction Name	Short Description
Weighted Isolated Edge Removal (WIER)	This reduction identifies an edge e such that its closed neighbourhood forms a clique. If edge e has the largest weight in the neighbourhood, then e is in an optimal matching.
Neighbourhood Removal (NR)	Sum over every vertex v of an edge the $b(v)$th heaviest edge. If the weight of the edge is larger than or equal to this sum, then the edge is an optimal matching.
Weighted Domination (WD)	For two edges e, f such that e is a subset of f and the weight of e is larger than or equal to the weight of f and there is a vertex with capacity one that is in both edges, then f can be removed from the hypergraph.
Abundant Vertices (AV)	A vertex whose capacity is larger than or equal to its number of incident edges can be removed from the hypergraph.
Weighted Edge Folding (WEF)	This reduction folds an edge e with its open neighbourhood N, if N is independent and the weight of e is smaller than $w(N)$ but strictly larger than $w(N) - min_{f \in N} w(f)$.
Weighted Twin (WT)	If two non-adjacent edges e, f have the same open neighbourhood N and N is independent, and $w(e) + w(f) < w(N)$ and $w(e) + w(f) > w(N) - min_{g \in N} w(g)$, then e, f and N can be folded together.

Table 3.1: A summary of the existing reductions

CHAPTER 4

Reductions

In this chapter, we present the theoretical results of the reductions for the hypergraph b-matching problem. Großmann et al. [30] presented six exact reductions for hypergraph b-matching which we expand up on. We adapt a folding to hypergraph b-matching that was originally a technique for independent sets presented by Lamm et al. [42] under the name "Isolated Weight Transfer". The name remains unchanged for the adapted folding and the correctness of the reduction is demonstrated. We provide a proof that shows that "Weighted Edge Folding" can be extended to cover more cases, and we adapt and generalize the "Extended Twin" reduction by Großmann et al. [29]. Additionally, we present two new exact reductions and their proofs, namely "Subset Vertices" and "Subset Duo Vertices". We generalize "Weighted Domination" and show an improvement in runtime complexity for finding edges for "Weighted Domination". Finally, we propose shared-memory parallelization algorithms for these reductions.

4.1 Isolated Weight Transfer

The following reduction is a generalization of "Weighted Isolated Edge Removal" [30]. We drop the requirement that the "isolated" edge has the largest weight in the "clique". For details on the version for the independent set problem, we refer to Lamm et al. [42]. Briefly stated, in the independent set problem, the reduction identifies a vertex v with a set of adjacent vertices N such that, for every pair of distinct vertices in $N \cup \{v\}$, there is an edge in the graph (namely the clique property). The reduction allows to remove all vertices in N that have a smaller or equal weight compared to v. If there is no vertex in N that has a higher weight than v, then v is an optimal solution. If there are vertices in N that have a higher weight than v, then v is removed from the graph and the weight of the remaining vertices is decreased by $w(v)$. For the reconstruction, if some vertex in N is in the optimal solution, then the independent set stays the same. Otherwise vertex v is added to the solution.

Reduction 1 (Isolated Weight Transfer)**.** *For a hypergraph $H = (V, E, w)$ and a function $b : V \to \mathbb{N}_{\geq 1}$, let e be an edge in the hypergraph such that for every pair $f, g \in N(e)$ there is a vertex v incident to f and g such that v has capacity 1. Then, the following statements hold.*

1. *For all $e_n \in N(e)$ such that $w(e_n) < w(e)$, there is no optimal b-matching M in H such that $e_n \in M$.*

2. *For all $e_n \in N(e)$ such that $w(e_n) = w(e)$, if there is an optimal b-matching M in H such that $e_n \in M$, then there is also an optimal b-matching M' in H such that $e \in M'$.*

3. *Let U be the edges in $N(e)$ that have a higher weight than e. Let H' be a hypergraph induced on $(E \backslash N(e)) \cup U$ and redefine the weights for every edge e_n in U with $w_{H'}(e_n) = w(e_n) - w(e)$. Let $M_{H'}$ be an optimal b-matching in H'. If some edge in U is matched, then $M_H = M_{H'}$ is an optimal solution in H. If no edge in U is matched, then $M_H = M_{H'} \cup \{e\}$ is an optimal b-matching in H. In both cases, the resulting b-matching has weight $w(M_H) = w_{H'}(M_{H'}) + w(e)$.*

Proof. For **Statement 1**, assume that there is an optimal b-matching M containing an edge e_n such that $e_n \in N(e)$ and $w(e_n) < w(e)$. Since edge e_n is in the matching all edges in $N(e) \backslash \{e_n\}$ are not the in the matching. Therefore, all vertices in e that are at full capacity are also in e_n, and we can construct a new b-matching $M' = (M \backslash \{e_n\}) \cup \{e\}$. It follows that $w(M') = w(M) - w(e_n) + w(e) > w(M)$, which is a contradiction to the optimality of M and proves Statement 1.
For **Statement 2**, assume that there is an optimal b-matching M containing an edge e_n such that $e_n \in N(e)$ and $w(e_n) = w(e)$. Since edge e_n is in the matching all edges in $N(e) \backslash \{e_n\}$ are not the in the matching. Therefore, all vertices in e that are at full capacity are also in e_n, $M' = (M \backslash \{e_n\}) \cup \{e\}$ is b-matching in H and it follows that $w(M') = w(M)$, which proves Statement 2.
For **Statement 3**, if U is empty, then this reduction satisfies the same conditions as for weighted isolated edge removal. Suppose that U is non-empty. We first show that M_H is a b-matching in H. Given an optimal b-matching $M_{H'}$ in H'. For every vertex v in $V(H) \backslash e$, $E_{H'}(v) = E_H(v) \backslash (N(e) \backslash U)$. Since all edges in $(N(e) \backslash U)$ are removed from H', it follows that $(N(e) \backslash U) \cap M_{H'} = \emptyset$. Therefore, $|M_H \cap E_H(v)| = |M_{H'} \cap E_{H'}(v)| \leq b(v)$. It remains to consider the vertices in e. Let some edge e_n in U be matched. Then for every vertex $v \in e$, the number of matched incident edges stays the same when passing from H' to H. Therefore, for every vertex v in $V(H)$, $|M_H \cap E_H(v)| = |M_{H'} \cap E_{H'(v)}| \leq b(v)$. If $U \cap M_{H'} = \emptyset$, then the number of matched incident edges for the vertices in e is 0. Therefore, e can be added to the matching in H. Hence M_H is b-matching in H.
We now show that $w(M_H) = w_{H'}(M_{H'}) + w(e)$. If some edge e_n in U is matched, then all edges in $U \backslash \{e_n\}$ are unmatched. The weight of e_n in H is $w(e_n) = w_{H'}(e_n) + w(e)$. For every edge e' in $M_{H'} \backslash \{e_n\}$, the weight remains unchanged when going from H' to H.

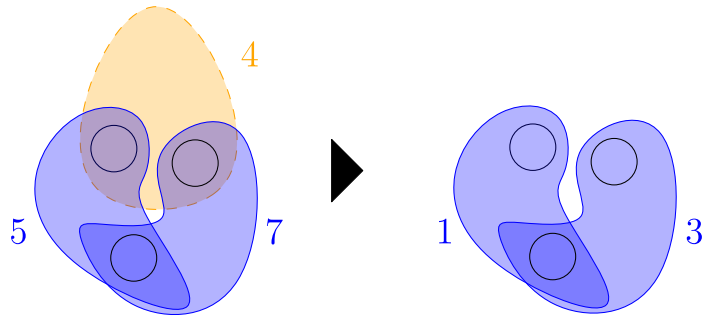


Figure 4.1: Isolated Weight Transfer. The vertices have a b-value of one. The numbers next to the edges represent the edge weights. The orange edge is "folded" with the two blue edges.

Therefore $w(M_H) = w_{H'}(M_{H'}) + w(e)$. If no edge in U is matched, then e can be added to $M_{H'}$ and it follows that $w(M_H) = w_{H'}(M_{H'}) + w(e)$.
We prove the optimality by contradiction. Assume that there is an optimal b-matching M in H such that $w(M) > w(M_H)$. Therefore, $w(M) > w(M_{H'}) + w(e)$. The matching M falls in two categories.

1. If $e \in M$, then $M \backslash \{e\}$ is a b-matching in H' with weight $w(M) - w(e)$. This contradicts the optimality of $M_{H'}$.

2. If $e \notin M$, then some edge e_n in $N(e)$ is matched. Otherwise M is not optimal. By statement 1, $w(e_n) \geq w(e)$. If $w(e_n) = w(e)$, by statement 2 we can create a new b-matching M'' with the same weight such that $e \in M''$, which leads to the same contradiction as in case 1. If $e_n \in U$, then M is also a b-matching in H'. In hypergraph H', the edge e_n has a weight of $w(e_n) - w(e)$. Therefore, the total weight of M in H' is $w(M) - w(e)$, which contradicts the optimality of $M_{H'}$.

□

Remark. Algorithm 1 specifies a procedure that finds edges for this reduction. This algorithm has a worst-case time complexity in $O(m\Delta_E^2\Delta_V^2)$ and requires additional memory in $O(m)$. Figure 4.1 is a visualization of an edge that applies to "Isolated Weight Transfer". The generalization that this reduction contributes is only feasible for non-uniform weighted b-matching problems. In uniform weighted b-matching problems, for example, when all edge weights are one, no folding will be found since all edges in $N(e)$ have the same weight as e. To minimize computation time, we limit candidates by the edge size and by the number of adjacent edges.

Algorithm 1: Isolated Weight Transfer

Data: b-matching M, hypergraph $H = (V, E, w)$
$m_1 = [false] \times m$;
$m_2 = [0] \times m$;
for $e \in E$ **do**
  $C \leftarrow \{e' \in E | e' \cap e \neq \emptyset\}$;
  for $e' \in C$ **do**
    $m_1[e'] = true$;
  for $i \in \{0, ..., |C| - 1\}$ **do**
    $e' = C[i]$;
    for $v \in e'$ **do**
      if $b(v) = 1$ **then**
        for $e'' \in E(v)$ **do**
          if $m_1[e''] \wedge (m_2[e''] = i)$ **then**
            $m_2[e''] \leftarrow m_2[e''] + 1$
  $isAClique = true$;
  $largest = true$;
  for $e' \in C$ **do**
    if $m_2[e'] \neq |C|$ **then**
      $isAClique = false$
    if $w(e) < w(e')$ **then**
      $largest = false$
  if $isAClique$ **then**
    if $largest$ **then**
      $M \leftarrow M \cup \{e\}$;
    else
      $Fold(C)$;
  reset(m_1, m_2) by iterating over C;

4.2 Weighted Edge Folding

For the reduction "Weighted Edge Folding" [30], we show that it can be slightly extended to cover more cases. Specifically, the third condition can be changed from $w(e) > w(N) - min_{f \in N} w(f)$ to $w(e) \geq w(N) - min_{f \in N} w(f)$.

Reduction 2 (Weighted Edge Folding)**.** *For a hypergraph $H = (V, E, w)$ and a function $b : V \to \mathbb{N}_{\geq 1}$, let e be an edge and let $N = N(e) \backslash \{e\}$ be the adjacent edges such that*

1. *for every edge e' in N, there is a vertex with capacity 1 in $e \cap e'$.*
2. *N is independent,*
3. *and $w(N) > w(e)$ and $w(e) \geq w(N) - min_{f \in N} w(f)$.*

Given these conditions, let H' be an hypergraph induced by $(E \backslash N(e)) \cup \{e'\}$ with $e' = \cup_{f \in N} f$ and $w(e') = w(N) - w(e)$. Let $M_{H'}$ be an optimal b-matching in H'. If $e' \in M_{H'}$, then $M_H = M_{H'} \backslash \{e'\} \cup N$ is an optimal b-matching in H. If $e' \notin M_{H'}$, then $M_H = M_{H'} \cup \{e\}$ is an optimal solution in H.

Proof. We first show that M_H is a b-matching in H. Let $M_{H'}$ be an optimal b-matching in H'. For every vertex $v \in V \backslash (e' \cup e)$, $E_{H'}(v) = E_H(v)$. Hence, $|M_H \cap E_H(v)| = |M_{H'} \cap E_{H'}(v)| \leq b(v)$. If e' is not in $M_{H'}$, then all vertices in e have 0 matched incident edges. The vertices in $e' \backslash e$ do not get an additional matched incident edge when passing from H' to H. Hence $M_{H'} \cup \{e\}$ is b-matching in H. If e' is in $M_{H'}$, then for every vertex v in e', there is one additional matched edge and one matched edge less in $E_H(v)$ compared to $E_{H'}(v)$. For every vertex in $e \backslash e'$ there is no additional matched incident edge when passing from H' to H. Hence, for every v in $e' \cup e$, $|M_H \cap E_H(v)| = |M_{H'} \cap E_{H'}(v)| \leq b(v)$. Therefore, M_H is a matching in H.
We now show that $w(M_H) = w(M_{H'}) + w(e)$. If e' is in $M_{H'}$, then N replaces e' which increases the weight by $w(e)$. If e' is not in $M_{H'}$, then e is added to $M_{H'}$ which increases the weight by $w(e)$.
We show the optimality by contradiction. Assume that there is an optimal b-matching M' in H such that $w(M') > w(M_H)$. Thus, $w(M') > w(M_{H'}) + w(e)$. The matching falls in two categories.

1. If $e \in M'$, then $M'' = M' \backslash \{e\}$ is a b-matching in H' with weight $w(M') - w(e)$. This is a contradiction to the optimality of $M_{H'}$.
2. If $e \notin M'$, then some edges in N are matched. Let N' be the matched edges in N. It holds that $w(N') \geq w(e)$. Otherwise M' is not optimal. For the case that $w(N') = w(e)$, $M'' = M' \backslash N'$ is a b-matching in H' and it holds that $w(M'') = w(M') - w(N') = w(M') - w(e) > w(M'_H)$. This a contradiction to the optimality of $M_{H'}$. If $w(N') > w(e)$, then $N' = N$. $M'' = (M' \backslash N') \cup \{e'\}$ is a b-matching in H' and it holds that $w(M'') = w(M') - w(N) + w(e') = w(M') - w(N) + w(N) - w(e) = w(M') - w(e) > w(M_{H'})$. This a contradiction to the optimality of $M_{H'}$.

□

Remark. This improved version of the reduction in combination with Isolated Weight Transfer removes all edges that have exactly two adjacent edges in the unweighted hypergraph matching problem. For an edge e with two adjacent edges u, v, if u, v are non-adjacent, then weighted edge folding can be applied. If u, v are adjacent, then Isolated Weight Transfer can be applied. To identify edges for this reductions, we use the same algorithm as proposed by Großmann et al. [30].

4.3 Tuple Folding

The following reduction is an adaptation and generalization of "Extended Twin" by Großmann et al. [29]. We adapt "Extended Twin" from the independent set problem to hypergraph b-matching and we do not limit the number of edges that are folded together.

Reduction 3 (Tuple Folding)**.** *For a hypergraph $H = (V, E, w)$ and a function $b : V \to \mathbb{N}_{\geq 1}$, let N be a subset of E such that*

1. *N is independent,*
2. *$\forall f, g \in N$ it holds that $N(f) \backslash \{f\} = N(g) \backslash \{g\}$,*
3. *and for every edge $e \in N$ and every edge e' in $N(e)$, there is a vertex in $e \cap e'$ that has capacity 1.*

Let H' be a hypergraph induced on $(E \backslash N) \cup \{e'\}$ with $e' = \bigcup_{j \in N} j$ and $w(e') = w(N)$. Let $M_{H'}$ be an optimal solution in H'. If $e' \in M_{H'}$, then $M = (M_{H'} \backslash \{e'\}) \cup N$ is an optimal b-matching in H. If $e' \notin M_{H'}$, then $M = M_{H'}$ is an optimal b-matching in H. It holds that $w(M) = w(M_{H'})$.

Proof. The correctness is shown by Reduction 4 which is a generalization of this reduction. □

Remark. A weighted twin folding is a combination of this folding and weighted edge folding which is why we will not explicitly search for weighted twin anymore. Tuple folding can be combined with weighted edge folding to generalize the weighted twin folding to reductions such as weighted triple folding, weighted quadruple folding and so on. Figure 4.2 visualizes the tuple folding reduction. For $b \equiv 1$, Algorithm 2 specifies a procedure that finds edges for this reduction. The algorithm iterates over all edges. For every edge e it takes one adjacent edge e', and lists the adjacent edges of e'. In this list of adjacent edges, for every edge e'' it is tested whether edges e and e'' are non-adjacent and if they have same set of adjacent edges. If that is the case, edges e and e'' are folded together. It is sufficient to only check the adjacent edges of e', because two edges that are folded together have the same neighbourhood. Therefore, e' is the neighbourhood of every edge that is foldable with e. Iterating through all edges is linear in the number of edges. Iterating through the vertices of an edge and through the incident edges of a vertex is linear in the number of vertices of an edge and the number of incident edges of a vertex, respectively. Hence, the algorithm has a worst-case time complexity in $O(m\Delta_E^2\Delta_V^2)$. For $b \not\equiv 1$, an additional check is executed in the algorithm that ensures that the third condition of tuple folding is satisfied. To minimize computation time, we limit the candidates by the number of adjacent edges.

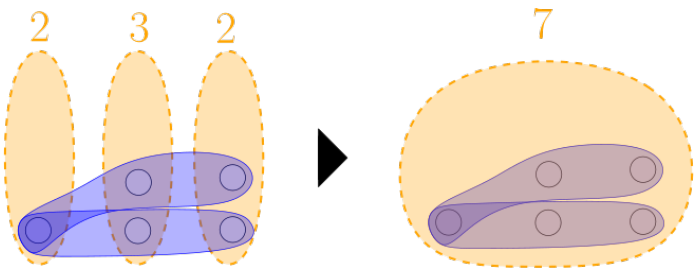


Figure 4.2: Tuple Folding. The orange edges are "folded" together, since they have the same neighbourhood, the blue edges, and are connected to them via vertices of capacity one.

Algorithm 2: Tuple Folding

Data: b-matching M, hypergraph $H = (V, E, w)$
for $e \in E$ **do**
  $N_e = \{f \in E \backslash \{e\} | f \cap e \neq \emptyset\}$;
  e' : an edge adjacent to e;
  $C = \{e'' \in E \backslash \{e\} | e'' \cap e' \neq \emptyset\}$;
  $B = \emptyset$;
  for $e'' \in C$ **do**
    if $e'' \cap e = \emptyset$ **then**
      $N' = \{f \in E \backslash \{e''\} | f \cap e'' \neq \emptyset\}$;
      if $N_e = N'$ **then**
        $B \leftarrow B \cup \{e''\}$
  $E \leftarrow E \backslash B$;
  $w(e) \leftarrow w(e) + w(B)$

4.4 General Tuple Folding

The following reduction is a generalization of tuple folding. In this case, we do not require that N is independent. This reduction can be classified as a "theoretical" reduction since it requires to calculate an optimal b-matching.

Reduction 4 (General Tuple Folding)**.** *For a hypergraph $H = (V, E, w)$ and a function $b : V \to \mathbb{N}_{\geq 1}$, let N be a set of edges such that*

1. *for every pair f, g in N it holds $N(f) \backslash (\{f\} \cup N) = N(g) \backslash (\{g\} \cup N)$,*
2. *for every $e \in N$, and every edge in e' in $N(e) \backslash N$, there is a vertex in $e \cap e'$ that has capacity 1.*

Let M_N be an optimal b-matching in the hypergraph induced by N. Let H' be a hypergraph induced on $E \backslash N \cup e'$ with $e' = \bigcup_{f \in N} f$ and $w_{H'}(e') = w(M_N)$. Let $M_{H'}$ be an optimal

b-matching in H'. If $e' \in M_{H'}$ then $M = M_{H'} \backslash \{e'\} \cup M_N$ is an optimal b-matching in H. If $e' \notin M_{H'}$ then $M = M_{H'}$ is an optimal b-matching in H.

Proof. First, we show that M is a b-matching in H. The starting condition is that $M_{H'}$ is a b-matching in H'. By definition of a b-matching, every vertex $v \in V(H')$ has less than or equal to $b(v)$ matched incident edges. The vertices in $V(H) \backslash e'$ have the same incident edges as the vertices in $V(H') \backslash e'$. Therefore, for every vertex v in $V \backslash \{e'\}$ it holds that $|M \cap E_H(v)| = |M_{H'} \cap E_{H'}(v)| \leq b(v)$. It remains to consider the vertices in e'. If $e' \notin M_{H'}$, then by construction $N \cap M = \emptyset$. There is no additional matched edge in the incident edges of the vertices in e'. Hence, for all $v \in e', |M \cap E_H(v)| = |M_H \cap E_{H'}(v)| \leq b(v)$. Let S be $\bigcup_{e \in N} \bigcup_{e' \in N(e) \backslash N} e \cap e'$. If $e' \in M_{H'}$, then for every $v \in S$ there is one matched edge less and one additional matched edge in $E_H(v)$ compared to $E_{H'}(v)$. Thus, for all $v \in S, |M \cap E_H(v)| = |M_{H'} \cap E_{H'}(v)| \leq b(v)$. For every vertex $v \in e' \backslash S$, the only incident matched edges in $E_H(v)$ are in M_N. Thus, for all $v \in e' \backslash S, |M \cap E_H(v)| \leq b(v)$. Consequently, M is a matching in H.
We show the optimality of M by contradiction. Suppose that there is an optimal b-matching M' such that $w(M') > w(M)$. Let e be an edge in N. If some edge in $N(e) \backslash N$ is matched, then all edges in N are unmatched. Hence, M' is a b-matching in H' which is a contradiction to the optimality of $M_{H'}$. If some edge in N is matched, then all edges in $N(e) \backslash N$ are unmatched. The weight of $N \cap M'$ is the weight of an optimal b-matching in the hypergraph induced by N. Otherwise M' is not optimal. Therefore, $M'' = (M' \backslash N) \cup e'$ is a b-matching in H' and $w(M'') = w(M') - w(N \cap M) + w(e') = w(M')$. This means that M'' has a strictly greater weight than $M_{H'}$ which is a contradicts the optimality of $M_{H'}$. □

4.5 Subset Vertices

Reduction 5 (Subset Vertices)**.** *For a hypergraph $H = (V, E)$ and a function $b : V \to \mathbb{N}_{\geq 1}$, let u, v be two vertices in V such that $E(u) \subseteq E(v)$ and $b(u) \geq b(v)$. If M is an optimal b-matching in the hypergraph H' induced on $V \backslash \{u\}$, then M is an optimal b-matching in H.*

Proof. First we show that a b-matching in H' is a b-matching in H. Let M_e be a b-matching in H'. By definition, for every vertex w in $V \backslash \{u\}$, the number of matched edges in $E_{H'}(w)$ is smaller than or equal to $b(w)$. In particular, this also holds for vertex v. For every vertex w in $V \backslash \{u\}$, $E_H(w) = E_{H'}(w)$ and since $E_H(u) \subseteq E_H(v)$, it follows that $|M_e \cap E_H(u)| \leq |M_e \cap E_H(v)| = |M_e \cap E_{H'}(v)| \leq b(v) \leq b(u)$. Consequently, M_e is a b-matching in H. To show the optimality, suppose there is a b-matching M' in H such that $w(M') > w(M)$. For every vertex w in V it holds that $|M' \cap E(w)| \leq b(w)$. Thus, this also holds for $V \backslash \{u\}$. Therefore, M' is a b-matching in H' which contradicts the optimality of M. □

Algorithm 3 specifies the details on the procedure that finds vertices for this reduction. Briefly summarized, it iterates over all vertices and tries to find candidates in the edges that incident to it. Iterating through all vertices is linear in the number of vertices. Iterating

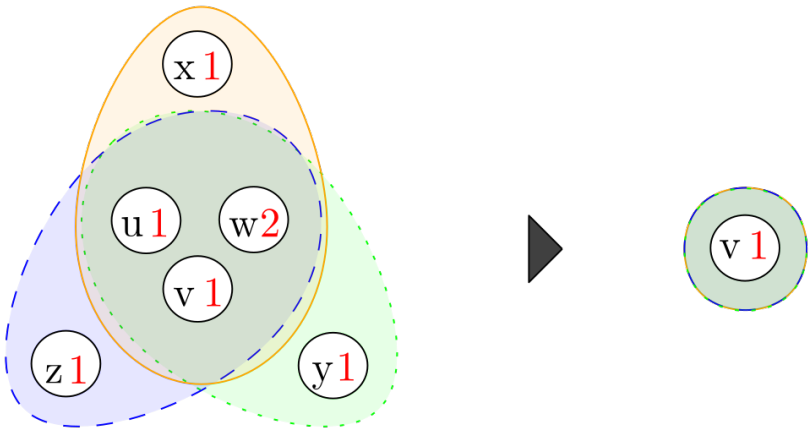


Figure 4.3: Subset Vertices. The white circles represent vertices, the number stands for the b-value of the vertex. According to the reduction, vertex u, w, x, y, z can be removed from the hypergraph, which leaves the three edges and vertex v.

Algorithm 3: Find Subset Vertices

```
Data: edge e
for v ∈ V do
    C = {v' ∈ V | ∃e ∈ E(v) : v' ∈ e} \ {v};
    for v' ∈ C do
        if |E(v')| ≤ |E(v)| ∧ b(v') ≥ b(v) then
            if E(v') ⊆ E(v) then
                V ← V \ {v'};
```

over all incident edges of a vertex and all vertices of an edge is linear in Δ_V and Δ_E, respectively. Since the data structure stores edges of a vertex sorted, checking if an edge set is a subset of another edge set is linear in number of incident edges. Therefore, the algorithm has a total worst-case time complexity in $O(n\Delta_E\Delta_V^2)$. It requires additional memory in $O(\Delta_V\Delta_E)$. Figure 4.3 visualizes this reduction.

4.6 Subset Duo Vertices

Reduction 6. *For a hypergraph $H = (V, E)$ and a function $b : V \to \mathbb{N}_{\geq 1}$, let u, v, w be three distinct vertices in V such that $E(u) \subseteq E(v) \cup E(w)$ and $b(u) \geq b(v) + b(w)$. If M is an optimal b-matching in the hypergraph H' induced on $V \setminus \{u\}$, then M is an optimal b-matching in H.*

Proof. First we show that a matching in H' is a matching in H. Let M_e be a matching in H'. Hence, for every vertex t in $V \setminus \{u\}$, it holds that $|M_e \cap E_{H'}(t)| \leq b(t)$. In particular,

this also holds for v, w. Therefore, $|E(v) \cap M_e| + |E(w) \cap M_e| \geq |(E(v) \cup E(w)) \cap M_e| \geq |E(u) \cap M_e|$. Hence, $|E(u) \cap M_e| \leq b(v) + b(w) \leq b(u)$. Therefore, M_e is a b-matching in H. To show the optimality, suppose there is a matching M' such that $w(M') > w(M)$. For every vertex t in V it holds that $|M' \cap E(t)| \leq b(t)$. Thus, this also holds for $V \backslash \{u\}$. Therefore, M' is also a b-matching in H' which contradicts the optimality of M.

□

Remark. This reduction is only feasible for a b-function that maps some vertices to a value larger than 1.

4.7 General Weighted Domination

The following reduction is a generalization of "Weighted Domination"[30].

Reduction 7. *For a hypergraph $H = (V, E, w)$ and a function $b : V \to \mathbb{N}_{\geq 1}$, let N be a set of edges and e be an edge in $E \backslash N$ such that*

1. *N is a matching in H,*
2. *$w(N) \geq w(e)$*
3. *$\forall e' \in N, e' \subseteq e$ and there is a vertex with capacity 1 in $e \cap e'$,*

then e can be removed from the hypergraph.

Proof. For a b-matching M, it holds that, if e is in M, then $M \cap N = \emptyset$. This holds because for every edge e' in N, there is a vertex v that is in e and in e' and has capacity 1. Therefore, if e is in M then e' is not in M. Now we proof that for any b-matching M such that $e \in M$, $(M \backslash \{e\}) \cup N$ is a b-matching in H. Let M be a b-matching such that e is in M. For the b-matching $M \backslash \{e\}$, every vertex $v \in e$ has $b(v) - |M \cap E(v)| > 0$. Therefore, for every vertex in e one incident edge can be added. Since N is a matching, there is no vertex in more than one edge for the edges in N. Every edge in N is a subset of e. Hence, $(M \backslash \{e\}) \cup N$ is a b-matching in H. Now, we will prove that we can remove e, which means there is an optimal solution M in H such that e is not in M. In case $w(N) > w(e)$, then there is no optimal solution M such that $e \in M$. Assume there is an optimal b-matching M such that $e \in M$. All edges in N are unmatched. Therefore, $(M \backslash \{e\}) \cup N$ is a b-matching with a strictly larger weight which contradicts the optimality. In case $w(N) = w(e)$, let M be an optimal b-matching in H such that $e \in M$, then $(M \backslash \{e\}) \cup N$ is an optimal b-matching with the same weight. Both cases show that there is an optimal b-matching M such that e is not in M. □

Remark. For $|N| = 1$, the same conditions as in the weighted domination reduction are satisfied. An algorithm that finds all instances of this reduction requires solving the decision version of hypergraph matching which is a NP-complete problem: For an edge e, given

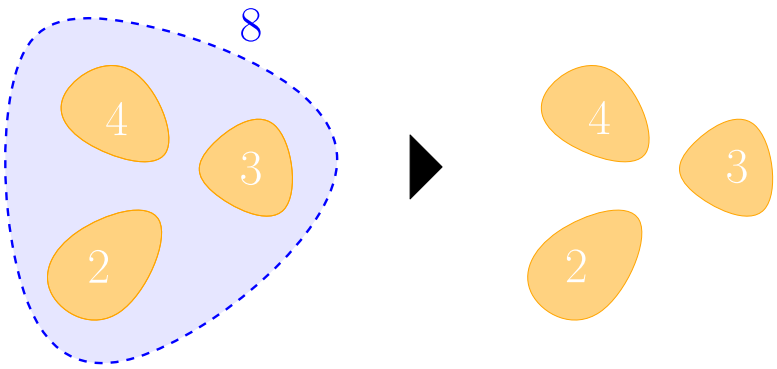


Figure 4.4: General Weighted Domination. The numbers are the edge weights. If two edges overlap, then they share a capacity one vertex. The orange edges are a matching that has a higher weight than the blue edge. According to the reduction, the blue edge can be removed.

all edges (execpt for e) that are subsets of e and share a vertex with e that has capacity one, the reduction requires to know if there is a matching in the hypergraph induced by these edges that has at least weight $w(e)$. Figure 4.4 visualizes an example of general weighted domination.

Algorithm. We show an improvement in worst-case runtime complexity for the not-hashed algorithm [30] that finds edges for "Weighted Domination". Currently, it has a worst-case time complexity in $O(m(\Delta_E \Delta_V \log \Delta_V))$. At a certain point in the algorithm, a set of candidates has been identified and it is required to know if the candidates are supersets of the current edge e_{sub}. Let e_{super} be one of the candidates. Determining if e_{sub} is a subset of e_{super} is done the following way: For every vertex v in e_{sub}, do a binary search in $E(v)$ to find e_{super}. If for all vertices in e_{sub}, e_{super} can be found in its incident edge list, then e_{sub} is a subset of e_{super}. This has worst-case time complexity in $O(\Delta_E \log \Delta_V)$. This works because the edges in every incident edge list are sorted. However, the data structure also sorts the list of vertices in an edge which is why Algorithm 4 works. In this algorithm, the function $vertex(i, e)$ returns the ith vertex of edge e. This algorithm runs linear in the number of vertices of e_{super} and leads to an overall algorithm for "Weighted Domination" that has a worst-case time complexity in $O(m(\Delta_E \Delta_V))$.

Algorithm 4: Check Subsets

```
Data: b-matching M, edge e_sub, edge e_super
i, j = 1;
while i ≤ |e_super| do
    if vertex(i, e_super) = vertex(j, e_sub) then
        j ← j + 1;
        if j > |e_sub| then
            return True;
    i ← i + 1;
return False;
```

4.8 Parallelization

We parallelize the newly presented reduction, Subset Vertices, Isolated Weight Transfer and Tuple Folding, as well as the reductions by Großmann et al. [30], excluding Weighted Isolated Edge Removal and Weighted Twin (since they are special cases of the new reductions as mentioned earlier). For all reductions, the chosen parallelization technique is to partition the edges or vertices into p blocks and let each processing element check for each edge or vertex in its block if a reduction is applicable. Edges that are applicable for a reduction are gathered simultaneously. After collecting the edges, applying the reductions is executed sequentially. For reductions that determine whether an edge necessarily is the optimal b-matching or not, adding and removing cannot be executed simultaneously for two reason. First, the data structure does not allow it. Secondly, a reduction applied to one edge can affect the conditions of another edge (mostly adjacent edges). For example, in the case of $b \equiv 1$ two unmatched adjacent edges can be identified to execute a neighborhood removal but not both can be added to the b-matching. This means that after gathering the edges that can perform a reduction an additional check is necessary. This is the case for all reductions except for abundant vertices. For subset vertices, two vertices can be identified to be removed but can not be removed both if they have the same incident edges and the same b-value. To avoid collision in this case, the vertex with the smaller id is chosen.

Load Balancing. One technique that turned out to be helpful was load balancing. Instead of partitioning the edges or vertices into p equal sized blocks at the beginning, every processing element grabs a small number of edges/vertices from a list of unchecked edges/vertices and checks whether the reductions are possible for this batch of edges/vertices. When the processing element has finished checking its batch, it grabs another one as long as the list of unchecked edges/vertices is non-empty. This avoids situations where one processing element might be unlucky to get a lot of edges/vertices that are time consuming to check, which might lead to one processing element finishing later. This makes a

difference for reductions in which reviewing one edge depends on edge properties such as the number of adjacent edges. This can be observed in reductions such as neighbourhood removal or isolated weight transfer. However, not for every reduction the time for checking an edge/vertex depends on edge/vertex properties. In the "abundant vertices" reduction, it does not make a difference to dynamically assign vertices since it is in constant time to check whether a vertex has less incident edges than the residual capacity (assuming that the data structure stores the size of the list of incident edges, and the lists of incident edges do not store duplicate edges).

CHAPTER 5

Heuristic Approaches

In this chapter, we summarize the results for the heuristic approaches for hypergraph b-matching. We introduce the D-Core algorithm for hypergraph b-matching, an algorithm inspired by the D-Core for the independent set problem [29]. This includes an initialization, perturbation and parallelization techniques. Furthermore, we propose two improvements for the local search procedure.

5.1 D-Core Algorithm

The D-Core heuristic restricts the search space of a local search algorithm. Given multiple solutions, a subgraph is induced on edges where the solutions do not agree whether the edges are in the b-matching or not. Its intuitive idea is to assume that it is worth spending more time on this subgraph rather than on the subgraph where the solutions agree. This can be compared to an empirical induction argument (not to be confused with mathematical induction).

Definition 1 (D-Core Set)**.** *For a hypergraph $H = (V, E)$ and b-matchings $M_1, M_2, ..., M_l$ in H, the Difference Core Set, short D-Core Set, of $M_1, M_2, ..., M_l$ is $\{e \in E \mid \exists\, i, j \in \{1, ..., l\}\, e \in M_i \wedge e \notin M_j\}$.*

Definition 2 (D-Core)**.** *For a hypergraph $H = (V, E)$ and b-matchings $M_1, M_2, ..., M_l$ in H, a D-Core is the hypergraph induced on the D-Core set of $M_1, M_2, ..., M_l$.*

Figure 5.1 visualizes an example for a D-Core computed by 3 solutions.

Algorithm D-Core. Given a local search technique, the general idea of the D-Core algorithm consists of two steps. First, the algorithm computes multiple solutions. We refer to these solutions as base solutions. In the second step, from the base solutions, compute the D-Core, and execute the local search on the D-Core. Since it is not necessarily true that

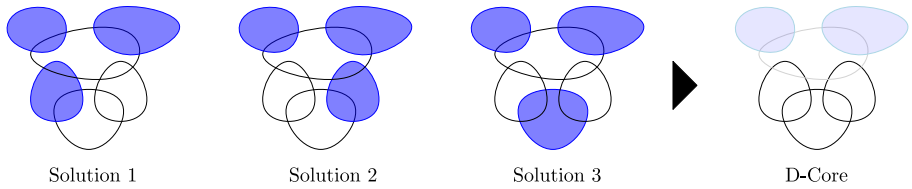


Figure 5.1: D-Core Example. Filled edges are in the matching.

the solution is correct in the hypergraph induced by the complement of the D-Core set, at some point we return back to continue the local search on the entire hypergraph. This step is repeated until some stopping criterion is met. In our case, the stopping criterion is a time limit, indicated by t_{run}. For this algorithm, we propose the following extensions. After the D-Core is computed, the reductions for hypergraph b-matching are applied to it. If at some iteration step the D-Core set is empty, then half of the base solutions will be perturbed. If the size of the D-Core set is smaller than some constant, then instead of running local search on the D-Core, the exact solution of the b-matching in the D-Core is computed by an optimization tool. Algorithm 5 specifies the D-Core procedure. As a starting point for the local search procedure in the D-Core, the matching of the base solution is transferred by selecting the corresponding edges. As mentioned in Section 3, the local search procedure has some randomized elements, which is why the base solutions differ after the local search and the D-Core is not zero (even if all base solution are initialized the same). To ensure that the random values are not calculated the same, we set for each base solution a distinct randomization seed. We denote t_{entire} the time that the local search is executed for each base solution on the entire hypergraph and we denote t_{DCORE} the time that the local search is executed for each base solution on the D-Core. If at some point in the execution, the additional time t_{entire} or t_{DCORE} surpasses the total time of the run, t_{entire} or t_{DCORE} are adjusted.

Computing D-Core set. The standard algorithm to calculate the D-Core set is: for each edge in the hypergraph iterate through all base solutions. If there are two b-matchings that do not have the same result on whether the edge is in the matching, then add the edge to the D-Core set. For l base solutions, this has a worst-case time complexity in $O(lm)$. In addition, we propose a parallel algorithm. Let p be the number of processing elements. Partition the edge array into p equal sized blocks. Every processing element is assigned to one distinct block and gathers for its block the edges that are in the D-Core set. Afterwards, the processing elements write in parallel their local D-Core set into one result array. Each block has $\frac{m}{p}$ elements. Determining the D-Core set on a block has a worst case time complexity in $O(\frac{m}{p}l)$ because we need iterate over l b-matchings for each element in a block. Communicating where to write the local array into the global array has worst case time complexity in $O(p)$. Writing the local array into the global array has a worst-case time complexity in $O(\frac{m}{p})$. This leads to an algorithm that has a worst-case time complexity in

Algorithm 5: DCORE algorithm

```
Data: a = [b-matching] × l
stop ← false;
while ¬stop do
    for M_b ∈ a do
        M_b ← LocalSearch(M_b);
    H_D ← compute DCORE using a;
    if |E(G)| = 0 then
        for M_b ∈ {M_1, .., M_{l/2}} do
            M_b ← Pertubation(M_b);
    H_D ← apply reductions on H_D;
    if |E(G)| ≤ c_ILP then
        M_DCORE ← ILP(H_D);
        Insert M_DCORE into all M_b in a;
    else
        for M_b ∈ a do
            M_DCORE ← LocalSearch(M_b, H_D);
            Insert M_DCORE into M_b;
    stop ← evaluateTimePassed();
```

$O(\frac{m}{p}l + p)$. In case that p is equal to the number of base solutions, then the worst-case time complexity is in $O(m + l)$.

Inserting D-Core Matching into the Base Solution. The procedure for inserting a b-matching in the D-Core into a base solution in the entire hypergraph is to remove all edges from the base solution that are in D-Core set, and then add the edges in the D-Core matching to the base solution. To ensure that the base solution is correct after inserting the D-Core matching we do the following: When creating the D-Core based on the D-Core set, we adjust the b-function for the vertices that have matched incident edges that are not in the D-Core set. Let M' be the matching in the hypergraph induced on the complement of the D-Core set. For all vertices, it should hold that $|(M' \cup M_{DCORE}) \cap E(v)| \leq b(v)$. Since M' and M_{DCORE} are disjoint, it follows that $|M_{DCORE} \cap E(v)| \leq b(v) - |M' \cap E(v)|$ which means that that the b-value for a vertex v in the D-Core is $b(v) - |M' \cap E(v)|$.

Perturbation. In case the D-Core has size zero, which means that all b-matchings are the same, half of the base solutions are perturbed by the following procedure. For a base solution with index i, in ascending order of the $(\frac{1}{4} + \frac{1i}{4l})m^{th}$ heaviest edges, force the

Algorithm 6: Initialization

```
Data: b-matching M
while M is not maximal do
    e ← random edge in RCL;
    M ← M ∪ {e}
```

edges into the solution. The ascending order ensures that large edges are more likely to be in the solution.

Initialization. The b-matchings are initialized via a randomized greedy algorithm. The edges of the hypergraph are evaluated according to a cost function $c : E \to \mathbb{R}$. While there is some free edge in the hypergraph, the algorithm calculates a restricted candidate list (RCL). It chooses a random edge in the RCL, and adds it to the matching. Algorithm 6 specifies this procedure. Let α be a real number in $[0, 1]$. Let c_{min} be the minimum value over all free edges and c_{max} be the maximum value over all free edges. The restricted candidate list RCL is defined as a subset of the free edges in the hypergraph such that for every edge e in RCL, it holds that $c_{min} \leq c(e) \leq c_{min} + \alpha(c_{max} - c_{min})$. Every base solution is initialized this way. Note that in case the cost function is actually a "gain" function, meaning high values are preferred, the formula for the RCL is adjusted to $c_{max} \geq c(e) \geq c_{max} - \alpha(c_{max} - c_{min})$.

Exact D-Core Solution. In case that the size of the D-Core set is smaller than a constant c_{ILP}, we will make use of an ILP solver to determine the exact solution in the D-Core. We make use of the ILP formulation of weighted hypergraph b-matching by Großmann et al. [30]. The runtime for the ILP solver is limited by t_{DCORE}. If the ILP solver does not find the optimal solution, we extract the variable values and calculate a b-matching based on it. We iterate in order of the edge ids. If the corresponding variable for an edge in ILP formulation has a value larger than or equal to 0.5 and the edge is free, then we add the edge to the b-matching. Afterwards, we check if this calculate b-matching has a strictly larger weight than the current base solution intersected with the D-Core set. If that is the case, we write the D-Core solution into the base solution.

Parallelization. The D-Core algorithm offers a simple parallelization. For p base solutions, the local search is calculated simultaneously on p processing elements, as well as the p execution of the local search on the D-Core. Additionally, we calculate concurrently the initialization, the determination of the D-Core set, the reductions on the hypergraph induced on the D-Core, the perturbation in case the D-Core size is zero and solving the ILP in case the D-Core size is smaller than some constant.

5.2 Local Search

Local Search is a technique that explores the solution space by visiting neighboring solutions. It is a process that consists of "improving a sub-optimal solution" and "overcoming local optima". To improve the solution, the $(1, 2)$-Swap algorithm has shown to be a reliable technique. We extend this with a $(1, 1)$-Swap technique which can improve the solution or deviate the solution without improvement in order to find new neighboring solutions in the solution space. In addition, we present a copying method for b-matchings.

(1,1)-Swap. A $(1, 1)$-Swap is an algorithm that removes an edge from a b-matching and adds an adjacent edge to the b-matching that became free due to the removal. The $(1, 1)$-Swap algorithm is integrated as follows. During the search for $(1, 2)$-Swaps, if for a matched edge e_m a $(1, 2)$-Swap is not executable, the $(1, 1)$-Swap is checked for the same edge. It reuses the list of blocked edges that only have vertices at full capacity in e_m. A $(1, 1)$-Swap is feasible if $w(e_m) \leq w(e)$ for some edge e whose blocking vertices are all in e_m. We integrate this algorithm into the candidate version of the local search, which selects possible edges for future $(1, 1)$-Swaps and $(1, 2)$-Swaps.

Copying b-Matchings. The following algorithm works in a setting with two objects of the b-matching data structure M_1 and M_2, M_2 is an improved version of M_1, both matchings are maximal, and the goal is to copy the results of M_2 into M_1. We see such a setting, for example, at the end of the local search procedure. The current algorithm for this procedure is to reset M_1. Then iterate over the matched edges in M_2 and add them to M_1. This has a worst case-time complexity in $O(m\Delta_V\Delta_E)$. This works well in practice, because in the b-matching data structure, the matched edges are stored in a consecutive array. This means, it is usually not necessary to iterate over the entire edge set to get the matched edges. However, it is likely that most edges did not change after the local search procedure, which means that the intersection of the matched edges in M_1 and M_2 is large. And resetting M_1 will remove edges that will be added back, which will execute unnecessary operations. Additionally, in the data structure by Großmann et al. [30], the time complexity of removing and adding edges from and to the b-matching is in $O(\Delta_E\Delta_V)$. Thus, when copying a b-matching it makes sense to reduce the number of removals and additions. Therefore, we propose Algorithm 7. The worst-case time complexity is in the same complexity class as the first algorithm, in $O(m\Delta_V\Delta_E)$.

Algorithm 7: Copying b-matching

```
Data: b-matching M_1, b-matching M_2
for e ∈ M_2 do
    if e ∉ M_1 then
        for v ∈ e do
            if b(v) = |M_1 ∩ E(v)| then
                for e' ∈ E(v) do
                    if e' ∈ M_1 ∧ e' ∉ M_2 then
                        M_1 ← M_1\{e'};
                        break;
        M_1 ← M_1 ∪ {e};
```

CHAPTER 6

Experimental Evaluation

In this chapter, we present the experimental results of the thesis. This includes specifying which hardware the experiments were run on in Section 6.1, what kind of instances were used in Section 6.2, which methodology was applied in Section 6.3, and a presentation of the resulted data in Section 6.4.

6.1 Hardware

For the experiments, we make use of two machines. At the beginning of each paragraph it specified which machine is used for the experiments in the paragraph. The machines are identified by A and B:

A The first machine contains an AMD EPYC 7702P 64-Core Processor. It has a total of 1 TB RAM. 3 Cache levels: L3: 256 MiB, L2: 32 MiB, L1(i+d): (2+2) MiB. The processor has a nominal base clock speed of 1.5 GHZ with a maximum boost frequency of up to 2 GHZ and the machine runs on Ubuntu 20.04.3 LTS.

B The second machine contains an Intel(R) Xeon(R) Silver 4216 CPU. It has a total of 96 GB RAM. It has 16 cores and 3 cache levels: L3: 22 MiB, 16 MiB, (512 + 512) KiB. The processor has a nominal base clock speed of 0.8 GHZ with a maximum boost frequency of up to 3.2 GHZ and the machine runs on Ubuntu 24.04.3 LTS.

6.2 Instances

The algorithms are tested on two collections of hypergraphs, namely M_{HG} and L_{HG}. Both sets are provided by Gottesbüren et al. [28]. M_{HG} and L_{HG} consist of 488 medium-sized hypergraphs and 96 large-sized hypergraphs, respectively. The hypergraphs are retrieved from 4 sources covering 3 applications: the ISPD98 VLSI Circuit Benchmark Suite [2],

Benchmark Set	Range #Edges	Av./Median #Edges	Av./Median Edge Size
M_{HG}	[163, 13 378 617]	871 796 / 134 756	24.36 / 6.82
L_{HG}	[89 315, 139 353 211]	8 241 500 / 4 382 246	25.32 / 9.94

Benchmark Set	Range #Nodes	Av./Median # Nodes	Av./Median Degree
M_{HG}	[7 502, 13 378 617]	642 155 / 104 244	26.64 / 4.99
L_{HG}	[89 315, 139 353 211]	5 934 130 / 2 380 515	23.02 / 8.25

Table 6.1: Statistics for hypergraphs

the DAC 20212 Routabilty-Driven Placement Contest [55], the SuiteSparse Matrix Collection [12], and the International SAT Competition 2014 [5]. Table 6.1 displays statistics for the benchmark sets.

6.3 Methodology

We analyze the algorithms with respect to runtime and quality. We make use of optimization tool Gurobi [32] version 13 to solve ILP of hypergraph b-matching. The algorithms are parallelized with OneTBB, a thread-based task-oriented parallelism library. All algorithms are implemented in C++17. The project is compiled with gcc and full optimizations (flag -O3) on. All experiments are executed sequentially. We refer to $DCORE_XS$ as the D-Core algorithm executed with X base solutions. We abbreviate Isolated Weight Transfer as IWT, Subset Vertices as SV, and Tuple Folding as TF.

For evaluating the experiments and visualizing the data, we make use of performance profiles as proposed by [13]. A solution quality performance profile plot visualizes for each included algorithm A a function f_A which is defined as follows: Let $\mathcal{A}$ be the algorithms included in the plot. Let $\mathcal{G}$ be the hypergraphs for which each algorithm has computed a solution. The function f_A maps a value $\tau \in [0, 1]$ to the fraction of instances that are at least as good as τ times the best quality calculated by all other algorithms in $\mathcal{A}$. More formally, $f_A(\tau) = \frac{|S^A_{\geq\tau}|}{|\mathcal{G}|}$ with $S^A_{\geq\tau} = \{G \in \mathcal{G} | A_{sol}(G) \geq \tau \max\{B_{sol}(G) | B \in \mathcal{A}\}\}$. In a time performance profile plot, for every algorithm a function f_A is visualized as well. For this type of plot, $f_A(\tau) = \frac{|S^A_{\leq\tau}|}{|\mathcal{G}|}$ with $\tau \in [1, \infty]$ and $S^A_{\leq\tau} = \{G \in \mathcal{G} | A_{time}(G) \leq \tau \min\{B_{time}(G) | B \in \mathcal{A}\}\}$. Heuristically speaking for both types of plots, an algorithm A is better (with respect to solution quality or runtime compared to the other included algorithms) if f_A is close to the constant 1 function.

Parameters. We briefly summarize the standard values of the parameters for the experiments. The standard execution time on one hypergraph instance is 250 seconds for the heuristic algorithms. The standard value in the D-Core algorithm for t_{entire} is 30 seconds and t_{DCORE} is 15 seconds. The standard α value in the initialization of the D-Core algo-

Reduction	Constraint	Default
Isolated Weight Transfer	Edge Size	8
Isolated Weight Transfer	Clique Size	80
Weighted Edge Folding	Adjacent Edges	2
Tuple Folding	Adjacent Edges	80
Weighted Domination	Edge Size	6
Neighbourhood Removal	Edge size	6

Table 6.2: Constraints for the reductions

rithm is 0.1. The cost function in the initialization algorithm is the edge weight divided by the edge size. The standard value for the D-Core size at which the ILP is calculated is 1 000 edges. All experiments are performed with uniformly-random selected integer edge weights in $[1, 100]$. These are calculated once and reused for all experiments. One reduction round executes the algorithms in the following order: Abundant Vertices, Subset Vertices, Neighbourhood Removal, Weighted Domination, Abundant Vertices, Weighted Edge Folding, Abundant Vertices, Isolated Weight Transfer, Abundant Vertices, Tuple Folding. The reductions on the D-Core run for 10 rounds or less if the reductions are not able to decrease the kernel size anymore. Table 6.2 gives an overview for standard values of the reduction parameters. The reductions and the final performance of the D-Core are tested for $b \equiv 1, b \equiv 3, b \equiv rand.$. "$b \equiv rand.$" denotes a b-function that maps a vertex v to a uniform random natural number in $[1, |E(v)|]$. These values were computed once for every hypergraph and then reused for all experiments.

6.4 Experiments

In this section, we present the data of the experiments, beginning with reduction experiments and local search experiments, following up with experiments on parameters for the D-Core, and finishing with a comparison of the D-Core to other algorithms. Each subsection begins with an overview of all corresponding results.

6.4.1 Reductions

Overview. The results for the reduction show that the improvement in runtime complexity of weighted dominations finds all applicable edges for $b \equiv 1$ on average 2.85 faster. The extension for weighted edge folding shrinks the kernel on average by 37.73 edges. The new reductions shrink the average kernel for $b \equiv 1$ by 24.6%; for $b \equiv 3$ by 5.06%; for $b \equiv rand.$ by 20.05%. The runtime of the new reductions is roughly 95% of runtime of all reductions. The parallel version computes on average for $b \equiv 1$ the reduction 32 times faster than the sequential algorithm. The average runtime for solving the ILP of hypergraph

b-matching with reductions compared to without reductions is, for $b \equiv 1$, 2.26 times faster: for $b \equiv 3$ 2.21 times faster; for $b \equiv rand.$ 1.94 times faster.

Weighted Domination Runtime. Using Machine B, we look into the improvement in runtime for the sequential "Weighted Domination" algorithm. In this experiment, we only apply the weighted domination reduction to the hypergraphs and we apply it exhaustively. We test for $b \equiv 1$. The metric that we analyze is the runtime of algorithm. We define the speedup as the the runtime of the old algorithm divided by the runtime of the new algorithm. The general result is that the new algorithm outperforms the old one. The speedup values are in $[0.07, 94.43]$. The arithmetic mean of the speedup values over all hypergraphs is 2.85 and the median value is 1.74. 94.5% of hypergraphs have a speed up value that is strictly larger than 1.0. The largest absolute time that the new algorithm improved over the old one is $440.5s$ and largest absolute time that the old one was faster than the new one is $2.04s$.

Weighted Edge Folding. Using Machine B, we analyze in this experiment the extension of applicable edges for Weighted Edge Folding. For this, we execute exhaustively Weighted Edge Folding once with the new condition and once with the old condition. The relative size is the number of edges of the kernel obtained with the new condition divided by the number of edges of the kernel obtained with the old condition. The relative kernel size over all hypergraphs are in $[0.9974, 1]$. The arithmetic average is 0.99993. The absolute largest difference is 1 360 edges. The absolute arithmetic average is 37.73.

Kernel Size. Using Machine B, we analyze the reductions: Isolated Weight Transfer, Tuple Folding and Subset Vertices with respect to the kernel size. We perform for every listed reduction type a separate experiment that only applies this reduction exhaustively. Additionally, we execute one experiment that applies all reductions and compare it to one experiment that executes the reductions without the 3 listed reductions. We test for following b-functions: $b \equiv 1$, $b \equiv 3$ and $b \equiv rand.$ The metric that we analyze is the size of the kernel, which is measured either by the number of edges or by the number of vertices. The relative kernel size is the kernel size divided by the size of the original hypergraph. Table 6.3 and 6.4 display the kernel size calculated by the number of edges and the number of vertices, respectively. Briefly summarized, for all b-functions, we see that the 3 additional reductions affect the kernel size. For example, for $b \equiv 1$, in combination of all reductions, the kernel size improves from 81.7% to 61.6%, a decrease of 24.6%; for $b \equiv 3$ by 5.06%; for $b \equiv rand.$ by 20.05%. For all tested b-functions, the new reductions also lead to an increase in the number of fully solved hypergraphs and a decrease in the number of hypergraphs that are unaffected by the reductions. For the run with the old reductions (without the three new reductions), 26 hypergraphs (9 in M_{HG} and 19 in L_{HG}) failed due to timeout, and in the run with the three new reductions 35 hypergraphs (11 in M_{HG} and 24 in L_{HG}) failed due to timeout. Every hypergraph run was limited by 1 000 seconds and was executed in parallel with 16 processing elements.

Reduction Type	$b(v)$	Range	Arithmetic Average	Median Value	Fully Solved	Unaffected
Isolated Weight Transfer	1	[0.00,1.00]	0.885	0.997	4	111
Tuple Folding	1	[0.103, 1.00]	0.996	1	0	469
All Reductions without TF, IWT, and SV	1	[0,1]	0.817	0.914	2	80
	3	[0,1]	0.710	0.898	34	99
	R	[0.009, 1]	0.668	0.649	0	57
All Reductions	1	[0,1]	0.616	0.748	17	32
	3	[0,1]	0.674	0.885	36	73
	R	[0,1]	0.534	0.511	2	16

Table 6.3: This table displays the results of the relative kernel size calculated by the number of edges. "Fully Solved" and "Unaffected" refers to the number of fully solved hypergraph and hypergraphs that are unaffected by the reductions, respectively.

Reduction Type	$b(v)$	Range	Arithmetic Average	Median Value	Fully Solved	Unaffected
Subset Vertices	1	[0.016,1]	0.731	0.939	0	90
All Reductions without TF, IWT and SV	1	[0,1]	0.853	0.960	2	96
	3	[0,1]	0.604	0.656	34	108
	R	[0.002, 0.996]	0.533	0.499	0	0
All Reductions	1	[0,1]	0.526	0.489	17	30
	3	[0,1]	0.464	0.340	36	24
	R	[0,0.996]	0.369	0.335	2	0

Table 6.4: This table displays the results of the relative kernel size calculated by the number of vertices. "Fully Solved" and "Unaffected" refers to the number of fully solved hypergraph and hypergraphs that are unaffected by the reductions, respectively.

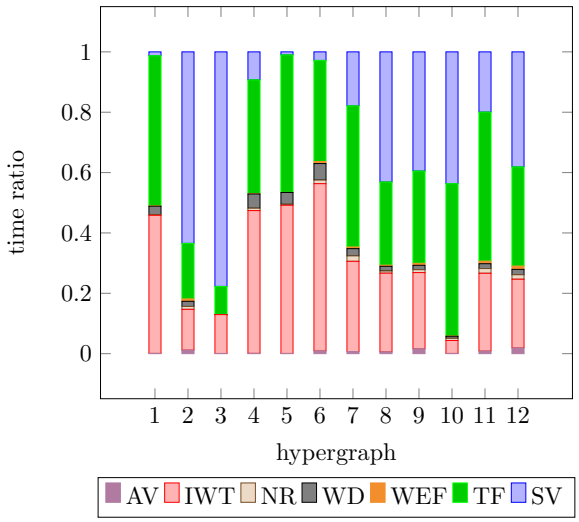


Figure 6.1: The relative runtime for the seven reductions types run for ten rounds on twelve hypergraphs.

Runtime. Using Machine B, we analyze the runtime of Subset Vertices, Tuple Folding and Isolated Weight Transfer and compare it to the other reductions. It is run for $b \equiv 1$. For this experiment, we sequentially perform ten reduction rounds. For every reduction, its runtime in each round is added together. For every reduction, we compute the relative runtime by the runtime during the ten rounds divided by the runtime of all reductions. For each reduction, the arithmetic average of the relative runtime over all hypergraphs in ascending order is: Weighted Edge Folding 0.61%, Abundant Vertices 0.75%, Neighbourhood Removal 1.15%, Weighted Domination 2.22%, Isolated Weight Transfer 25.73%, Tuple Folding 34.16%, Subset Vertices 35.59%. Figure 6.1 visualizes the relative runtime of twelve randomly selected hypergraphs.

Parallelization. Using Machine A, we analyze the parallelized version of the reduction algorithms. The metric that we look at is the runtime of the algorithms. The speedup is defined as the time of the sequential version divided by the time of the parallel version. The parallel algorithm is executed with 64 threads. The sequential version applies 10 reduction rounds to a hypergraph and terminates. The parallel version applies as many rounds as necessary until the number of edges in the kernel is smaller than the number of edges in kernel computed by the sequential version. In general, the parallel algorithm outperforms the sequential one. The speed values are in $[0.053, 464.68]$. The arithmetic average is 32.77. The median value is 23.73. Two hypergraphs have a speedup smaller than 1.0. 90% of the speedup values are in $[6.56, 63.56]$.

Solving ILPs with Reductions. Using Machine B, we compare the runtime for solving ILPs of hypergraph b-matching with respect to reductions. The first algorithm solves

the problem with an ILP and Gurobi as an optimization tool. The second algorithm solves the problem by first applying reductions and then solving the ILP of the kernel with Gurobi. The runtime includes the reduction time and time for the ILP solver. Both algorithms are assigned 16 cores to solve the problem in 500 seconds. Only hypergraphs are considered for which an exact solution is returned. For $b \equiv 1$, Figure 6.2a shows the time performance profile plot. The speedup values are in $[0.34, 18.8]$. The arithmetic average is 2.26, and the median value is 1.35. For $b \equiv 3$, Figure 6.2b shows the time performance profile plot. The speedup values are in $[0.029, 34.83]$. The arithmetic average is 2.21, and the median value is at 1.14. For $b \equiv rand.$, Figure 6.2c shows the time performance profile plot. The speed up values are in $[0.014, 13.27]$. The arithmetic average is 1.94 and the median value is 1.33.

6.4.2 Local Search

Overview. The $(1, 1)$-Swaps improve the results for $b \equiv 1$ on average by 0.57%. The copying algorithm speeds up the average runtime by a factor of 3050.18.

(1,1)-Swaps. Using Machine A, we examine how the $(1, 1)$-Swaps affect the quality of the results computed by the local search procedure. For this, we execute the local search for 250 seconds and $b \equiv 1$ once with the $(1, 1)$-Swaps and once without the $(1, 1)$-Swaps. Figure 6.3 shows the quality performance profile for the 2 algorithms. The relative improvement values are in $[-1.05\%, 7.61\%]$. The arithmetic average over the relative improvement values is 0.57% and the median value is 0.245%.

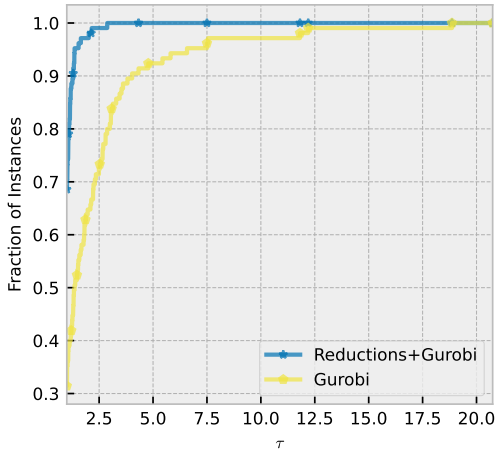
Copying Algorithm. Using Machine A, we analyze the copying method with respect to runtime. For an equal comparison, we execute the experiments as follows. For each hypergraph, we execute the local search procedure for 50 seconds. After the local search has finished, the best result is copied into the target object once with the new copying method and once with the old copying method. This ensures that we copy the same number of edges. The speedup values are in $[9.82, 200\,403]$. The arithmetic average is $3\,050.18$. The median value is 480.68.

6.4.3 D-Core Algorithm

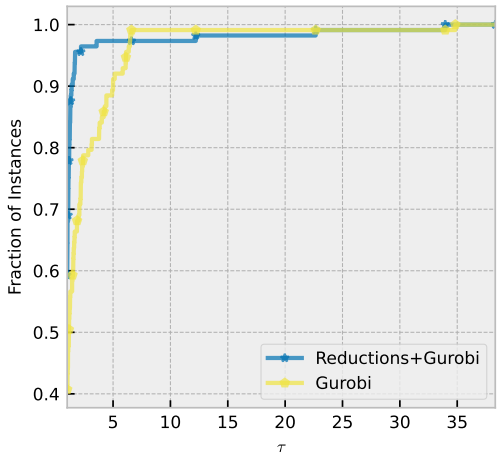
We will begin this subsection by evaluating parameters of D-Core algorithm. Our strategy is to evaluate one parameter for different domain values and keep all other parameters constant. Afterwards, we analyze the impact of the D-Core reductions on the quality and runtime. Finally, we compare the D-Core algorithm with other contestants.

Overview. For $b \equiv 1$, the D-Core algorithm performs best with 32 base solutions and an initialization value $alpha = 0.5$. The reductions on the D-Core improve the results for $b \equiv 1$ in runtime by an average speedup of 3.80 and an average improvement in solution

(a) Capacity 1



(b) Capacity 3



(c) Random Capacity

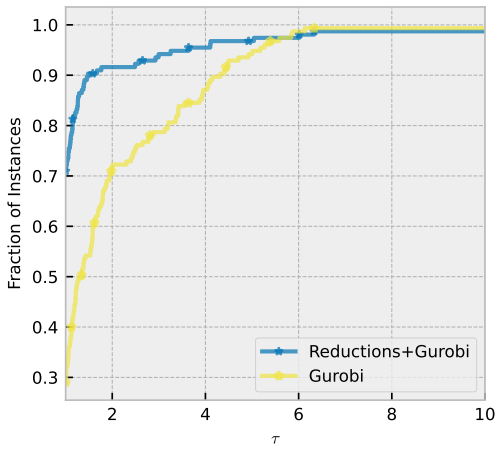


Figure 6.2: Time performance profile plots for ILP optimization combined with reductions compared to solely ILP optimization. Both run on 16 processing elements for 500 seconds and 3 different capacities.

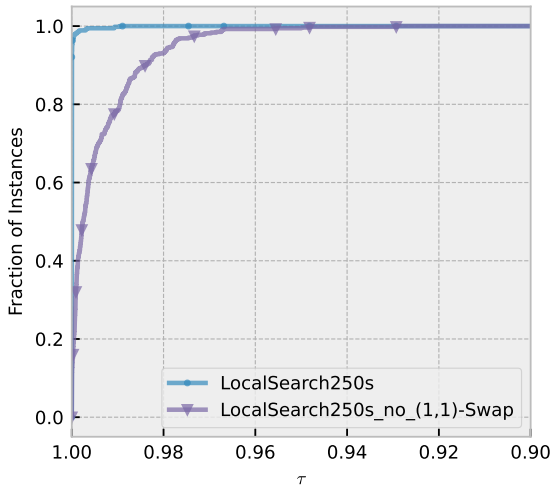


Figure 6.3: Quality performance profile for the local search run once with $(1, 1)$-Swaps and once without $(1, 1)$-Swaps for $b \equiv 1$ and 250s.

quality of 0.17%. The parallel algorithm for the D-Core set calculation computes the D-Core set with 64 processing elements on average 14.74 faster compared to the sequential algorithm. For $b \equiv 1$, the average relative size in the number of edges of the D-Core and the reduced D-Core compared to the original hypergraph is 11.93% and 5.79%, respectively. The D-Core algorithm improves on average a local search algorithm that was adjusted to use the same computational resources for $b \equiv 1$ by 0.5%; $b \equiv 3$ by 0.4%; $b \equiv rand.$ by 0.4%; For $b \equiv 1$, Gurobi finds the best solution in roughly 77% of the instances, for the rest of the instances it barely finds a feasible solution. For $b \equiv 3$ and $b \equiv rand.$, the proportion of best solution computed by Gurobi increases to 82% and 98%, respectively. The D-Core algorithm, the local search and Gurobi fail on $5 - 7\%$ of the tested hypergraphs for all b-functions.

Parameter: Number Of Base Solutions. Using Machine A, we analyze how the number of base solutions affects the quality of the result. For $b \equiv 1$, we run the D-Core algorithm for 4, 8, 16, 32 and 64 base solutions. Figure 6.4a shows the performance profile plot. We observe that the quality of the solutions is best at 32 base solutions and worst at 4 base solutions. The run with 32 base solutions improves the solution quality compared to the run with 4 base solutions up to 2.63% and on average 0.17%. In general, the solution quality scales with the number of base solutions but is not heavily affected by it.

Parameter: Alpha in Initialization Using Machine B, we analyze the D-Core with respect to different alpha values in the initialization step. We test for $\alpha \in \{0.1, 0.4, 0.5, 0.6\}$, and once without random permutations. The experiments are run on 166 randomly selected hypergraphs and for $b \equiv 1$. Figure 6.4b visualizes the quality performance profile. We see

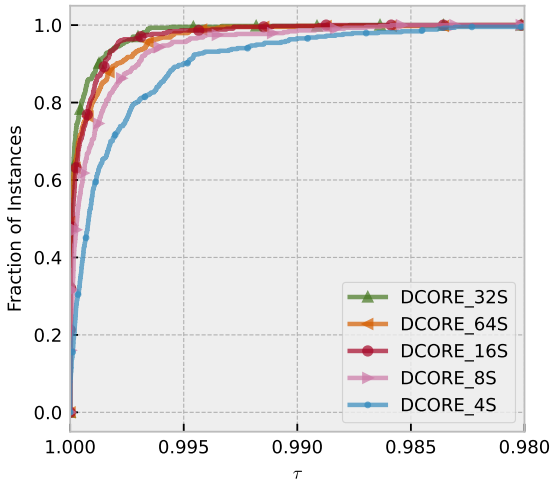
that no random permutation perform the worst and 0.5 performs the best. Comparing the algorithm without random permutations and the one with $\alpha = 0.5$, the relative improvements are in $[-0.3\%, 2.64\%]$. The average relative improvement value is 0.129% and the median value is at 0.005%.

Perturbation Using Machine B, we analyze the effect of the perturbation that is deployed when the D-Core is size zero. For this, we run the D-Core algorithm once with perturbations and once without perturbations. This experiment is run for $b \equiv 1$ and for hypergraphs that in a previous D-Core algorithm run have shown that they reach at least once in the run a D-Core of size zero. Figure 6.4c shows the quality performance plot of the two algorithms. The relative improvement values are in $[-0.038\%, 0.792\%]$. The arithmetic average is 0.042% and the median value is 0.037%.

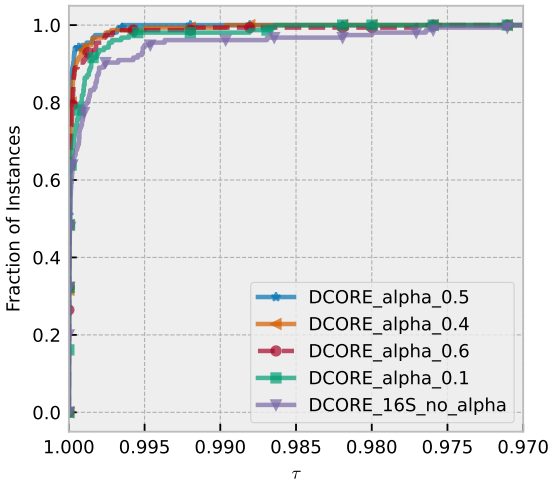
Reductions vs No-Reduction on D-Core. Using Machine B, we analyze the effect of reductions on the D-Core. For this, we execute the D-Core algorithm once with reductions on the D-Core and once without reductions on the D-Core. This is analyzed with respect to runtime and quality. For the quality comparison, we run the two algorithms for the same time and compare the quality of 1-matchings. For the runtime metric, we first run the algorithm without the reductions and then we run the algorithm with reductions and break the D-Core iteration loop as soon as the solution has surpassed the quality of solution computed by the algorithm without the reductions. The experiments show that the reductions on the D-Core improve the algorithm. Figure 6.5 (left) is the time performance profile for the two algorithms. The speed up values are in $[0.37, 20.29]$. The arithmetic average is at 3.80 and the median value is at 2.76. Furthermore, we can see in Figure 6.5 (right) that the reduction-based algorithm is capable of finding solutions of better quality. The arithmetic average of the quality ratio values is 1.0017. The improvements are in $[0.9914, 1.0091]$ and the median value is 1.0015.

Parallel D-Core Set Calculation. Using Machine A, we examine in this experiment the speedup achieved by the parallel version of the D-Core set calculation. For $b \equiv 1$ We initiate a run for every hypergraph with 64 base solutions. The algorithm runs a local search for 30 seconds. After the local search algorithm has finished, the D-Core is computed once sequentially and once in parallel. We measure the time for both computations. The speedup is the runtime of the sequential D-Core set calculation divided by the parallel D-Core set calculation. The speedup values over all hypergraphs are in $[2.21, 30.34]$. The arithmetic average is at 14.74 and the median value is at 14.71.

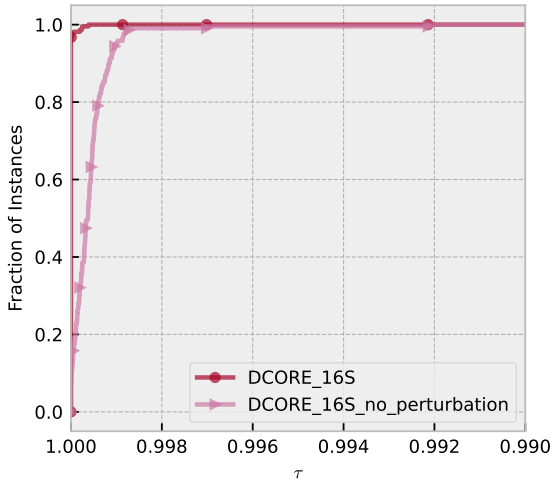
D-Core Size. Using Machine B, we analyze the size of the D-Core and the size of the reduced D-Core. Over all hypergraphs, the local search is executed for 30 seconds, after which the D-Core is computed, as well as the reduced D-Core. The experiments are executed for $b \equiv 1$ and 16 base solution. We measure the size of a hypergraph in the number



(a) Quality performance profile plot for the D-Core algorithm run for different number of base solutions.



(b) Quality performance profile the D-Core algorithm run with different alpha values and once without any randomization.



(c) Quality performance profile the D-Core algorithm run once with perturbation and once without perturbation.

Figure 6.4: For $b \equiv 1$ and $250s$ runtime, three plots for the parameter experiments.

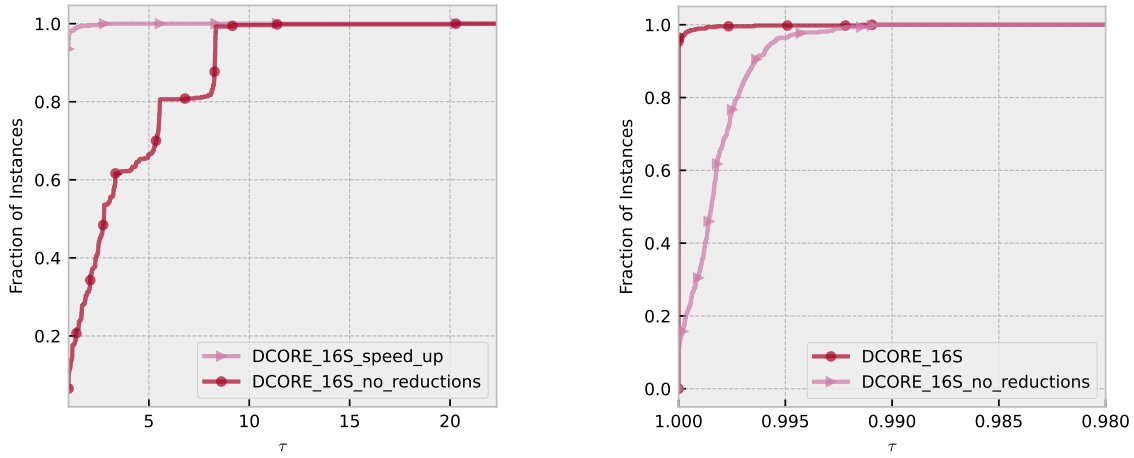


Figure 6.5: On the left: A time performance profile that that runs the D-Core algorithm without reductions on the D-Core and once with reductions on the D-Core stopping when it surpasses the run without reductions. On the right: A quality performance profile that compares a run of the D-Core algorithm for 250 seconds once with reductions and once without reduction.

of edges. The average size of the D-Core relative to the size of the original hypergraph is 11.93%, the median value is at 10.34%. All values are in $[0\%, 48.78\%]$. The average size of the reduced D-Core relative to the size of the original hypergraph is 5.79% and the median value is at 1.6%. Comparing the kernel to the D-Core, on average the kernel that has a size of 41.19% and a median value at 29.46%. All relative kernel sizes are in $[0.01\%, 100\%]$. In absolute numbers, the size of the D-Core is on average $161\,322$ edges. The size of the reduced D-Core is on average $61\,456.9$ edges.

D-Core Iterations. Using Machine B, we measure the advantage that the D-Core allows to concentrate the computational resources on a smaller subgraph. We do this by measuring the performance of the local search on the D-Core compared to the entire hypergraph. As mentioned in Section 3, the local search procedure is a repetition of perturbation and optimization steps. These steps are repeated as long as a time limit is not reached. In this experiment, we measure the performance of the local search by counting the number of iteration rounds the local search can do in 30 seconds on the entire hypergraph compared to the number of rounds in 30 seconds on the D-Core. The experiments are executed for $b \equiv 1$ and 16 base solution. We define the speedup as the number of iterations that are counted on the D-Core divided by the number of iterations that are counted on the entire hypergraph. The speedup values are in $[1.28, 7\,764.87]$. On average, the speedup values are at 290.55 and the median is at 24.86.

D-Core vs Gurobi and Local Search. Using Machine B, we compare in this experiment the Gurobi optimization tool and the local search with the D-Core algorithm. All algorithms are executed for 250 seconds with 16 available cores. For a fair comparison, we adapt the local search. We execute 16 runs of the local search simultaneously (on 16 cores) and take the best result of the 16 solutions. Furthermore, we initialize this local search with the same greedy randomization algorithm as for the D-Core algorithm. The algorithm is denoted by LocalSearch_Max16. For a better overview, we also include the single-core run of the local search in the plot. For 584 hypergraph, Gurobi fails due to timeout for $b \equiv 1$ on 39 instances; for $b \equiv 3$ on 28 instances; for $b \equiv r$ on 29 instances, the D-Core algorithm fails due to timeout for $b \equiv 1$ on 39 instances; for $b \equiv 3$ on 40 instances; for $b \equiv r$ on 43 instances; the adapted local search algorithm fails due to timeout for $b \equiv 1$ on 39; for $b \equiv 3$ on 35; for $b \equiv r$ on 30. The quality ratio is the weight of the solution computed by the D-Core algorithm divided by the weight of the solutions computed by Gurobi or the adapted local search. For $b \equiv 1$, Figure 6.6a (left) shows the quality performance profile. The quality ratio values compared to Gurobi are in $[0.984, 34.65]$. We see an arithmetic average of 1.32 and a median value of 0.999. The quality ratio values compared to the adapted local search are in $[0.993, 1.063]$. We see an arithmetic average of 1.005 and a median value of 1.003. For $b \equiv 3$, Figure 6.6b (left) shows the quality performance profile. The quality ratio values are in $[0.981, 187.295]$. We see an arithmetic average of 3.71 and a median value of 0.998. The quality ratio values compared to the adapted local search are in $[0.995, 1.035]$. We see an arithmetic average of 1.004 and a median value of 1.003. For $b \equiv rand.$, Figure 6.6c (left) shows the quality performance profile. The quality ratio values are in $[0.96, 3.85]$. We see an arithmetic average of 1.01 and a median value of 0.998. The quality ratio values compared to the adapted local search are in $[1.00001, 1.0193]$. We see an arithmetic average of 1.004 and a median value of 1.003.

Speedup D-Core vs Local Search. Using Machine B, we compare the local search algorithm to the D-Core algorithm with respect to runtime. For this, we first run the local search for 250 seconds. Afterwards, we run the D-Core algorithm and stop the D-Core iteration loop as soon as it has surpassed the local search solution (measured in solution weight). For $b \equiv 1$, Figure 6.6a (right) shows the time performance profile. The speedup values are in $[0.94, 8.33]$. The arithmetic average is 6.00 and the median value is 5.53. For $b \equiv 3$, Figure 6.6b (right) shows the time performance profile. The speedup values are in $[0.95, 8.33]$. The arithmetic average is 6.35 and the median value is 7.74. For $b \equiv rand.$, Figure 6.6c (right) shows the time performance profile. the speedup values are in $[0.87, 8.33]$. The arithmetic average is 6.90 and the median value is 8.17.

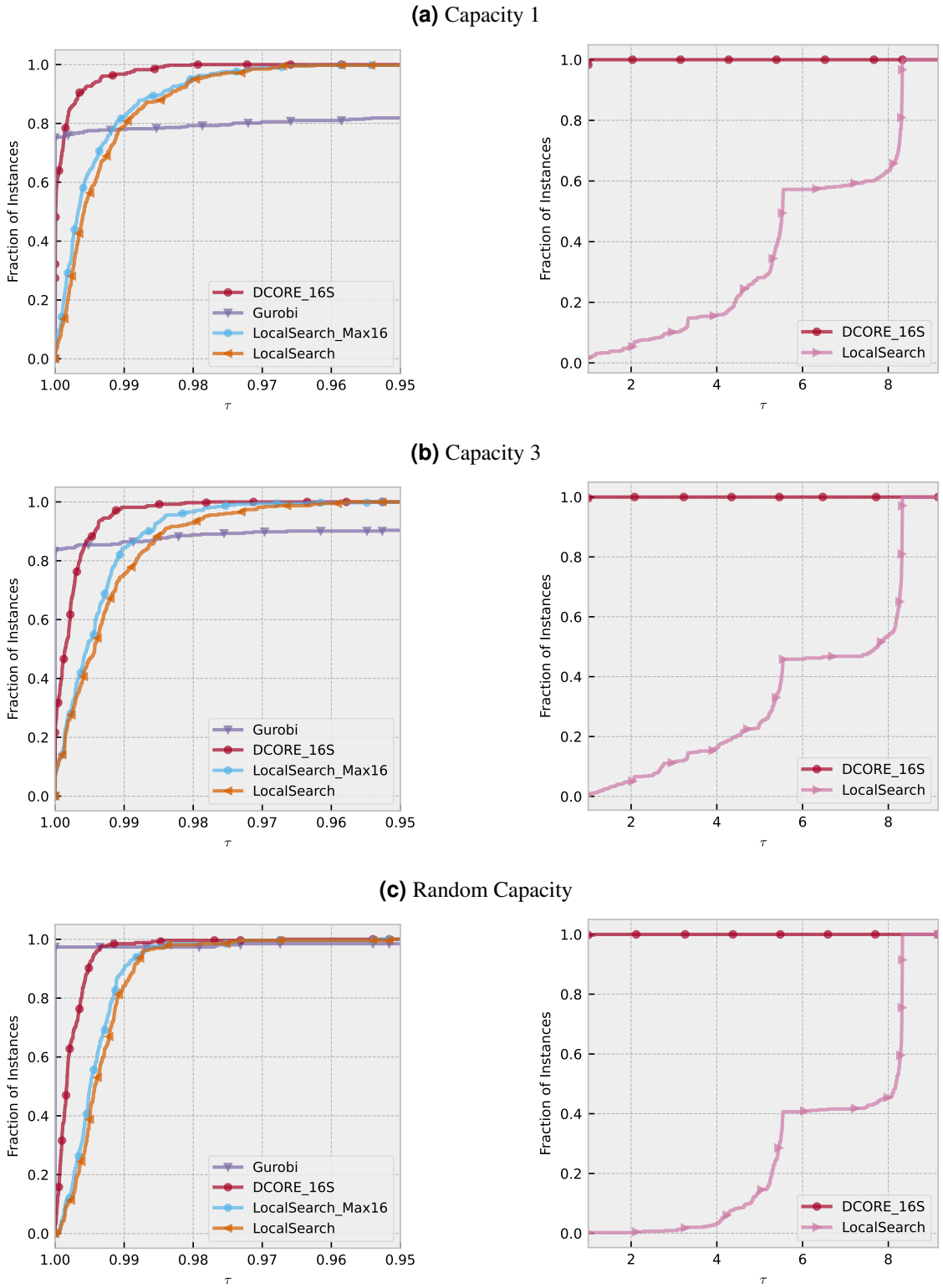


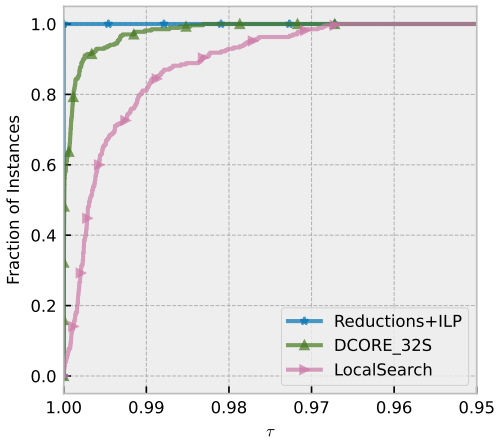
Figure 6.6: For three different b-functions; On the left column: Quality performance profile plots for Gurobi, D-Core algorithm and LocalSearch with equal number of processing elements and 250s runtime. On the right column: Time performance profile plots for a D-Core algorithm that is stopped as soon as it finds a solution of the same weight as the single-core 250s Local Search algorithm.

Comparison to Exact Solutions. Using Machine A, we will compare the results of the D-Core algorithm to only exact solutions. The exact solutions are obtained by reduction and running Gurobi for $1\,500$ seconds. The D-Core is run for 250 seconds and 32 base solutions. The solver failed to compute the exact solutions for 244 hypergraphs for $b \equiv 1$, for 219 hypergraphs for $b \equiv 3$, 113 for $b \equiv rand.$. For $b \equiv 1$, Figure 6.7a visualizes the quality performance profile plot. It shows that the D-Core is capable to outperform the local search and get closer to the exact solutions. The quality ratio values of the D-Core compared to the exact solutions are in $[0.983, 1]$. The arithmetic average of the ratio values over all hypergraphs is 0.998 and the median value is 0.999. The quality ratio values of the local search compared to the exact solutions are in $[0.967, 1]$. The arithmetic average of the ratio values over all hypergraphs is 0.993 and the meadian value is 0.996. This means that the D-Core algorithm is capable of decreasing the difference to the exact solution in comparison to the local search on average by 3.5 times.

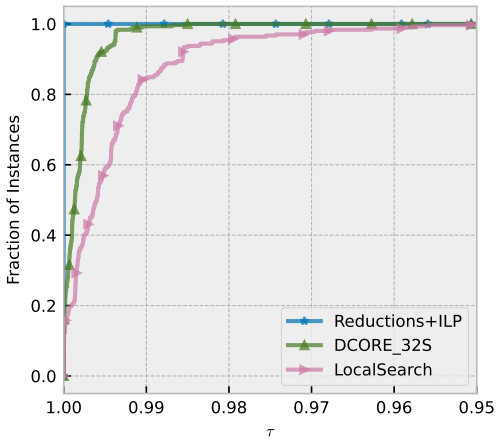
For $b \equiv 3$, Figure 6.7b visualizes the quality performance profile plot. It shows that the D-Core is capable of producing better solutions. The quality ratio values of the D-Core compared to the exact solutions are in $[0.985, 1]$. The arithmetic average of the ratio values over all hypergraphs is 0.998 and the median value is 0.998. The quality ratio values of the local search compared to the exact solutions are in $[0.863, 1]$. The arithmetic average of the ratio values over all hypergraphs is 0.993 and the median value is 0.995. This means that the D-Core algorithm is capable of decreasing the difference to the exact solution in comparison to the local search on average by 3.5 times.

For $b \equiv rand.$, Figure 6.7c visualizes the quality performance profile plot. It shows that the D-Core produces results that are closer to the optimal values. The quality ratio values of the D-Core compared to the exact solutions are in $[0.983, 1]$. The arithmetic average of the ratio values over all hypergraphs is 0.997 and the median value is 0.998. The quality ratio values of the local search compared to the exact solutions are in $[0.972, 1]$. The arithmetic average of the ratio values over all hypergraphs is 0.993 and the median value is 0.994. This means that the D-Core algorithm is capable of decreasing the difference to the exact solution in comparison to the local search on average by 2.3 times.

(a) Capacity 1



(b) Capacity 3



(c) Random Capacity

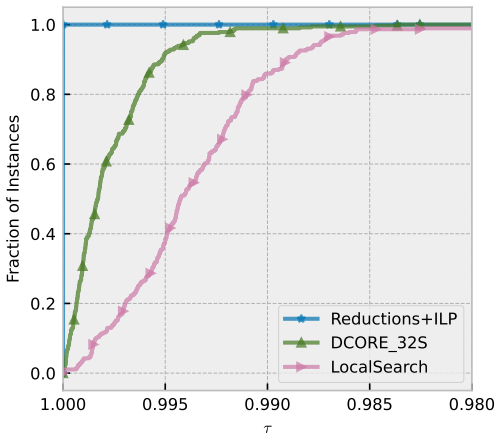


Figure 6.7: Quality performance profile plots of D-Core and local search with 250 seconds runtime for each instance over all hypergraphs for three capacity functions compared to exclusively exact solutions.

CHAPTER 7

Discussion

In this chapter, we discuss the results of the experiments. Afterwards, we conclude the thesis and give an outlook on future ideas and approaches.

Reductions. The theoretical improvement in runtime complexity for finding edges for Weighted Domination shows a clear improvement in runtime in the experiments. A small percentage of graphs that did not improve sounds appropriate, since some instances might not behave according to the worst case. More applicable edges can be found for Weighted Edge Folding. The small decrease in kernel size seems reasonable, because of the constraint restrictions and the theoretical condition is only a slight extension. Even though Isolated Weight Transfer and Tuple Folding do not decrease the number of edges significantly, the clear improvement in kernel size for all combined reductions still makes sense, since the new reductions enable old reduction to find more applicable edges. Subset Vertices, Tuple Folding and Isolated Weight Transfer are observed to be the most time-consuming compared to the other reductions, which matches the fact that they have the worst worst-case time complexity. Although Tuple Folding and Isolated Weight Transfer have the same worst-case time complexity, the algorithm for Isolated Weight Transfer performs faster. We assume that this is due to the fact that Isolated Weight Transfer has an additional restriction, namely not reviewing edges that have an edge size larger than some constant. The parallel version of the reduction algorithms is capable of finding reductions faster. A sublinear scaling of the speedup values is is reasonable, since some parts of the parallel algorithms are executed sequentially and some redundant operations are executed when, for example, gathering two applicable edges that are applicable by themselves but not in combination. The reductions improve the runtime of an ILP solver, even by such a factor that the combined time of the reductions and the ILP solver is smaller than the ILP solver without reductions. For this measurement, the algorithm slows down in some instances since the reductions do not decrease the kernel for all hypergraphs. The speedup decreases slightly for $b \equiv 3$ and $b \equiv rand$, we assume this is because the new edge reductions are more effective for uniform capacity which the experiments on the kernel sizes show.

Local Search. The high speedup values for the copying algorithm can be explained by two factors. Firstly, M_1 and M_2 are often similar matchings, only differing by a few edges. Secondly, in the data structure, the time complexity of adding and removing edges is in $O(\Delta_E \Delta_V)$ which slows down the algorithm because in the old copying method every edge in the matching is removed and most of them are added back which leads to many addition and removals which the new copying method not do. Predicting the effects of a heuristic is a challenging task. However, since the $(1, 1)$-Swap algorithm has a reasonable runtime and some data that was computed for the $(1, 2)$-Swaps could be reused for $(1, 1)$-Swaps, and the $(1, 1)$-Swaps allow the local search to further explore the solution space, an improvement of up to 7% is not too surprising.

D-Core. Decreasing the instances with the D-Core allows the algorithm to concentrate its computational resources, which leads to an increase in speed of the local search. Interestingly for $b \equiv 1$, the average relative kernel size of the D-Core is at 40% while the kernel size of the entire graph is on average 60%. Although the reduced D-Core is on average roughly 20 times smaller than the entire graph, the local search on the D-Core is on average 290 faster on the D-Core than on the entire graph. This improvement could be explained through the fact that some sub-algorithms in the D-Core algorithm depend on the number of adjacent edges often also in a superlinear relation, for example when checking if an edge is feasible for a $(1, 2)$-Swap. The observed speedup in the parallel D-Core set calculation is reasonable. We assume that the sublinear scaling of the speedup values is due to the following fact. The worst case-time complexity of the sequential algorithm is only achieved if the last solution is the only one that differs from the other solution which seems rather unlikely. The experiments show that, for some hypergraphs, Gurobi is barely able to find a feasible solution. For these instances, the D-Core performs better. However, for $b \equiv 3$ and $b \equiv rand$ the fraction of these instances decreases. Furthermore, the optimization tool can prove that a solution is optimal, which also allows the algorithm to terminate at the point of proving the optimality. The D-Core algorithm outperforms the local search procedure on average and on most instances for all tested b-functions, which seems reasonable since it uses the same optimization and perturbation techniques and is able to concentrate its computational resources on a smaller instance. Furthermore, the D-Core algorithm heuristically looks at a subgraph where solutions seem to have difficulties agreeing. It outperforms the adapted local search on average and on most instances, which makes use of the same computational resources. These results fit to the explanation for single-core run.

7.1 Conclusion

In this thesis, we have developed and implemented reductions and a heuristic for the hypergraph b-matching problem. We presented seven novel contributions for the exact reductions for hypergraph b-matching. This includes applying two reduction techniques from the in-

dependent set problem to hypergraph b-matching, namely Isolated Weight Transfer and Tuple Folding. We extended three reductions by relaxing some conditions, and we show the correctness of the relaxed versions. Additionally, we provided two new vertex reductions, namely, Subset Vertices and Subset Duo Vertices. The experiments showed that the reductions effectively decrease the kernel size and it revealed that the parallelization of the reductions is a successful technique to speed up the computations. Furthermore, the reductions help the optimization tool to find solutions in a shorter time. One advantage of the D-Core heuristic is the concentration of computational resources, which could be observed in our experiments. The D-Core algorithm begins by initializing solutions with a greedy randomized algorithm, which helps spread the base solutions across the solution space. If the D-Core is zero, then the base solutions are perturbed, potentially finding new areas in the solution space. We observed that the optimization tools seem to have difficulties in finding a feasible solution for some hypergraphs. For these hypergraphs, heuristic tools such as the local search and the D-Core find better solutions. We furthermore saw that the proportion of these hypergraphs decreases for b-functions that map to higher values. Compared to the local search for all tested b-functions, the D-Core heuristic improved the quality of the solutions, and it could also be observed that the D-Core algorithm finds b-matchings of equal quality in a shorter time.

7.2 Future Work

The success of the D-Core for the independent set problem and for the hypergraph b-matching problem suggest it may be feasible for other problems too. Such problems could be Vertex Cover, SAT, Graph Coloring, Knapsack or Feedback Vertex Set. A variation of the current algorithm could be to calculate the D-Core from solutions that were computed by more diverse algorithms. Currently, all solutions that define a D-Core are calculated by the same algorithm with different randomization seeds. This could be extended by including suboptimal solutions that were retrieved by algorithms such as a time-limited ILP solver or a time-limited branch-and-bound algorithm. We see that the runtime complexity of many (sub-)algorithms in our implementation depends on the edge size or the number of incident edges such as in the operations for the data structure. This leads to long computation times for dense graphs and raises the question of how to deal with these cases in the feature. It might be worth to let go off the current data structure in these cases, or/and to look into heuristics or reductions that tackle "high degree" edges. Subset Duo Vertices, General Weighted Domination and General Tuple Folding remain unimplemented due to limited time. Compared to the other reductions, the runtime of the newly implemented once is proportionally large. This raises the question whether there are some restrictions that can be applied such that the speed increases but not too many applicable edges/vertices are lost. The exhaustive reductions resulted in some kernels of size less than 100 edges, which (at first glance) may seem like a good starting point to find more reductions or a natural barrier.

APPENDIX A

Further Results

D-Core vs SCIP and Local Search. Here, we compare the D-Core algorithm to the data that resulted in the experiments by Großmann et al. [30]. For the methodology that was used to execute the experiments, we refer to [30]. For SCIP, an open-source optimization tool [1], the computation on one hypergraph was limited to run for 1500 seconds. If SCIP did not find the best solution after 1500 seconds, it returns the current best. Additionally, we compare the D-Core algorithm to iterative local search (ILS) algorithm by [30]. The algorithm is limited by the number of consecutive iterations rounds without improvements which was set to 1500. Figure A.1 (left column) visualizes a quality performance profile for the 3 algorithms. Since the local search has a different stopping criteria, we compare in Figure A.1 (right column) the runtime. In total, we see that D-Core is able to outperform both algorithms in quality. For $b \equiv 1$, the relative improvement values of the solution quality of the D-Core compared to the local search are in $[0.21\%, 8.96\%]$. The arithmetic average is 2.29% and the median value is 1.98%. Additionally, the D-Core algorithm runs on average 26.3 times faster than the local search with speedup values in $[0.36, 132.22]$ and a median value at 7.17. The quality ratio values of the D-Core algorithm compared to SCIP are in $[0.98, 2013.74]$ with an arithmetic average at 15.85 and a median value at 1.0092. For $b \equiv 3$, the relative improvement values of the solution quality computed the D-Core compared to the local search are in $[-0.085\%, 7.277\%]$. The arithmetic average is 1.69% and the median value is 0.948%. Additionally, the D-Core algorithm runs on average 20.7 times faster than the local search with speedup values in $[0.03, 232.01]$ and a median value at 6.71. The quality ratio values of the D-Core algorithm compared to SCIP are in $[0.98, 179\,206]$ with an arithmetic average at 451.37 and a median value at 0.999. For $b \equiv rand$, the relative improvement values of the solution quality computed by the D-Core compared to the local search are in $[-0.03\%, 4.54\%]$. The arithmetic average is 1.14% and the median value is 1.01%. Additionally, the D-Core algorithm runs on average 15.39 times faster than the local search with speedup values in $[0.31, 74.99]$ and a median value at 8.16. The quality ratio values of the D-Core algorithm compared to SCIP are in $[0.97, 9803]$ with an arithmetic average at 24.78 and a median value at 0.999.

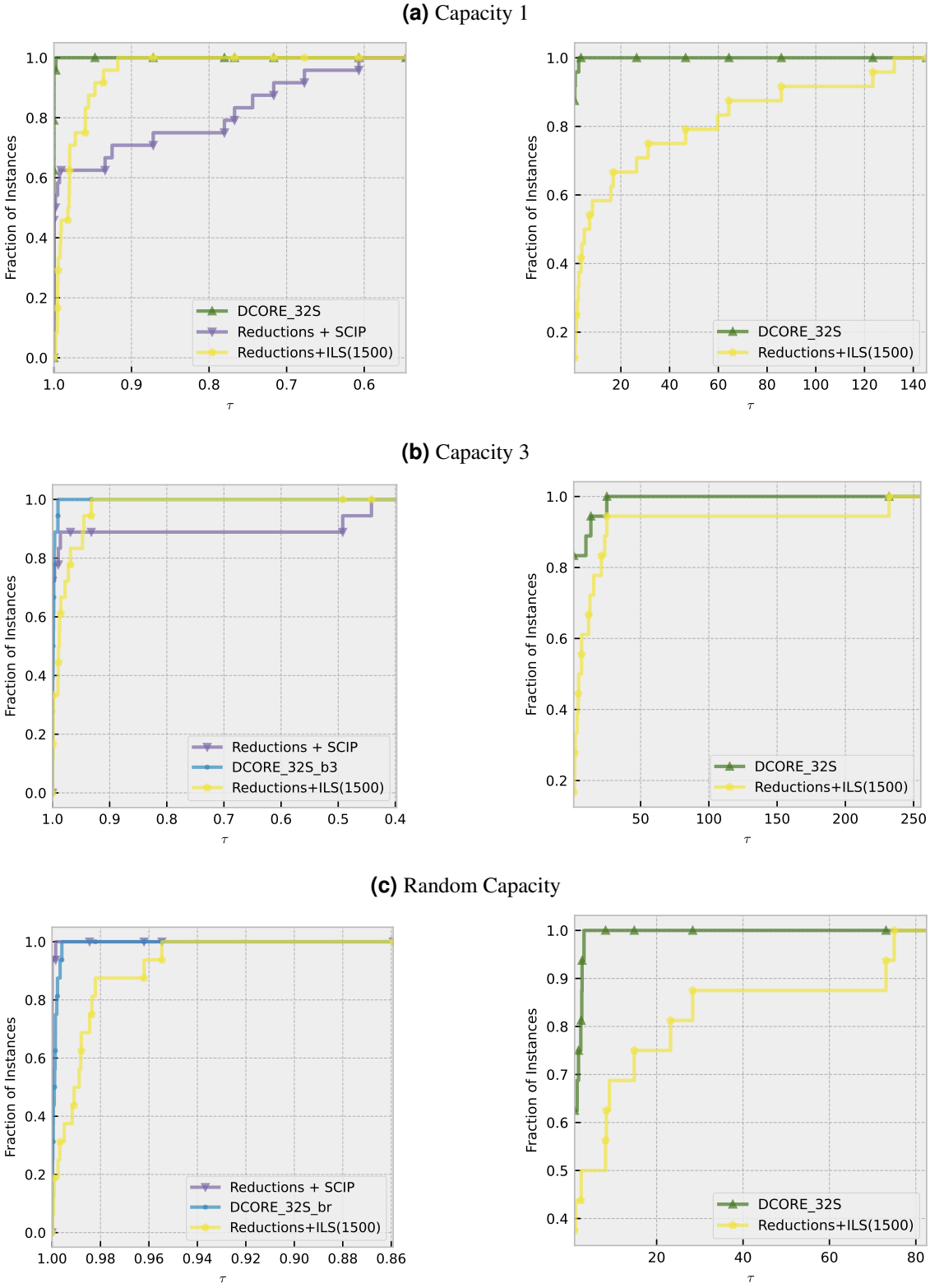


Figure A.1: For three b-functions; On the left column: Quality Performance Profile Plots for D-Core compared to local search algorithm and SCIP. On the right column: Time Performance Profile Plots for D-Core compared to the local search algorithm.

Zusammenfassung

Das Matching Problem ist ein klassisches kombinatorisches Problem der Graphen Theorie. Dazu exisitiert eine Variation namens Hypergraph b-Matching. In der Graphenstruktur von dieser Variation dürfen Kanten mehr als zwei Knoten verbinden und im Matching dürfen ausgewählte Knoten in mehr als einer Matching Kante enthalten sein. Im Gegensatz zum klassischen Matching Problem ist Hypergraph b-Matching nur in polynomieller Zeit lösbar wenn $P = NP$. Das Hypergraph b-Matching Problem gehen wir in dieser Arbeit mit einer Heuristik und Reduktionen an. Die Heuristik verkleinert den Lösungsraum und erlaubt es die Rechenressourcen zu konzentrieren. Das macht sie, in dem sie mehrere Lösungen berechnet, und einen Subgraphen bestimmt aus den Kanten, für die die Lösungen sich uneinig sind, ob die Kanten im Matching sind oder nicht. Zusätzlich stellen wir einen Abweichungs- und ein Initizierungsalgorithmus vor, sowie Parallelisierungen von Unteralgorithmen der Heuristik. Für die Reduktionen stellen wir sieben neue Techniken vor. Das beinhaltet Generalisierungen von bereits existierenden Reduktionen, die Übertragung von Reduktionen von dem Stabilen Mengen Problem und zwei neue Reduktionen. Diese werden dann auch parallelisiert. Zuletzt stellen wir zwei Verbesserungen für den Algorithmus der Loken Suche vor. Die Experimente ergeben, dass der Kern der Reduktion im Durchschnitt 25% kleiner wird für $b \equiv 1$ und die parallelen Algorithmen mit 64 Prozessoren einen durchschnittlichen Beschleunigungsfaktor von 32 erreicht. Die Reduktionen beschleunigen das Optimierungsprogamm Gurobi bei der Suche nach exakten Lösungen um einen durschnittlichen Faktor von 2.26 für $b \equiv 1$. Auch zeigen die Experimente, dass die beiden Verbesserungen der lokalen Suche die Lösungen um bis zu 7.61% verbesseren, durchschnittlich um 0.57%. Zuletzt zeigen die Experimente auch, dass die Heuristik Lösungen berechnet, die ein Gewicht von 99.8% der optimal Lösungen haben für Hypergraph, bei denen wir die genaue Lösung berechnen konnten. Verglichen mit der lokalen Suche reduziert die Heuristik im Durchschnitt 3.5 mal die Distanz zu den optimalen Lösungen und berechnet Lösungen, die bis zu 6.3% besser sind. Weiterhin, ist die Heuristik in der Lage die Lösungen der lokalen Suche mit 16 Prozessoren im Durchschnitt 6 mal schneller zu erreichen. Ähnliche Ergebnisse gehen auch aus Experimenten mit anderen b-Funktionen hervor. Für zufällige Knotenkapazitäten, findet das Optimierungsprogramm Gurobi bei den meisten Instanzen die besten Lösungen.

Bibliography

[1] Tobias Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009. doi: 10.1007/s12532-008-0001-1.

[2] Charles J. Alpert. The ispd98 circuit benchmark suite. In *Proceedings of the 1998 International Symposium on Physical Design*, ISPD '98, page 80–85, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 158113021X. doi: 10.1145/274535.274546. URL https://doi.org/10.1145/274535.274546.

[3] Diogo V. Andrade, Mauricio G. C. Resende, and Renato F. Werneck. Fast local search for the maximum independent set problem. In Catherine C. McGeoch, editor, *Experimental Algorithms*, pages 220–234, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-68552-4.

[4] Georg Anegg, Haris Angelidakis, and Rico Zenklusen. *Simpler and Stronger Approaches for Non-Uniform Hypergraph Matching and the Füredi, Kahn, and Seymour Conjecture*, pages 196–203. doi: 10.1137/1.9781611976496.22. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611976496.22.

[5] Anton Belov, Daniel Diepold, {Marijn J.H.} Heule, and Matti Järvisalo, editors. *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. University of Helsinki, Finland, 2014.

[6] Bert Besser and Matthias Poloczek. Greedy matching: Guarantees and limitations. *Algorithmica*, 77(1):201–234, 2017. doi: 10.1007/s00453-015-0062-2. URL https://doi.org/10.1007/s00453-015-0062-2.

[7] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. Efficient parallel and external matching. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, pages 659–670, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40047-6.

[8] Yuk Hei Chan and Lap Chi Lau. *On Linear and Semidefinite Programming Relaxations for Hypergraph Matching*, pages 1500–1511. doi: 10.1137/1.9781611973075.122. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611973075.122.

[9] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. *Operations Research Forum*, 3(20), 2022. doi: 10.1007/s43069-021-00101-z.

[10] Marek Cygan. Improved approximation for 3-dimensional matching via bounded pathwidth local search. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 509–518, 2013. doi: 10.1109/FOCS.2013.61.

[11] Marek Cygan, Fabrizio Grandoni, and Monaldo Mastrolilli. *How to Sell Hyperedges: The Hypermatching Assignment Problem*, pages 342–351. doi: 10.1137/1.9781611973105.25. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611973105.25.

[12] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011. ISSN 0098-3500. doi: 10.1145/2049662.2049663. URL https://doi.org/10.1145/2049662.2049663.

[13] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002. ISSN 1436-4646. doi: 10.1007/s101070100263. URL https://doi.org/10.1007/s101070100263.

[14] Yuanyuan Dong, Andrew V. Goldberg, Alexander Noe, Nikos Parotsidis, Mauricio G. C. Resende, and Quico Spaen. A metaheuristic algorithm for large maximum weight independent set problems. *Networks*, 85(1):91–112, October 2024. ISSN 1097-0037. doi: 10.1002/net.22247. URL http://dx.doi.org/10.1002/net.22247.

[15] Doratha E Drake and Stefan Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85(4):211–213, 2003. ISSN 0020-0190. doi: https://doi.org/10.1016/S0020-0190(02)00393-9. URL https://www.sciencedirect.com/science/article/pii/S0020019002003939.

[16] Ran Duan, Seth Pettie, and Hsin-Hao Su. Scaling algorithms for approximate and exact maximum weight matching, 2011. URL https://arxiv.org/abs/1112.0790.

[17] Ran Duan, Seth Pettie, and Hsin-Hao Su. Scaling algorithms for weighted matching in general graphs. *ACM Transactions on Algorithms*, 14(1):1–35, 2018. doi: 10.1145/3155301. URL https://doi.org/10.1145/3155301.

[18] Fanny Dufossé, Kamer Kaya, Ioannis Panagiotas, and Bora Uçar. Effective Heuristics for Matchings in Hypergraphs. In Ilias S. Kotsireas, Panos M. Pardalos, Konstantinos E. Parsopoulos, Dimitris Souravlias, and Arsenis Tsokas, editors, *Analysis*

of Experimental Algorithms - Special Event, SEA² 2019, Kalamata, Greece, June 24-29, 2019, Revised Selected Papers, volume 11544 of *Lecture Notes in Computer Science*, pages 248–264, 2019. doi: 10.1007/978-3-030-34029-2_17. URL https://doi.org/10.1007/978-3-030-34029-2_17.

[19] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17: 449–467, 1965. doi: 10.4153/CJM-1965-045-4.

[20] Jack Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards, Section B: Mathematics and Mathematical Physics*, 69B, 1965. doi: 10.6028/JRES.069B.013. URL https://doi.org/10.6028/JRES.069B.013.

[21] Mourad El Ouali and Gerold Jäger. The b-matching problem in hypergraphs: Hardness and approximability. In Guohui Lin, editor, *Combinatorial Optimization and Applications*, pages 200–211, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31770-5.

[22] Mourad El Ouali, Antje Fretwurst, and Anand Srivastav. Inapproximability of b-matching in k-uniform hypergraphs. In Naoki Katoh and Amit Kumar, editors, *WALCOM: Algorithms and Computation*, pages 57–69, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19094-0.

[23] M. Fujii, T. Kasami, and K. Ninomiya. Optimal sequencing of two equivalent processors. *SIAM Journal on Applied Mathematics*, 17(4):784–789, 1969. doi: 10.1137/0117070. URL https://doi.org/10.1137/0117070.

[24] Z. Füredi, J. Kahn, and P. D. Seymour. On the fractional matching polytope of a hypergraph. *Combinatorica*, 13(2):167–180, 1993. ISSN 1439-6912. doi: 10.1007/BF01303202. URL https://doi.org/10.1007/BF01303202.

[25] Martin Fürer and Huiwen Yu. Approximating the $$k$$-set packing problem by local improvements. In Pierre Fouilhoux, Luis Eduardo Neves Gouveia, A. Ridha Mahjoub, and Vangelis T. Paschos, editors, *Combinatorial Optimization*, pages 408–420, Cham, 2014. Springer International Publishing. ISBN 978-3-319-09174-7.

[26] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[27] Alexander Gellner, Sebastian Lamm, Christian Schulz, Darren Strash, and Bogdán Zaválnij. *Boosting Data Reduction for the Maximum Weight Independent Set Problem Using Increasing Transformations*, pages 128–142. doi: 10.1137/1.9781611976472.10. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611976472.10.

[28] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable high-quality hypergraph partitioning. *ACM Trans. Algorithms*, 20(1), January 2024. ISSN 1549-6325. doi: 10.1145/3626527. URL https://doi.org/10.1145/3626527.

[29] Ernestine Großmann, Kenneth Langedal, and Christian Schulz. *Accelerating Reductions Using Graph Neural Networks for the Maximum Weight Independent Set Problem*, pages 155–168. doi: 10.1137/1.9781611979084.12. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611979084.12.

[30] Ernestine Großmann, Felix Joos, Henrik Reinstädtler, and Christian Schulz. Engineering hypergraph b-matching algorithms. *Journal of Graph Algorithms and Applications*, 30(1):1–24, 2026. doi: 10.7155/jgaa.v30i1.3166. URL https://doi.org/10.7155/jgaa.v30i1.3166.

[31] Qian-Ping Gu and Jiajian Leo Liang. Exact set packing in multimodal transportation with ridesharing system for first/last mile. In Henning Fernau and Binhai Zhu, editors, *Combinatorial Algorithms*, pages 418–432, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-98740-3.

[32] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026. URL https://www.gurobi.com.

[33] Elad Hazan, Shmuel Safra, and Oded Schwartz. On the complexity of approximating k-set packing. *Computational Complexity*, 15(1):20–39, May 2006. ISSN 1420-8954. doi: 10.1007/s00037-006-0205-6. URL https://doi.org/10.1007/s00037-006-0205-6.

[34] Johan Håstad. Clique Is Hard to Approximate Within $N^{1-\epsilon}$. *Acta Mathematica*, 182 (1):105 – 142, 1999. doi: 10.1007/BF02392825. URL https://doi.org/10.1007/BF02392825.

[35] Viggo Kann. On the approximability of np-complete optimization problems. 01 1992.

[36] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984. ISSN 1439-6912. doi: 10.1007/BF02579150. URL https://doi.org/10.1007/BF02579150.

[37] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972. ISBN 978-1-4684-2001-2. doi: 10.1007/978-1-4684-2001-2_9. URL https://doi.org/10.1007/978-1-4684-2001-2_9.

[38] Richard M. Karp and Michael Sipser. Maximum Matching in Sparse Random Graphs. In *Foundations of Computer Science, 1981. SFCS'81. 22nd Annual Symp. on*, pages 364–375. IEEE, 1981.

[39] Arif Khan, Alex Pothen, Md. Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Fredrik Manne, Mahantesh Halappanavar, and Pradeep Dubey. Efficient approximation algorithms for weighted b-matching. *SIAM Journal on Scientific Computing*, 38(5):S593–S619, 2016. doi: 10.1137/15M1026304. URL https://doi.org/10.1137/15M1026304.

[40] Christos Koufogiannakis and Neal E. Young. Distributed fractional packing and maximum weighted b-matching via tail-recursive duality. In Idit Keidar, editor, *Distributed Computing*, pages 221–238, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04355-0.

[41] Piotr Krysta. Greedy approximation via duality for packing, combinatorial auctions and routing. In Joanna Jdrzejowicz and Andrzej Szepietowski, editors, *Mathematical Foundations of Computer Science 2005*, pages 615–627, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31867-5.

[42] Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. *Exactly Solving the Maximum Weight Independent Set Problem on Large Real-World Graphs*, pages 144–158. doi: 10.1137/1.9781611975499.12. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611975499.12.

[43] Jens Maue and Peter Sanders. Engineering algorithms for approximate weighted matching. In Camil Demetrescu, editor, *Experimental Algorithms*, pages 242–255, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-72845-0.

[44] John W. May. *Cheminformatics for genome-scale metabolic reconstructions*. PhD thesis, Apollo - University of Cambridge Repository, 2015. URL https://www.repository.cam.ac.uk/handle/1810/246652.

[45] Silvio Micali and Vijay V. Vazirani. An o(sqrt(|v|) |e|) algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*, pages 17–27. IEEE Computer Society, 1980. doi: 10.1109/SFCS.1980.12. URL https://doi.org/10.1109/SFCS.1980.12.

[46] Angelo Monfroglio. Integer programs for logic constraint satisfaction. *Theoretical Computer Science*, 97(1):105–130, 1992. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(92)90389-W. URL https://www.sciencedirect.com/science/article/pii/030439759290389W.

[47] Bruno Nogueira, Rian G. S. Pinheiro, and Anand Subramanian. A hybrid iterated local search heuristic for the maximum weight independent set

problem. *Optimization Letters*, 12(3):567–583, May 2018. ISSN 1862-4480. doi: 10.1007/s11590-017-1128-7. URL https://doi.org/10.1007/s11590-017-1128-7.

[48] David A. Papa and Igor L. Markov. Hypergraph partitioning and clustering. In *Handbook of Approximation Algorithms and Metaheuristics*, 2007. URL https://api.semanticscholar.org/CorpusID:1159990.

[49] Ojas Parekh and David Pritchard. *Generalized Hypergraph Matching via Iterated Packing and Local Ratio*, page 207–223. Springer International Publishing, 2015. ISBN 9783319182636. doi: 10.1007/978-3-319-18263-6_18. URL http://dx.doi.org/10.1007/978-3-319-18263-6_18.

[50] Robert Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In Christoph Meinel and Sophie Tison, editors, *STACS 99*, pages 259–269, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-49116-3.

[51] Henrik Reinstädtler, S M Ferdous, Alex Pothen, Bora Uçar, and Christian Schulz. Semi-Streaming Algorithms for Hypergraph Matching. In Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman, editors, *33rd Annual European Symposium on Algorithms (ESA 2025)*, volume 351 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 79:1–79:19, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-395-9. doi: 10.4230/LIPIcs.ESA.2025.79. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2025.79.

[52] Henrik Reinstädtler, Christian Schulz, Nodari Sitchinava, and Fabian Walliser. Efficient parallel algorithms for hypergraph matching. In Massimo Torquati, Gabriele Mencagli, Valeria Cardellini, Alessia Antelmi, and Doriana Medić, editors, *Euro-Par 2026: Parallel Processing*, pages 526–540, Cham, 2027. Springer Nature Switzerland. ISBN 978-3-032-35251-4.

[53] Robert L. Thorndike. The problem of classification of personnel. *Psychometrika*, 15 (3):215–235, September 1950. ISSN 1860-0980. doi: 10.1007/BF02289039. URL https://doi.org/10.1007/BF02289039.

[54] R. Velásquez and M. T. Melo. A set packing approach for scheduling elective surgical procedures. In *Operations Research Proceedings 2005: Selected Papers of the Annual International Conference of the German Operations Research Society (GOR)*, pages 425–430. Springer, Berlin, Heidelberg, 2006.

[55] Natarajan Viswanathan, Charles Alpert, Cliff Sze, Zhuo Li, and Yaoguang Wei. The dac 2012 routability-driven placement contest and benchmark suite. In *Proceedings of*

the 49th Annual Design Automation Conference, DAC '12, page 774–782, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311991. doi: 10.1145/2228360.2228500. URL https://doi.org/10.1145/2228360.2228500.

[56] S. Warren and I. Hicks. Combinatorial branch-and-bound for the maximum weight independent set problem. 01 2007.

[57] Julian Yarkony, Yossiri Adulyasak, Maneesh Kumar Singh, and Guy Desaulniers. Data association via set packing for computer vision applications. *INFORMS J. Optim.*, 2:167–191, 2019. URL https://api.semanticscholar.org/CorpusID:198344009.

[58] Long Zhang, Hongliang Zhang, Lisu Yu, Haitao Xu, Lingyang Song, and Zhu Han. Virtual resource allocation for mobile edge computing: A hypergraph matching approach. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2019. doi: 10.1109/GLOBECOM38437.2019.9013384.