Finding 2-Packing Sets using Hypergraph Matching

Thanh Lan Tran

November 4, 2024

4209012

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

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

Co-Referee:
Ernestine Großmann
Henrik Reinstädtler

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken Übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, November 4, 2024

T. Tran

Thanh Lan Tran

Abstract

A 2-packing set is a subset of vertices in which no two vertices are within a distance of two from each other. This thesis explores an approach to find a 2-packing set by building a hypergraph on which a hypergraph matching solution is equivalent to a 2-packing set. We also focus on pre-processing techniques by investigating the efficiency of several existing data reduction rules for (hyper-)graphs. Additionally, we introduce new hypergraph data reduction rules inspired by established methods. Our experimental evaluation shows that some of the reductions perform exceptionally well. Furthermore, we show that while our heuristic algorithm partly only achieves a solution quality of 88% the best size, the running time on certain graph classes is exceptionally fast. Additionally, we will see that our heuristic algorithm does not consume a lot of memory.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Many real-world scenarios can be effectively modeled and analyzed using graph theory. Applications span fields such as telecommunications, logistics, and social networks. A critical challenge in these areas often involves the optimal placement of resources which can be modeled as vertices in a graph. The goal is often to avoid interference, redundancy, or other forms of conflict. One classic issue in this domain is the independent set problem, where a subset of vertices is selected such that no two vertices are adjacent. While useful in many scenarios, the independent set model assumes a relatively weak form of mutual exclusion, which is not always adequate for more demanding applications. In many practical cases, a stronger level of separation is required, ensuring greater spacing between selected vertices.

This is where the 2-packing set concept comes into play. In a 2-packing set, the selected vertices must be spaced at least three edges apart, ensuring that no two vertices are within a distance of two from each other. This criterion provides greater separation than an independent set, making the 2-packing set a more suitable model for situations where higher levels of conflict avoidance or resource isolation are necessary. For instance, in wireless networks, 2-packing sets could be used to minimize signal interference by ensuring that transmitters are not placed too close together.

From a theoretical standpoint, the 2-packing set is a natural extension of the independent set problem (which can be viewed as a 1-packing set). Like the independent set problem, the 2-packing set is NP-hard, making the development of efficient algorithms for optimal or near-optimal solutions a key challenge in graph theory and optimization.

1.2 Our Contribution

In this thesis, we propose a solver for the 2-packing set problem based on hypergraph matching. The core of our approach involves transforming the original graph into a suitable hypergraph representation. Once the graph is converted, we apply a set of data reduction rules that are crucial for simplifying the hypergraph. The reductions help us classify hyperedges as either included to the matching or excluded from the solution. In some cases, we postpone the decision until after we obtain more information on the neighboring hyperedges.

Our work introduces five new data reduction rules for the hypergraph, along with several pre-existing reduction techniques. These rules effectively decrease the number of (hyper-)edges and some vertices, which is essential for improving the efficiency of subsequent computations. We then utilize existing hypergraph matching algorithms to solve the reduced hypergraph.

To assess the effectiveness of our solver and reductions, we conduct extensive experiments, evaluating performance across multiple dimensions, including running time, memory usage, and solution quality. We benchmark our results against those of the solvers developed by Borowitz et al. [4], who approach the 2-packing set problem by transforming it into a maximum independent set problem. By comparing both approaches, we provide insights into the strengths and weaknesses of hypergraph-based solutions for tackling the 2-packing set problem.

1.3 Structure

The remainder of this thesis is organized as follows. Chapter 2 introduces the definitions and notations used throughout the thesis. Chapter 3 provides an overview of related work on 2-packing set algorithms and hypergraph matching algorithms, including a description of the solvers we used. Chapter 4 presents the general concepts and ideas behind the competing algorithms. In Section 4.1, we explain our approach, while Section 4.2 describes the competing framework developed by Borowitz et al. [4]. Section 4.3 presents the data reduction rules and provides proofs for the newly implemented ones. Chapter 5 details the experimental evaluation of our work, and Chapter 6 discusses the results and includes a section on potential future work.

CHAPTER 2

Fundamentals

2.1 General Definitions

Undirected Graphs. An *undirected graph* is a graph that consists of a set of vertices V and a set of edges E. Each edge is an unordered pair of vertices. Formally, an undirected graph G can be defined as $G = (V, E)$, where V is the set of vertices and $E \subseteq \binom{V}{2}$ is the set of edges. Each edge $e = \{u, v\} \in E$ is a set of two vertices u and v such that $u, v \in V$ and $u \neq v$. The number of vertices in G is denoted by $n = |V|$, and the number of edges is denoted by $m = |E|$. We refer to the resulting graph after the application of data reduction rules as the *kernel graph* K.

Neighborhood. Two vertices u and v are said to be *adjacent* or *neighbors* if there is an edge $e = \{u, v\} \in E$ connecting them. The *neighborhood* of a vertex v in an undirected graph is the set of vertices adjacent to v, denoted by $N(v) = \{u \in V \mid \{u, v\} \in E\}$. The *degree* of a vertex, denoted by $\deg(v)$, in an undirected graph is the same as the size of the neighborhood of v.

2-Neighborhood The *2-neighborhood* of a vertex v (denoted as $N_2(v)$) is the set of vertices that are exactly two edges away from v in a graph. The *extended 2-neighborhood* $N_2[v]$ incorporates both the 2-neighbors of v and its direct neighbors: $N_2[v] = N_2(v) \cup N(v)$.

Clique. A *clique* in an undirected graph $G = (V, E)$ is a subset of vertices $C \subseteq V$ such that every pair of distinct vertices in C is connected by an edge in E. Formally, C is a clique if for every pair of vertices $u, v \in C$, the edge $\{u, v\}$ is in E. The size of a clique is the number of vertices it contains, and a clique of size k is called a k-clique.

Independent Set. In a graph, an *independent set* is a subset of vertices such that no two vertices in the subset are adjacent. Formally, given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a subset $I \subseteq V$ is an independent set if for every pair of distinct vertices $u, v \in I$, there is no edge $\{u, v\} \in E$. The *size* of an independent set I is the number of vertices in I. An independent set I is called a *maximum independent set* if there is no other independent set I' in G such that the size of I' is greater than the size of I.

2-Packing Set. A *2-packing set* is a subset of vertices in a graph such that no two vertices in the subset are within distance two of each other. Formally, given a graph $G = (V, E)$, a subset $S \subseteq V$ is called a 2-packing set if for any distinct vertices $u, v \in S$, the distance $d(u, v)$ in G is greater than 2. The distance $d(u, v)$ is the length of the shortest path between u and v. In other words, no two vertices in a 2-packing set S are adjacent, nor do they share a common neighbor.

Hypergraph. A *hypergraph* is a generalization of a graph where edges, called *hyperedges*, can connect any number of vertices. Formally, a hypergraph H is defined as $H = (V, E_H)$, where V is a set of vertices and E_H is a multiset of hyperedges. Each hyperedge $e \in E_H$ is a non-empty subset of V. A vertex v is incident to a hyperedge e if $v \in e$. Two hyperedges are adjacent to each other if they share at least one common vertex.

Hypergraph Matching. In the context of hypergraphs, a *matching* is a set of hyperedges such that no two hyperedges share a common vertex. Formally, given a hypergraph $H = (V, E_H)$ where V is the set of vertices and E_H is the multiset of hyperedges, a subset $M \subseteq E_H$ is a matching if for any two distinct hyperedges $e_1, e_2 \in M$, we have $e_1 \cap e_2 = \emptyset$. The *size* of a matching M is the number of hyperedges in M. A matching M is called a *maximum matching* if there is no matching M' in H such that the size of M' is greater than the size of M. An extension of the hypergraph matching problem is the *hypergraph b-matching problem*. It allows each vertex to be matched with a specified number of hyperedges. This is defined by a function $b : V \to \mathbb{N}$, indicating the maximum number of hyperedges incident to each vertex.

CHAPTER 3

Related Work

This chapter is divided into three sections. In the first section, we give an overview of research concerning 2-packing set algorithms. In the second section, we examine closely related work on hypergraph matching. The third section consists of an overview of the solvers that we used in our work.

3.1 2-Packing Set Algorithms

One of the early contributions to sequential algorithms for the 2-packing set problem was made by Trejo-Sánchez et al. [21]. They developed a genetic algorithm for solving the maximum 2-packing set problem in connected arbitrary graphs. Additionally, significant research has focused on distributed algorithms for solving the 2-packing set problem. For instance, Flores-Lamas et al. [11] introduced a distributed algorithm capable of efficiently determining a maximal 2-packing set in Halin graphs. Their approach achieves linear time complexity by employing graph reduction rules and vertex partitioning methods.

Further advancements in the area have concentrated on self-stabilizing algorithms, especially for maximal 2-packing sets. Shi [19] created a self-stabilizing algorithm applicable to arbitrary graphs, while Trejo-Sánchez and Fernández-Zepeda [23, 22] adapted these techniques for cactus and geometric outerplanar graphs. These algorithms are designed to function in dynamic environments where vertices must independently adjust their status to maintain a valid 2-packing set. Ding et al. [7] introduced a self-stabilizing algorithm that guarantees safe convergence by using a process where vertices are alternately locked and unlocked depending on their conflict status with adjacent vertices.

Furthermore, recent research has explored the application of dynamic programming techniques for solving the 2-packing set problem on special graph classes. Mjelde [18] provided an algorithm for tree graphs, while Flores-Lamas et al. [10] presented a solution based on dynamic programming for cactus graphs with a running time of $O(n^2)$.

Lastly, Borowitz et al. [4] developed algorithms focused on efficiently identifying optimal 2-packing sets within arbitrary graphs. Their approach draws on concepts related to the independent set problem and features advanced data reduction methods, as well as graph transformation techniques. Their work is examined in greater detail in Chapter 3.3, as their algorithms serve as a benchmark in this thesis.

3.2 Hypergraph Matching

Hazan et al. [16] established that the maximum d-set packing problem in d-partite, d-uniform hypergraphs is limited to approximations of $O(d/\log d)$. Additionally, Håstad [15] proved that matching problems in non-uniform hypergraphs, as well as the maximum independent set problem, are NP-hard, allowing for approximations only up to $n^{1-\epsilon}$, unless $P = NP$.

Cygan [5] introduced a local search algorithm for k-set packing in d-uniform hypergraphs, achieving an approximation ratio of $(k + 1 + \epsilon)/3$, which was later optimized for runtime by Fürer and Yu [12]. Dufosse et al. [9] extended the Karp-Sipser rules from graphs to hypergraphs, incorporating a Sinkhorn-Knopp [20] normalization technique for incident tensors, although their methods are largely focused on d-partite hypergraphs with uniform edge weights. Anegg et al. [2] further advanced the field by establishing tighter bounds for LP-relaxations applicable to b-matching in non-uniform settings.

In addition, Großmann et al. [13] introduced an algorithm aimed at efficiently solving the b-matching problem in hypergraphs. Their work employs data reduction techniques and optimization strategies to enhance the performance of hypergraph algorithms. Their contributions are discussed in more detail in the following section, as their matching framework is employed in our thesis.

3.3 Competing Algorithms

In this section, we outline the four algorithms utilized in our thesis, along with a description of them. The first two solvers are maximum independent set algorithms from the competing framework developed by Borowitz et al. [4]. The remaining two solvers are hypergraph matching algorithms developed by Großmann et al. [13].

3.3.1 Solvers from the competing framework

Exact Solver

The exact solver by Borowitz et al. [4] employs a branch-and-reduce approach that utilizes the maximum independent set solver (MIS) by Lamm et al. [17]. This algorithm is designed to solve the problem by systematically exploring possible solutions while simplifying the graph and pruning unnecessary computations. The process begins with a reduction phase, where a set of predefined rules is applied to the input graph. These rules remove or modify vertices and edges in ways that do not affect the final solution, reducing the complexity of the graph. This phase continues until no further reductions can be made, resulting in a simpler, smaller graph.

Following this, a local search algorithm is applied to the reduced graph to compute an initial lower bound for the solution. This lower bound provides an estimate of the independent set's weight and is useful later for pruning the search space. The idea is to establish a baseline solution early in the process, which helps the algorithm to skip over less promising branches.

Next, the algorithm calculates an upper bound on the maximum weight of the independent set that can be obtained from the current graph. If the sum of the current solution's weight and the upper bound is less than or equal to the best solution weight found so far, the algorithm abandons that particular branch. This pruning prevents unnecessary exploration of branches that cannot improve the solution.

If the graph remains connected after reductions, the algorithm enters the branching phase. Branching means dividing the problem into two subproblems: one where a selected vertex is included in the independent set, and one where that vertex is excluded. When the graph is disconnected, the algorithm solves each connected component separately. The independent set for each component is computed independently, and the final solution is obtained by combining the results from all components.

If the algorithm does not reach a solution within a specified time limit, it switches to a greedy approach to improve the current solution. In this phase, vertices are sorted by their weight in descending order, and added to the independent set as long as they are feasible, meaning they do not conflict with those already in the set. Ultimately, the algorithm continues this process of reduction, branching, pruning, and, if necessary, time-limited greedy improvement, until it finds the optimal solution.

Heuristic solver

The heuristic algorithm by Borowitz et al. [4] uses the MIS solver OnlineMIS by Dahlum et al. [6]. The algorithm begins by applying basic reductions such as removing isolated vertices or those with low degrees, marking them as part of the independent set or removing them if necessary. It also uses an adaptive strategy to handle high-degree vertices by marking the top 1% of them as removed from consideration, which speeds up the search process.

Once the reductions are applied, the algorithm starts with an initial solution, generated either through a greedy or maximal approach. The algorithm then iteratively improves this solution by forcing specific nodes into the independent set and removing conflicting neighbors, exploring different configurations of the set.

Throughout this process, the algorithm checks if the current solution can be improved, adjusting it as necessary. If a better solution is found, it is saved. Otherwise, the algorithm may undo recent changes and backtrack. Randomization and occasional vertex swaps help the algorithm explore new parts of the solution space to avoid getting stuck in local optima.

3.3.2 Hypergraph Matching Solvers

Exact Solver

This exact solver by Großmann et al. [13] first applies hypergraph reductions that simplify the problem instance. Then it proceeds with solving the remaining part exactly using Integer Linear Programming (ILP) with Gurobi [14]. The ILP model is constructed to maximize the cumulative weight of the hyperedges that are included in the solution.

Heuristic Solver

The heuristic algorithm by Großmann et al. [13] aims to iteratively improve a matching solution by applying local optimizations and random perturbations. This approach helps overcome local optima. The algorithm needs an a priori solution before running and in our case, we use a greedy algorithm. The initial matching solution after the greedy algorithm is assumed to be in a locally optimal state. However, as this solution may be trapped in a local minimum, the ILS algorithm incorporates a mechanism to overcome this limitation.

The first step is perturbation, where the current solution is randomly altered. This perturbation process randomly selects edges or vertices that are part of the matching and modifies them by either removing or swapping edges. By doing so, the algorithm disturbs the current solution, potentially leading it into unexplored areas of the solution space. This randomness is key to escaping local optima and ensuring that the algorithm doesn't simply get stuck in a suboptimal solution.

Following the perturbation phase, the algorithm engages in a local search utilizing a technique called a one-two swap. This phase is important in order to enhance the perturbed solution and make it more optimal. During the local search, the algorithm identifies edges that could improve the matching by examining the tight constraints on the capacities of the nodes. It then executes a "one-two swap" where two edges are exchanged to create a better matching, ideally elevating the overall quality of the solution. Additionally, the local search ensures that no conflicts occur regarding the capacities of the nodes, thereby complying with the constraints of the matching problem.

After the local search concludes, the algorithm assesses whether to accept the new solution. If the new solution outperforms the previous one, the changes are accepted, and the algorithm proceeds with this updated solution. If the new solution is less favorable, the algorithm utilizes a probabilistic acceptance criterion. This allows for the possibility of accepting an inferior solution with a certain probability, facilitating continued exploration of the solution space. This idea is inspired by Andrade et al. [1]. If the new solution is not accepted, the algorithm reverts to the previous state using a Replay Buffer mechanism, which keeps track of the changes made during perturbations and local searches. This approach ensures that the algorithm can backtrack and restore the last known optimal solution when necessary.

The iterative process of perturbation and local search continues until a stopping criterion is met. The stopping criteria are based on either reaching a maximum number of iterations or exceeding a predefined time limit. Upon completion of the process, the algorithm yields the best solution found during its iterations. This final solution represents the optimal matching obtained through a combination of random exploration and local optimization techniques.

CHAPTER 4

Approaches to find 2-Packing Sets

This thesis explores a method for solving the 2-packing set problem by transforming it into a hypergraph matching problem. At the end of this chapter, we present an alternative approach to solving the 2-packing set problem, by presenting the competing algorithms developed by Borowitz et al. [4].

4.1 Our Approach: Hypergraph Matching with Data Reduction Rules

Our approach addresses the 2-packing set problem by transforming it into a hypergraph matching problem. The first step of the algorithm involves converting the original graph into a hypergraph. To enhance the efficiency of the hypergraph matching process, we employ data reduction techniques. These techniques aim to simplify the hypergraph by iteratively removing vertices and hyperedges while maintaining solution quality. This reductions significantly decrease the size of the hypergraph. Once we have the reduced hypergraph, we apply a hypergraph matching algorithm to identify an optimal matching, thereby yielding a feasible solution for the 2-packing set.

4.1.1 Graph Transformation

In this section, we describe the graph transformation process used in our approach. The objective is to transform the original graph into a suitable hypergraph representation. In our case, each vertex and its direct neighborhood are converted into a hyperedge. We refer to a vertex from which a hyperedge originates as the corresponding vertex. Since each hyperedge represents a vertex and its direct neighborhood, ensuring that no two hyperedges share a vertex guarantees that the corresponding vertices in the original graph are non-adjacent. This creates the necessary separation between vertices, ensuring that the matching meets the distance constraint required for a 2-packing set. This transformation allows the problem of finding 2-packing sets to be addressed using hypergraph matching techniques.

Formally, the transformation can be described as follows: Given the original graph $G = (V, E)$, we construct the transformed hypergraph $H = (V, E_H)$ where V is the set of vertices, and E_H is the multiset of hyperedges. Each hyperedge $e_H \in E_H$ is defined as $e_H = \{v\} \cup N(v)$, where $v \in V$ and $N(v)$ is the neighborhood of v in G. In the following we will show that by transforming the original graph G into a hypergraph H, we can identify a valid 2-packing set through the process of finding a matching in the hypergraph.

First, we want to show that a solution to the 2-packing set problem will correspond to a valid hypergraph matching after the transformation. To do this, we need to confirm that the condition for a hypergraph matching is satisfied, meaning no vertices are shared among any of the hyperedges. Let $S \subseteq V$ be a 2-packing set in G. In a 2-packing set, no two vertices are connected by an edge, and they do not share a common neighbor. Thus, for any distinct vertices $u, v \in S$, we have $\{u, v\} \notin E$ and $N(u) \cap N(v) = \emptyset$. This implies that the hyperedges $e_H(u)$ and $e_H(v)$ do not intersect after the transformation, resulting in $e_H(u) \cap e_H(v) = \emptyset$. Therefore, when we form the set of hyperedges E_H from the vertices of the 2-packing set S, these hyperedges collectively create a valid matching in the hypergraph.

Conversely, let M be a matching in the hypergraph H. Each hyperedge e_H corresponds to a vertex v in the original graph G and its direct neighborhood $N(v)$. We will refer to v as the corresponding vertex of e_H. Since M is a matching, it guarantees that no two hyperedges in M share any vertices. Consequently, when we revert to the graph G, the neighborhood $N(v)$ of each corresponding vertex v is disjoint from the neighborhoods of all other vertices represented by the hyperedges in M. This separation guarantees that the vertices corresponding to the hyperedges in M together constitute a valid 2-packing set. Thus, finding a matching in H directly helps us identify a 2-packing set in G. This effectively enables us to solve the 2-packing set problem via hypergraph matching.

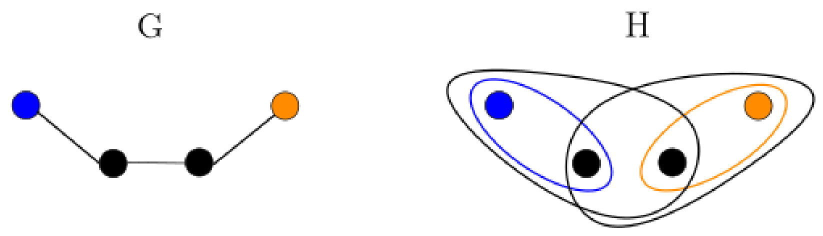

Figure 4.1: Minimal example for our graph transformation. On the left, we see the graph G and on the right, we see the corresponding hypergraph H, that results from the transformation of G. The blue and orange vertices/hyperedges are part of our solution for the 2-packing set/hypergraph matching. The colors also indicate from which vertex a hyperedge is derived from.

4.2 Alternative Approach: Transformation into Maximum Independent Set Problem

In this section, we describe an alternative method for finding 2-packing sets by transforming the problem into a maximum independent set (MIS) problem, as outlined by Borowitz et al. [4]. The core concept involves constructing an auxiliary graph, referred to as *square graph*, where an MIS directly corresponds to a maximum 2-packing set in the original graph. This transformation enables the application of established MIS solvers on the transformed graph to identify optimal solutions.

The transformation adds additional data to the graph by creating a new set of edges that connect vertices which share a common neighbor but are not directly connected. Consequently, this process increases the number of edges in the graph, resulting in a denser structure. This increase in graph density can lead to slower computations and higher memory usage.

To mitigate these drawbacks, Borowitz et al. [4] introduce a pre-processing phase that applies data reduction techniques to the graph. This phase simplifies the graph by removing unnecessary vertices and edges. This leads to a smaller and more manageable kernel graph K. Once the kernel graph K is obtained, the transformation is executed to generate the square graph. Following this, an MIS solver is applied to the square graph to find an optimal maximum independent set. Finally, the solution from the square graph is translated back to the original graph, yielding an optimal 2-packing set.

The strength of this approach lies in its combination of data reduction techniques and the utilization of well-known MIS solvers. By reducing the graph size before the transformation, they handle denser graphs more efficiently, resulting in faster computations and lower memory requirements.

4.3 Data Reduction Rules

We now present the data reduction rules used in our work. We utilize several data reduction rules to simplify the (hyper-)graph and facilitate more efficient processing of the 2-packing set problem. These rules are designed to reduce the size of the (hyper-)graph while preserving the optimality of the solution.

The following sections outline the data reduction rules that we have categorized into three groups. Each group is assigned a unique starting number. The first two groups consist of existing reductions from different frameworks [13, 4]. The first group originates from the competing algorithms by Borowitz et al. [4] and can only be applied on a graph. The second group includes rules developed for direct application on the hypergraph, designed by Großmann et al. [13]. The final group consists of our own reduction rules, mainly inspired by the data reductions from Borowitz et al. [4]. We designed our reduction rules so that they can be directly applied on the hypergraph.

We begin with a summary of the first group of reductions. Full details and proofs for these rules can be found in this paper from Borowitz et al. [4]. Note that these reductions are used exclusively on graphs.

Reduction 1.1 (Domination Reduction)
This rule identifies vertices in G that are dominated by others in their extended 2-neighborhood. If a vertex u is dominated by another vertex v (i.e., all 2-neighbors of v are a subset of the 2-neighbors of u), then u can be removed from the graph without affecting the solution. This reduction simplifies the graph by removing such dominated vertices.

Reduction 1.2 (Clique Reduction)
This reduction applies when a vertex is 2-isolated in its extended 2-neighborhood. If a vertex v is 2-isolated (i.e., all of its 2-neighbors are connected by a path of at most length 2), then v can be safely included in the solution, and its extended 2-neighborhood can be removed from the graph.

Reduction 1.3 (Degree Zero Reduction)
For vertices with degree zero, if their 2-neighborhood is small (i.e., at most one vertex), these vertices can be directly included in the solution. This reduction adds such vertices to the solution and removes their 2-neighborhood from the graph.

Reduction 1.4 (Degree Zero Triangle Reduction)
If a vertex v with degree zero has a 2-neighborhood that forms a triangle, then v can be included in the solution, and its 2-neighborhood can be removed. For a triangle structure, there must be exactly two 2-neighbors of v and they should either be neighbors or 2-neighbors to each other.

Reduction 1.5 (Degree One Reduction)
When a vertex has degree one and its extended 2-neighborhood consists only of its neighbor and his direct neighborhood, it can be included in the solution, and its extended 2-neighborhood can be removed. This reduction ensures that vertices with limited connections are handled efficiently.

Reduction 1.6 (Degree Two V-Shape Reduction)
Vertices with degree two whose 2-neighborhood have a size of zero can be included in the solution, and their extended 2-neighborhood can be removed.

Reduction 1.7 (Degree Two Triangle Reduction)
Similar to the Degree Two V-Shape reduction, if a vertex has degree two and its neighbors are adjacent to each other with no further other neighbors, it can be included in the solution, and its extended 2-neighborhood can be removed.

Reduction 1.8 (Degree Two 4-Cycle Reduction)
If the neighbors of a vertex v with degree two share the same direct neighborhood of size two then we can include v and exclude its extended 2-neighborhood.

Reduction 1.9 (Fast Domination Reduction)
When a vertex is dominated by another vertex and the degree of its extended 2-neighborhood is sufficiently small, the dominated vertex can be removed from the graph.

Reduction 1.10 (Twin Reduction)
For vertices with degree two that have twin neighbors (neighbors with identical neighborhoods), the vertex can be included in the solution, and its 2-neighborhood can be removed.

All of the data reduction rules presented from this point onward are exclusively used on the hypergraph. The following data reduction rules are from the b-matching framework by Großmann et al. [13] that we used. For further information and proofs, refer to their paper [13]. It is important to note that, in our specific case, both the capacity of each vertex in the hypergraph is always 1, and the weight of each hyperedge starts with 1.

Reduction 2.1 (Abundant Vertices)
A vertex v is called abundant if its capacity is equal to or greater than its degree. Such a vertex can be removed from the hypergraph without affecting the optimality of the solution. Any hyperedge incident to v that becomes empty as a result of this removal can be included in the optimal solution.

Reduction 2.2 (Neighborhood Removal)
A hyperedge $e \in E_H$ can be part of an optimal matching if its weight is greater than or equal to the sum of the weights of the heaviest hyperedge incident to each vertex v of e. If this condition holds, then e should be included in the optimal matching.

Reduction 2.3 (Weighted Isolated Edge Removal)
A hyperedge $e \in E_H$ with the heaviest weight among its neighbors, that is $\omega(e) \geq max_{f \in N(e)} \omega(f))$, such that every neighbor f of e shares a common vertex v, is guaranteed to be part of an optimal solution.

Reduction 2.4 (Weighted Edge Folding)
Let $e \in E_H$ be a hyperedge with neighbors $N = N(e) \setminus \{e\}$. N must be independent, meaning that no distinct hyperedges in N are adjacent to each other. If $\omega(e) > \omega(N) - \min_{f \in N} \omega(f)$ holds, then e and its neighbors N can be merged into a new edge e' with weight $\omega(e') := \omega(N) - \omega(e)$. In the end, whether N or e is contained in an optimal matching is dependent on whether e' is in an optimal matching M'_{opt} or not.

Reduction 2.5 (Weighted Twin)
If we have two non-adjacent hyperedges $e_1, e_2 \in E_H$ with identical independent sets of neighbors $L_1 = L_2$ and if $\omega(e_1) + \omega(e_2) > \omega(L_1) - \min_{f \in L_1} \omega(f)$, then e_1 and e_2 can be replaced by a new edge e' with weight $\omega(e_1) + \omega(e_2)$.

Reduction 2.6 (Weighted Domination)
If $e \in E_H$ is a subset of $f \in E_H$ and $\omega(e) \geq \omega(f)$, then f can be removed from the hypergraph. This is because e can replace f in any optimal solution, as e has a weight equal to or greater than f and both cannot be in the maximum matching simultaneously.

We now present the last group of data reduction rules. The idea behind them is mostly derived from Borowitz et al. [4]. Note that their reduction rules are designed for graphs and that we reformulated some of their reduction rules, so that they can be directly applied on the hypergraph. As we have implemented them ourselves, we will also provide proofs for them. Their names are derived from the reduction rules they are inspired from.

Reduction 3.1 (Degree One - Hypergraph Version)
Let $v \in V$ be a vertex in the hypergraph with degree 2. Furthermore, let $e \in E_H$ be an incident hyperedge of v of size 2. Then, e is in an optimal matching.

Proof. Suppose $e, f \in E_H$ are the hyperedges incident to vertex v. Assume that e is a hyperedge of size 2, containing the vertices $\{u, v\}$, which results from the transformation of the direct neighborhood of v in the original graph. Since we transformed from an undirected graph, there must exist another hyperedge containing at least {u,v} which was transformed from u and its neighborhood. This leads to the conclusion that e is a subset of f and for any optimal solution M_{opt} we can replace f with e due to Reduction 2.6.

Reduction 3.2 (Degree Two V-Shape - Hypergraph Version)
Let e be a hyperedge of size 3. Furthermore, assume that $f, g \in E_H$ are the only two adjacent hyperedges to e of size 2 and subsets of e. Then e is in an optimal matching.

Proof. Assume the above stated assumptions. As f, g are subsets of e of size 2, we know that all three hyperedges must have at least one vertice they are all incident to. Thus, we can apply Reduction 2.3 and add e to the solution.

Reduction 3.3 (Degree Two Triangle - Hypergraph Version)
Let e, f, g be hyperedges of size 3 containing the same set of vertices meaning they are identical. Furthermore, assume that there are no other adjacent edges to any of them. Then e is in an optimal matching.

Proof. If e, f, g contain the same set of vertices then we can say that e is a subset of f and of g. Thus, we can apply Reduction 2.6, add e to the matching and remove f, g from the hypergraph.

Reduction 3.4 (Degree Two 4-Cycle - Hypergraph Version)
Let $e, f, g, h \in E_H$ be hyperedges and let e, f, h have size 3. Furthermore, let f, h be the only adjacent hyperedges to e and similarily let g, e be the only adjacent hyperedges to both f and h. Then e is in some optimal matching.

Proof. Assume the above stated assumptions. Then we know that each adjacent hyperedges share at least two common vertices. This is due to the fact that a hyperedge always depicts the direct neighborhood of a vertex in the original graph. Naturally, the vertex then also has to be present in the hyperedges derived from its neighboring vertices. Because e only has a size of three, this means that there must exist a common vertex that e, f and h share. Then it holds that we can apply Reduction 2.3 on the hyperedge e. Assume g is part of an optimal matching. Then, we can create a new solution $M'_{opt} = M_{opt} \setminus \{g\} \cup \{e\}$ of same size. This way we always find an optimal matching including e.

Reduction 3.5 (Twin - Hypergraph Version)
Let $e \in E_H$ be a hyperedge of size 3 and f, g its only adjacent hyperedges, both of them containing the same set of vertices. Then e is in some optimal matching.

Proof. This is another case where the hyperedges share a common vertex and therefore we can apply Reduction 2.3 and add e to the matching.

In the following, we added some figures, visualizing some examples for the reductions. In each figure, the orange hyperedge indicates the edge that will be added to the matching through the reduction. The orange vertices are the corresponding vertices which the orange hyperedges were derived from. We did not add an additional figure for Reduction 3.1 as Figure 4.1 already serves as a suitable visualization for this reduction. Figure 4.2a and 4.2b show examples for the Reductions 3.2 and 3.3. Note that these two reductions only work when the hyperedges have exactly the structure as depicted in the figures. In Figure 4.3a and 4.3b, we see examples for Reductions 3.4 and 3.5. Note that in Figure 4.3a, g can be adjacent to more hyperedges than depicted. This subsequently means that the hyperedge is not constrained to the size of 3 like the other hyperedges in this figure. Furthermore, in Figure 4.3b only the three hyperedges that play a role for our reduction are depicted.

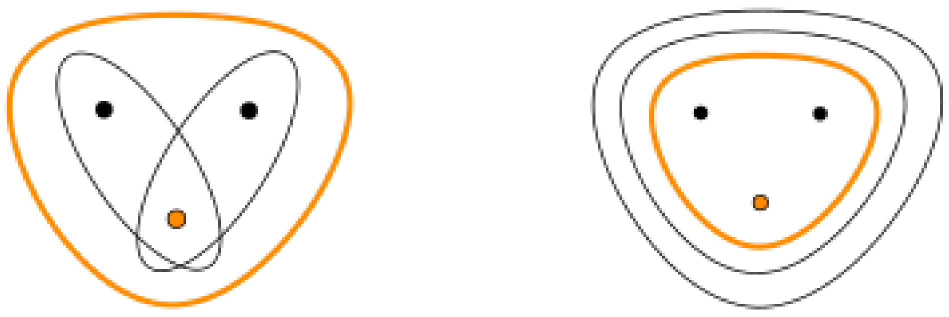
(a) Degree Two V-Shape **(b)** Degree Two Triangle

Figure 4.2: Hypergraph Reductions 3.2 and 3.3.

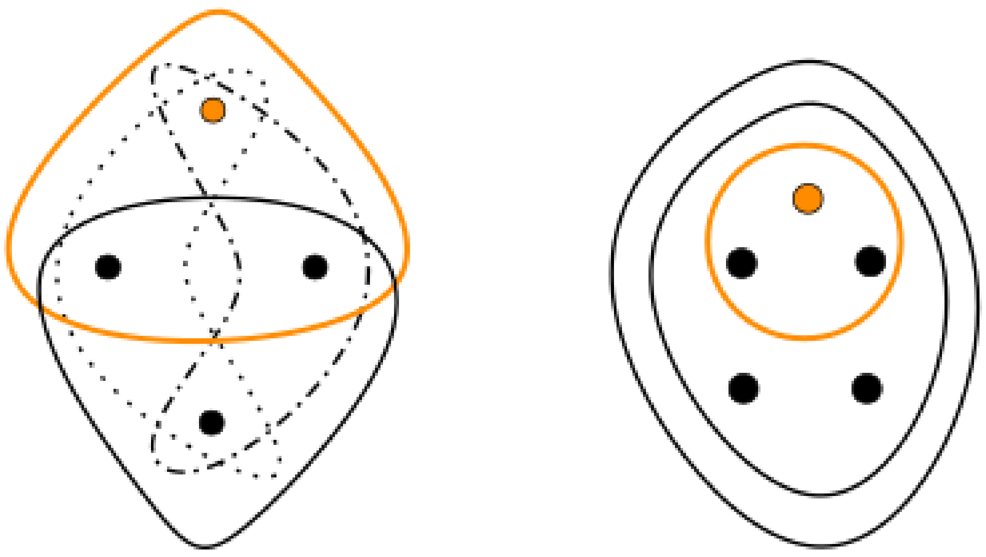
(a) Degree Two 4-Cycle **(b)** Twin

Figure 4.3: Hypergraph Reductions 3.4 and 3.5.

CHAPTER 5

Experimental Evaluation

5.1 Methodology

Our algorithms were implemented in C++ and compiled using g++ 13.1.0 with optimization level O3. Computations were performed on a machine equipped with an Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz processor (16 cores) and 92 GB RAM. The operating system is Ubuntu 22.04.1 LTS and the kernel version is 5.4.0-169-generic. In our experiments, each instance was run five times per algorithm. For the heuristic algorithms a different seed was chosen for each run on the same instance. We also use Gurobi Optimizer [14] 11.0.3 for our exact hypergraph matching solver. A time limit of 1 000 seconds was set for all algorithms.

We compare the algorithms mainly using performance profiles introduced by Dolan and Moré [8]. These profiles can assess both running time and solution size. We explain here how the performance profile works for running time, as the principle is similar for solution size. Performance profiles for running time visualize how algorithms compare in terms of their running times across various instances. On the y-axis, we represent the fraction of instances in which an algorithm's running time is at most τ times the fastest running time observed for that instance. τ, which is equal to or greater than 1, is plotted on the x-axis. Each algorithm's performance profile forms a step-like function incrementally increasing or remaining constant as τ increases. The performance profiles help us understand how different algorithms perform relative to each other across different problem instances. For size performance profiles, we show the fraction of instances where an algorithm's solution size is greater than or equal to τ times the best solution size on an instance among all the algorithms. Note that for solution size performance profiles, τ starts from 1 and gradually decreases.

We evaluate the performance of the reductions and the overall memory usage using bar charts. In both cases, we compute the geometric mean across all instances for each algorithm. Since some hypergraphs could not be reduced, only instances with at least one successful reduction were included in the geometric mean calculations.

5.2 Datasets

As test instances we have chosen 30 social graphs from here [3]. In addition to that, we added a total of 20 outerplanar graphs as test instances used by Trejo-Sánchez et al. in their paper [24]. For further informations on the graphs refer to Table 5.1 and 5.2. In our experiments, we seperately evaluate the social and the outerplanar graphs, as most algorithms seem to struggle with effectively reducing and solving the outerplanar graphs.

Table 5.1: Social graph instances

Name	n	m
chesapeake	39	170
dolphins	62	159
lesmis	77	254
polbooks	105	441
adjnoun	112	425
football	115	613
celegansneural	297	2 148
celegans_metabolic	453	2025
email	1 133	5 451
netscience	1 589	2 742
power	4 941	6 594
hep-th	8 361	15 751
PGPgiantcompo	10 680	24 316
astro-ph	16 706	121 251
cond-mat	16 726	47 594
as-22july06	22 963	48 436
cond-mat-2003	31 163	120 029
cond-mat-2005	40 421	175 691
preferentialAttachment	100 000	499 985
smallworld	100 000	499 998
G_n_pin_pout	100 000	501 198
caidaRouterLevel	192 244	609 066
coAuthorsCiteseer	227 320	814 134
citationCiteseer	268 495	1 156 647
coAuthorsDBLP	299 067	977 676
cnr-2000	325 557	2 738 969
coPapersCiteseer	434 102	16 036 720
coPapersDBLP	540 486	15 245 729
road_central	14 081 816	16 933 413
road_usa	23 947 347	28 854 312

Table 5.2: Outerplanar graph instances

Name	n	m
Outerplanar500_1	62 320	65 138
Outerplanar500_2	73 959	76 976
Outerplanar1000_1	148 564	154 609
Outerplanar1000_2	151 091	157 100
Outerplanar1500_1	227 107	236 077
Outerplanar1500_2	226 090	235 100
Outerplanar2000_1	301 431	313 433
Outerplanar2000_2	301 692	313 729
Outerplanar2500_1	375 728	390 874
Outerplanar2500_2	373 931	389 003
Outerplanar3000_1	448 689	466 782
Outerplanar3000_2	451 224	469 413
Outerplanar3500_1	523 959	545 122
Outerplanar3500_2	529 022	550 144
Outerplanar4000_1	600 173	624 188
Outerplanar4000_2	600 288	624 264
Outerplanar4500_1	675 339	702 423
Outerplanar4500_2	677 075	704 222
Outerplanar5000_1	748 383	778 411
Outerplanar5000_2	750 308	780 191

Reduction ID	**Name**	**Short Name**
1.1-1.10	red2pack reductions	R2P
2.1	Abundant Vertices	AV
2.2	Neighborhood Removal	NR
2.3	Weighted Isolated Edge Removal	WIER
2.4	Weighted Edge Folding	WEF
2.5	Weighted Twin	WT
2.6	Weighted Domination	WD
3.1	Degree One	DEG1
3.2	Degree Two V-Shape	DEG2-V
3.3	Degree Two Triangle	DEG2-TRI
3.4	Degree Two 4-Cycle	DEG2-CYCLE
3.5	Twin	TWIN

Table 5.3: List of short names for reduction rules used in subsequent figures

5.3 Configuration selection

To begin with, we want to clarify that for all algorithms, we used the best-performing parameter settings as specified in their respective papers [4, 13]. The primary objective of this work is to evaluate how effective our matching algorithms and reductions perform in comparison to the competing framework `red2pack` by Borowitz et al. [4]. To achieve this, we propose three distinct reduction variants using different sets of reductions for our matching algorithms. We name the reduction variants `bm`, `r2p`, and `r2p_bm`.

The first reduction variant, `bm`, serves as the baseline for our comparison and employs Reductions 2.1–2.6, which are applied directly to the hypergraph. It's important to note that both of the other reduction variants also include these reductions. For the second reduction variant `r2p`, we first use the reductions from `red2pack` on the original graph. Afterwards, we transform the kernel graph into a hypergraph and apply Reductions 2.1–2.6. Lastly, the third reduction variant, `r2p_bm`, combines Reductions 2.1–2.6 and Reductions 3.1–3.5, all of which are performed on the hypergraph. This setup enables us to compare the efficiency of our new reductions against both the `red2pack` reductions and the baseline reductions from the matching framework that we use by Großmann et al. [4]. In Table 5.3, we provide an overview of the short names for each reduction rule that are used in the subsequent figures.

5.4 Comparison of the Efficiency of the Reductions

In this section, we analyze the efficiency of the reductions. We do this by comparing the geometric mean of the number of excluded hyperedges and vertices for each reduction. Reductions that exclude hyperedges are colored blue, while the only reduction in our framework that excludes mainly vertices, `AV`, is colored orange. Note that `AV` is also able to exclude hyperedges if the removal of a vertex leaves an empty hyperedge behind though this is not depicted in our charts. Instances where no successful reductions were applied are excluded from the geometric mean calculations.

Among the 30 social graph instances, the number of graphs that could not be reduced by any of the reduction variants ranges from two to four. For the 20 outerplanar graphs, only the `r2p` reduction variant was able to reduce all instances to some extent. The other reduction variants did not achieve any successful reductions on the outerplanar graphs. In the following, we only provide charts for the number of excluded vertices/hyperedges per reduction. We intentionally did not include any visualizations on running time, as most of the reductions only needed at most a few (milli-)seconds, generally much less.

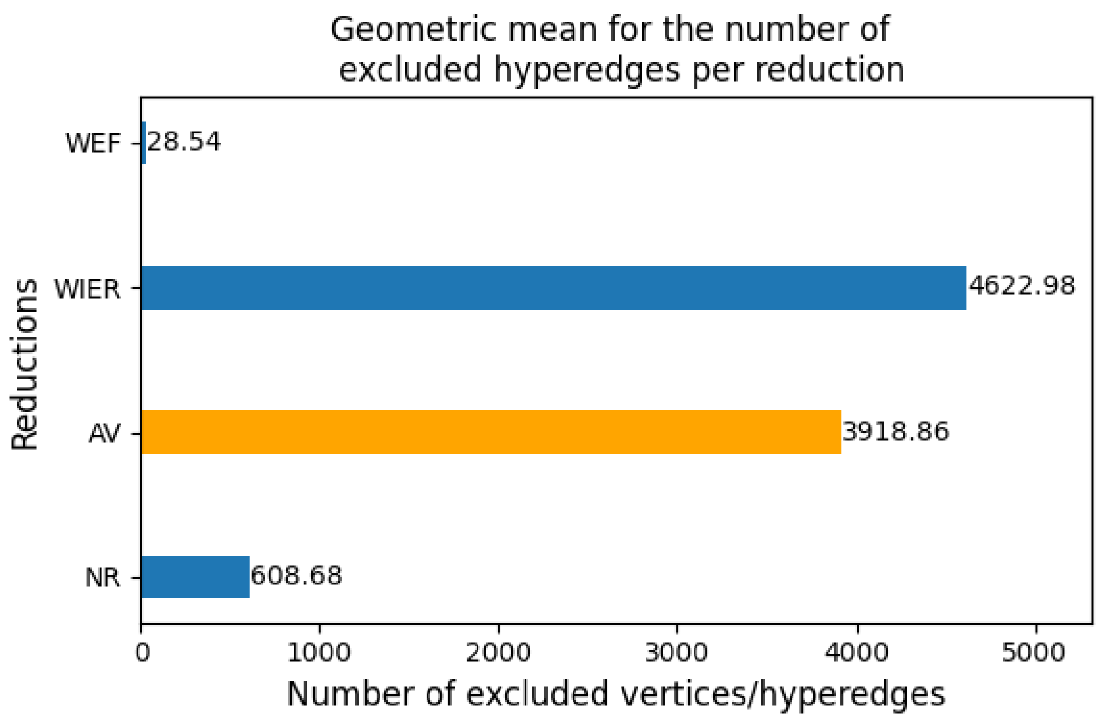

Figure 5.1: Variant `bm` on social graphs

In Figure 5.1, we see the results of the `bm` reduction variant on social graphs. The most effective reductions are the Weighted Isolated Edge Removal and the Abundant Vertices reduction, which exclude an average of 4 623 hyperedges and 3 919 vertices, respectively. The Neighborhood Removal reduction is also effective, though its impact is significantly smaller, averaging around 609 excluded hyperedges.

The least effective reduction in this variant is the Weighted Edge Folding, which removes only about 29 hyperedges. Notably, two out of the six employed reduction rules were not used at all: Weighted Domination and Weighted Twin. On one hand, this could mean that there were no suitable opportunities for these reduction rules to be applicable. Alternatively, the parameter constraints for these reduction rules could have been exceeded. According to the paper by Großmann et al. [4], the Weighted Domination rule only considers (sub-)edges up to a size of six. For the Weighted Twin, the edge size needs to be at most four to be considered.

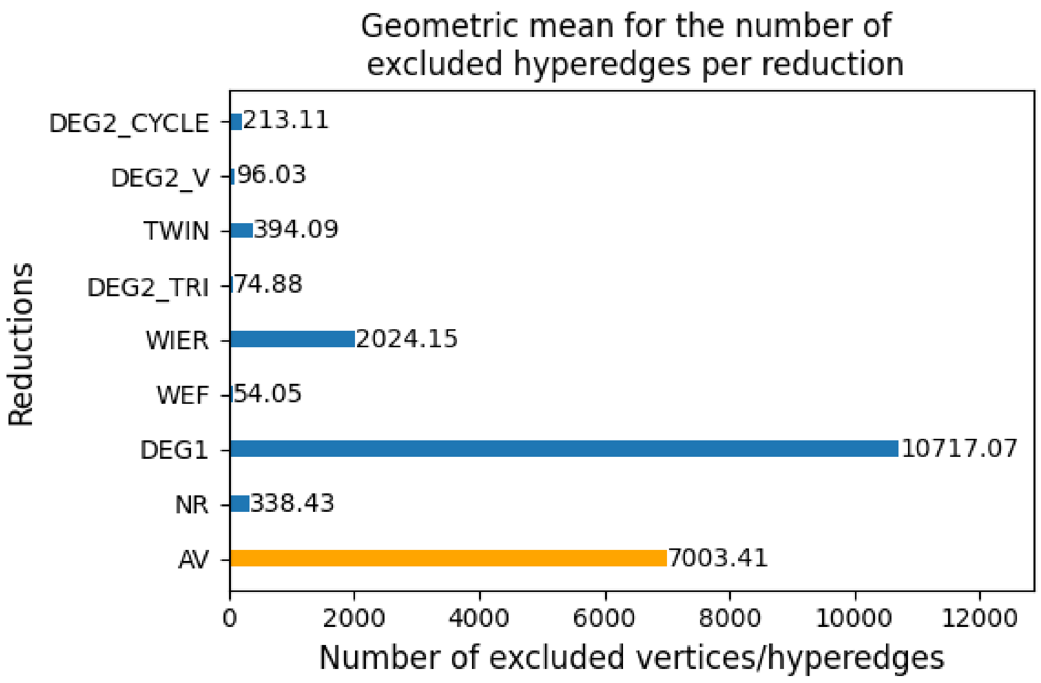

Figure 5.2: Variant `r2p_bm` on social graphs

For the `r2p_bm` reduction variant on social graphs (see Figure 5.2), the most effective reduction is our self-implemented reduction, `DEG1`, which excludes more than 10,000 hyperedges. This is followed by the Abundant Vertices reduction (`AV`), which excludes 7,108 vertices, and the Weighted Isolated Edge Removal (`WIER`), which excludes 2,024 hyperedges. The other reductions perform significantly worse, with the number of excluded hyperedges ranging only from 54 to 394. Similarly to the `bm` reduction variant, the reductions Weighted Domination and Weighted Twin were not used here either. Compared to the `bm` reduction variant, `r2p_bm` is considerably more effective in reducing the graph, as `DEG1` is able to exclude more than double the number of hyperedges compared to the other reductions on average. Furthermore, the use of the additional reductions also increased the average number of excluded vertices through `AV` by a factor of around 1,78. In comparison, the number of excluded hyperedges for `WIER` has decreased by a factor of nearly 2.

It is unsurprising that the reductions `DEG2_V` and `DEG2_TRI` don't perform so well. As we have mentioned earlier they are only applicable if the exact number of hyperedges and hyperedge sizes is fulfilled which makes them very inflexible. We did expect `DEG1` to be much more effective in comparison as it has fewer strict requirements. Even so, we are also surprised by its overwhelming performance.

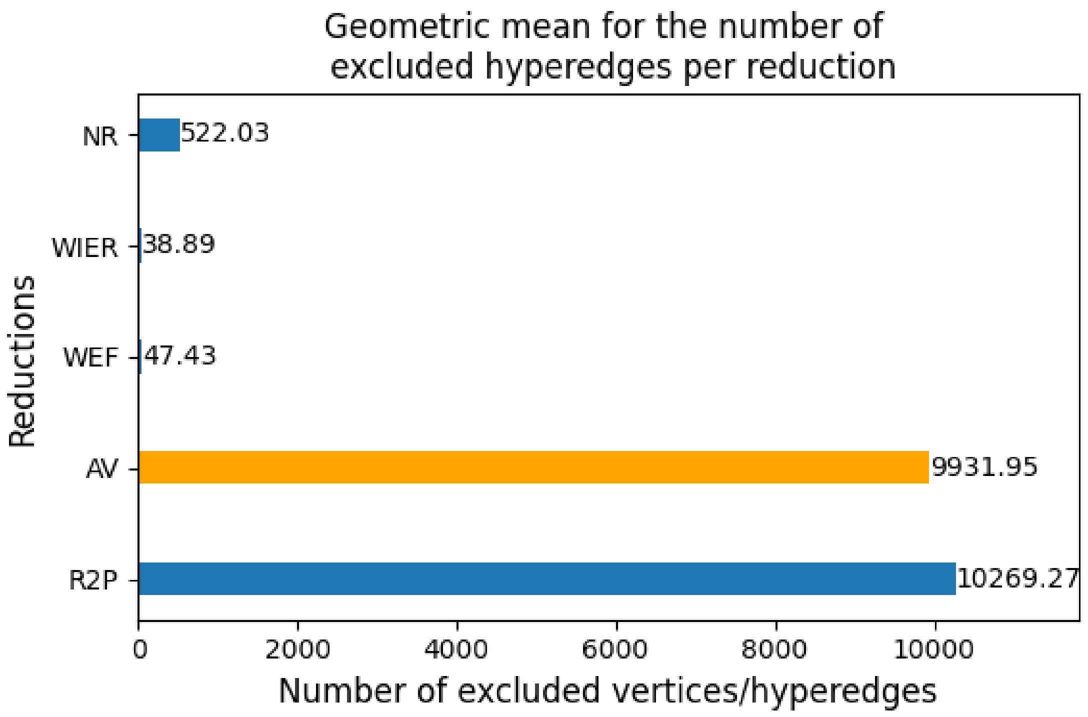

Figure 5.3: Variant `r2p` on social graphs

In Figure 5.3, for the `r2p` reduction variant on social graphs, we observe that the reductions from `red2pack` are very effective with over 10 000 excluded hyperedges. Nevertheless, the reduction Abundant Vertices from the matching framework is almost equally effective, excluding approximately 9 932 vertices. A large gap follows, then comes the reduction rule Neighborhood Removal with 522 excluded hyperedges. Furthermore, the rest of the reductions from our matching framework contribute minimally with fewer than 50 excluded hyperedges on average. In general, this reduction variant also seems to be more effective in reducing the hypergraph than `bm`. If we compare with the `r2p_bm` reduction variant, we see that more vertices are removed with this variant but fewer hyperedges overall.

Lastly, in Figure 5.4, we examine the outerplanar graphs using the `r2p` reduction variant. We see that the `red2pack` reductions exclude approximately 310 hyperedges, while Abundant Vertices only excludes about 20 hyperedges on average. The results on the outerplanar graphs suggest that the reduction rules are significantly less effective on this graph class in general. This is underlined by the fact that no other reduction variant could be applied on these instances. This suggests that outerplanar graphs are generally difficult to reduce.

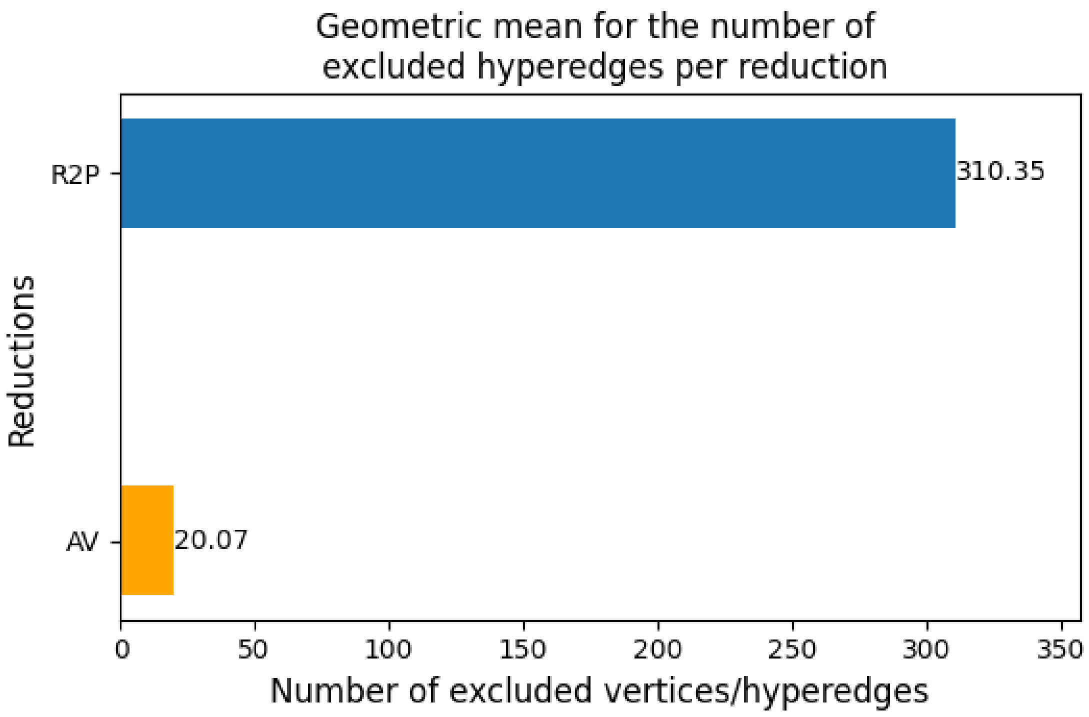

Figure 5.4: Variant `r2p` on outerplanar graphs

Overall, the results indicate that `r2p_bm` and `r2p` are indeed an improvement compared to the base reduction variant `bm`. Especially the reductions `DEG1` and `AV` seem to be able to exclude a high number of hyperedges similar to the number of reductions from `red2pack`. It is also interesting to see that only the `red2pack` reductions are able to reduce the outerplanar graphs slightly. Except for `AV`, no further reductions could be applied to the outerplanar graph at all. This may also be due to the parameter constraints on our reductions. If all of the hyperedges are very large in the outerplanar graphs, then it could be the reason why none of our reductions were applied. This is also our hypothesis as to why the reduction rules Weighted Domination and Weighted Twin weren't used at all for any instance.

Another interesting observation is that some of the social graphs could be completely reduced just by using the reductions. Among the reduction variants, `r2p` achieved the highest number of completely reduced instances. It reduced 13 out of 30 social graphs. Meanwhile, `r2p_bm` was only able to reduce 3 graphs completely and `bm` only 1. In conclusion, `r2p` demonstrated the best overall performance among the reduction variants that we have presented, followed by `r2p_bm` and then `bm`. In general, the reduction variants are effective in reducing the hypergraph size, particularly on social graphs.

5.5 Comparison of the Exact Algorithms

We proceed by comparing the overall running time and solution size of the exact algorithms from the matching framework and the competitor `red2pack`. For the exact matching algorithm, we use the short name `pre_ilp`. The branch-and-reduce algorithm from `red2pack` is denoted by `r2p[bnr]`. Even though we used exact algorithms, many instances couldn't be solved completely before the time limit, especially for the outerplanar graphs. Therefore, we also give performance profiles on solution size for the exact algorithms. If the time limit was exceeded, we used the solution size at the time of the timeout to compare the solution size in the performance profile. For instances where the time limit was not reached, all of the algorithms provided the same solution sizes, which was to be expected. Nevertheless, these solutions are also visualized in the performance profile.

Figure 5.5 shows the performance profile of the solution size for the social graphs, while Figure 5.7 illustrates the same for the outerplanar graphs. Figure 5.6 depicts the performance profile for running time on social graphs. For the outerplanar instances, all exact algorithms reached the maximum time limit of 1 000 seconds. This is why no meaningful distinction can be observed in the running time performance profile. Therefore, we omitted the performance profile on running time for the outerplanar graphs.

5.5.1 Solution Size for Social Graphs (Exact)

Out of 30 social graphs, around 5 to 6 instances couldn't be solved within the time limit. Looking at Figure 5.5, we see that for social graphs, the `red2pack` branch-and-reduce algorithm provides the best solution size across all instances. More than 10% of our instances only achieved 90% of the solution size of `r2p[bnr]`. Even worse, around 5% of the instances could only achieve half of the solution size of `r2p[bnr]`. This suggests that given a time limit, `red2pack` seems to work faster than our algorithm on social graphs, thus achieving a better solution when reaching the time limit.

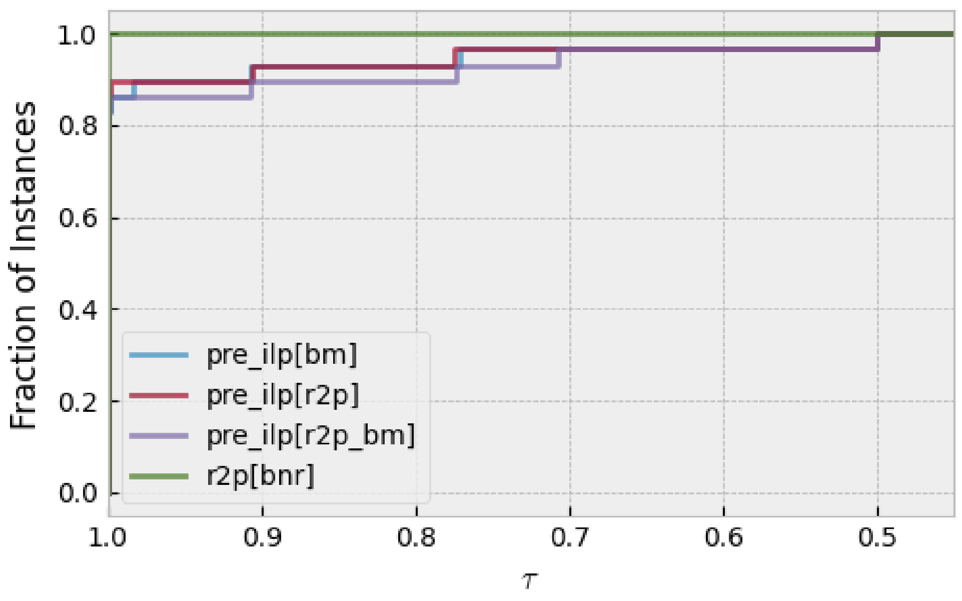

Figure 5.5: Solution size for social graphs

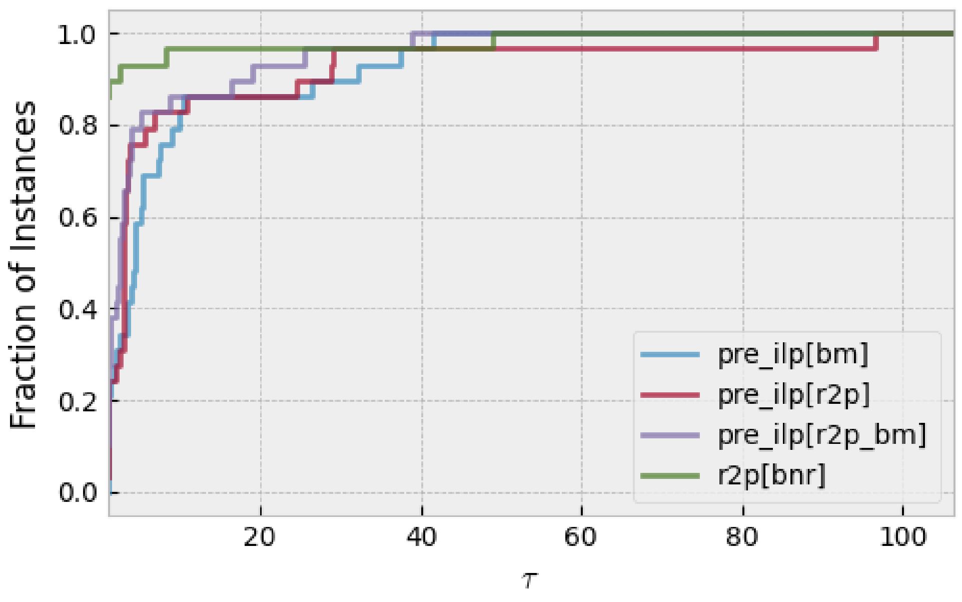

Figure 5.6: Running time for social graphs

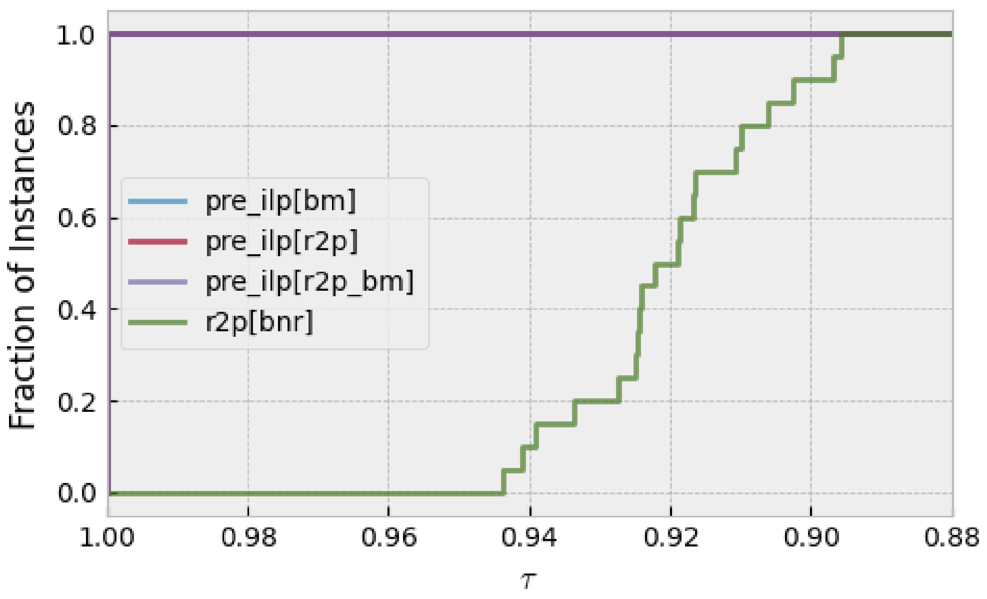

Figure 5.7: Solution size for outerplanar graphs

5.5.2 Running Time for Social Graphs (Exact)

It is important to note that for social graphs the running time was generally low. Most of the instances could be solved within a few milliseconds with all algorithms. There are only a few exceptions that required more than a few seconds or couldn't be solved within the time limit. Figure 5.6 shows that almost 90% of the instances were solved the fastest by `red2pack` on social graphs. As for our variants `pre_ilp[r2p_bm]` performs slightly better for a small percentage of instances compared to the other two reduction variants. In the best case, `pre_ilp[r2p_bm]` and `pre_ilp[bm]` need at least 40 times longer than the fastest running algorithm for all instances. The worst scaling factor has `pre_ilp[r2p]` with a value of over 95.

5.5.3 Solution Size for Outerplanar Graphs (Exact)

Turning to the outerplanar graphs in Figure 5.7, the situation is somewhat reversed. It is important to note that out of 20 outerplanar graphs, all algorithms exceeded the given time limit. Nevertheless, our matching algorithm provided the best solution sizes across all outerplanar instances. The solution sizes of `r2p[bnr]` range between 89,5% and 94,5% of the best solution size.

5.5.4 Summary (Exact)

Overall, the performance profiles indicate that red2pack offers superior effectiveness in both solution size and running time on social graphs. However, on outerplanar graphs, our variants provide slightly better solution sizes across the instances. This suggests that the effectiveness of the algorithms can be influenced by the graph class being considered. Furthermore, we can say that the different reduction variants don't seem to differ much in terms of running time and solution size as of right now. The only slightly notable difference was seen in the performance profile on running time for social graphs where the variant r2p_bm performed slightly better than the others. Nevertheless, we need to remember that the social graphs instances were solved quite fast for all algorithms. As a consequence, a time difference of only a second could significantly impact the performance profile. This is why the minor differences between the reduction variants seen in the running time performance profile don't necessarily tell us much.

5.6 Comparison of the Heuristic Algorithms

In this section, we compare the performance of our heuristic algorithm ils, against the red2pack heuristic r2p[h]. We focus on both the solution size and running time for social and outerplanar graphs. The results are presented in Figures 5.8 –5.11.

5.6.1 Solution Size for Social Graphs (Heuristic)

Figure 5.8 shows the performance profile for solution size on social graphs. Initially, the red2pack heuristic algorithm provides the best solution sizes for more than 90% of the instances. In comparison, our reduction variants only reach around 94% of the best solution size for 90% of the instances. We achieve the best solution quality on all instances with our reduction variants at around 88% of the best solution size. Meanwhile, r2p[h] achieves only 82% of the best solution size in at least one instance.

5.6.2 Running Time for Social Graphs (Heuristic)

Figure 5.9 presents the running time performance profile for the social graphs. The red2pack algorithm initially demonstrates superior efficiency since more than 30% of the instances could be solved faster than the other algorithms. Nevertheless, our variants show a steeper curve for increasing τ. When we scale the best running times with a factor of 3, the fraction of instances of our variants, except for ils[bm], is actually higher than that of r2p[h]. In the best case, ils[r2p_bm] only needs roughly 7 times longer than the fastest algorithm to solve all instances. In the worst case, the algorithms ils[r2p] and r2p[h] need almost 100 times longer than the best algorithm.

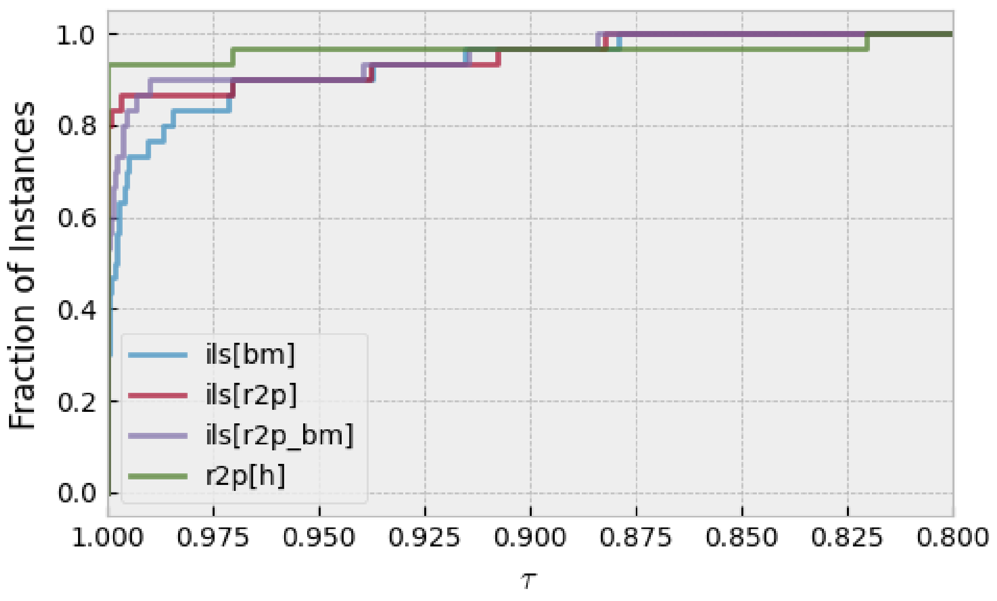

Figure 5.8: Solution size for social graphs

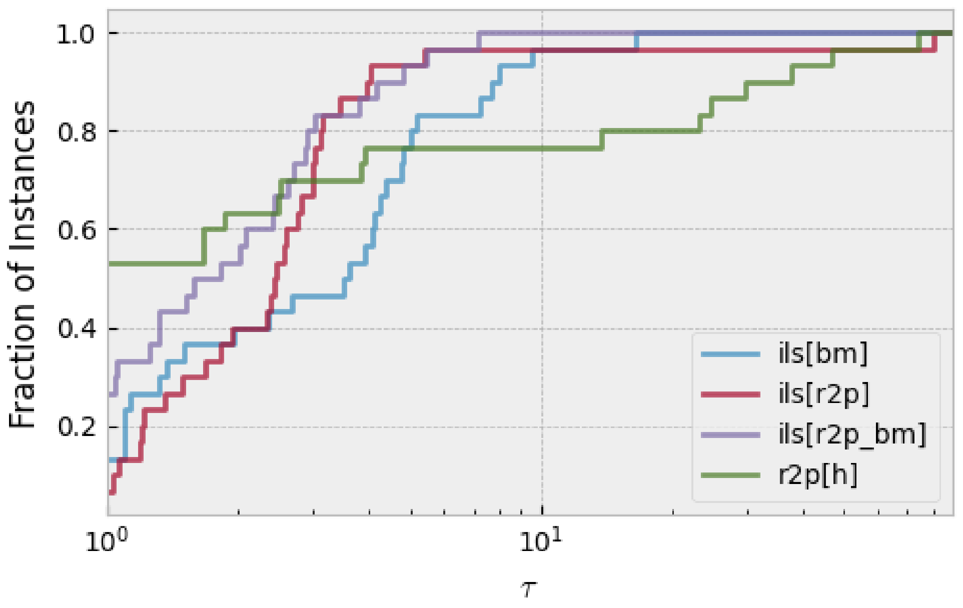

Figure 5.9: Running time for social graphs

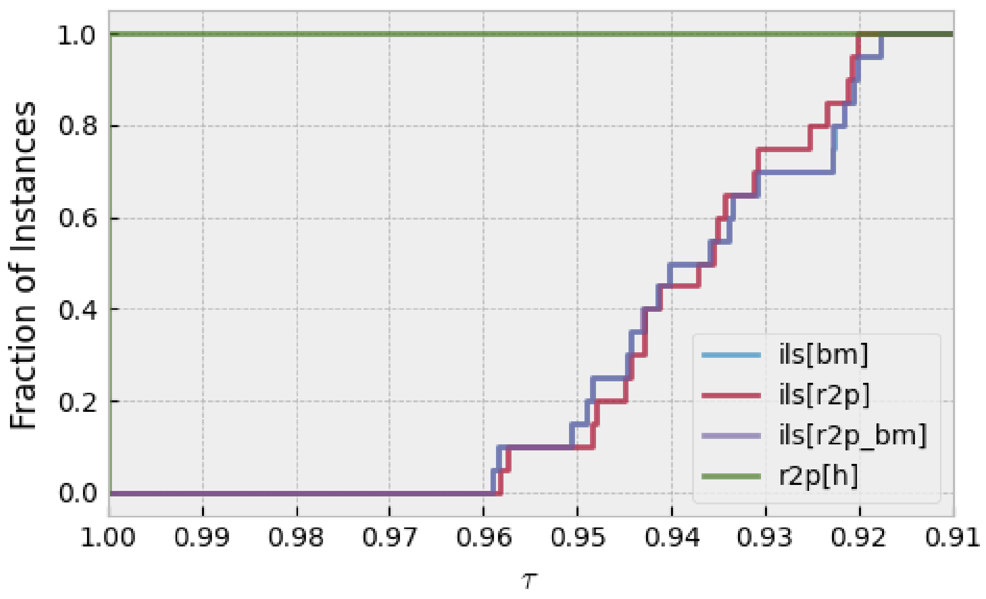

Figure 5.10: Solution size for outerplanar graphs

5.6.3 Solution Size for Outerplanar Graphs (Heuristic)

Moving on to the outerplanar graphs, Figure 5.10 illustrates the performance profile for solution size. Here, we see that `red2pack` performs better across all instances, compared to our algorithms. We achieve only solution sizes of approximately 92-96% of the best solution sizes.

5.6.4 Running Time for Outerplanar Graphs (Heuristic)

Finally, Figure 5.11 shows the running time performance for outerplanar graphs. In this case, the results are surprisingly good for our algorithms. All of our reduction variants are several orders of magnitude faster than `red2pack`. The `ils[bm]` reduction variant delivers the fastest performance, followed closely by `ils[r2p_bm]` and then `ils[r2p]`. Most of the instances are solved within twice the the best running time for all our variants. In contrast, `red2pack` takes at least 100 times longer than the best running time. Some of the instances of `r2p[h]` even need a factor of over 1 000 times more running time.

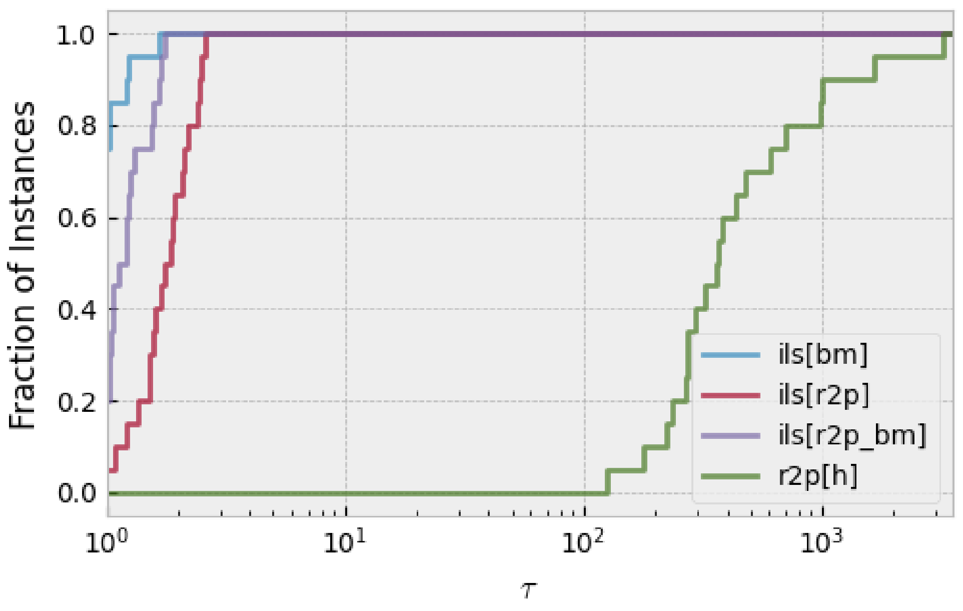

Figure 5.11: Running time for outerplanar graphs

5.6.5 Summary (Heuristic)

From the performance profiles, we can draw the conclusion that our heuristic algorithms generally offer faster solutions than `red2pack` on both social and especially on outerplanar graphs for most instances. Still, it is important to note that `red2pack` tends to provide better solution sizes. To be more precise, our heuristic algorithm provides solutions that achieve sizes of at least 88% of the best solution size. Furthermore, the running time performance of our algorithm is strong. This applies especially to large or complex graph instances like the outerplanar graphs, where `red2pack` falls behind significantly. Therefore, our heuristic algorithm is possibly better suited for faster execution on larger and more complex graphs. On the other hand, `red2pack` offers better solution sizes on most instances but at the cost of a much longer running time.

5.7 Memory Usage

We will now provide an overview of the memory usage. Note that we didn't separately evaluate the memory usage for the different graph classes (outerplanar and social). When comparing the memory usage of the algorithms, our reduction variants `ils[bm]` and `ils[r2p_bm]` perform the best, using the least memory overall with 61 595 and 62 660 kBytes respectively. Interestingly, the heuristic algorithm from `red2pack` closely follows these reduction variants with around 64 602 kBytes, which is somewhat surprising. Slightly higher memory consumption is observed for `ils[r2p]`, though it remains relatively efficient in comparison. In contrast, the `pre_ilp` algorithm requires about five to six times as much memory as the algorithm with the lowest memory usage. However, the worst memory consumption is seen in the `bnr` algorithm from `red2pack`. It uses more than ten times the memory of our most memory-efficient algorithm. The high memory usage by `r2p[bnr]` was expected, as the generation of the square graph creates a fair number of additional edges, which all require storage. Borowitz et al. [4] also stated in their paper that the `r2p[bnr]` algorithm had memory usage issues when applied to outerplanar graphs. We actually anticipated similar memory behavior from the `red2pack` heuristic algorithm, but it turned out to be much more memory-efficient than expected.

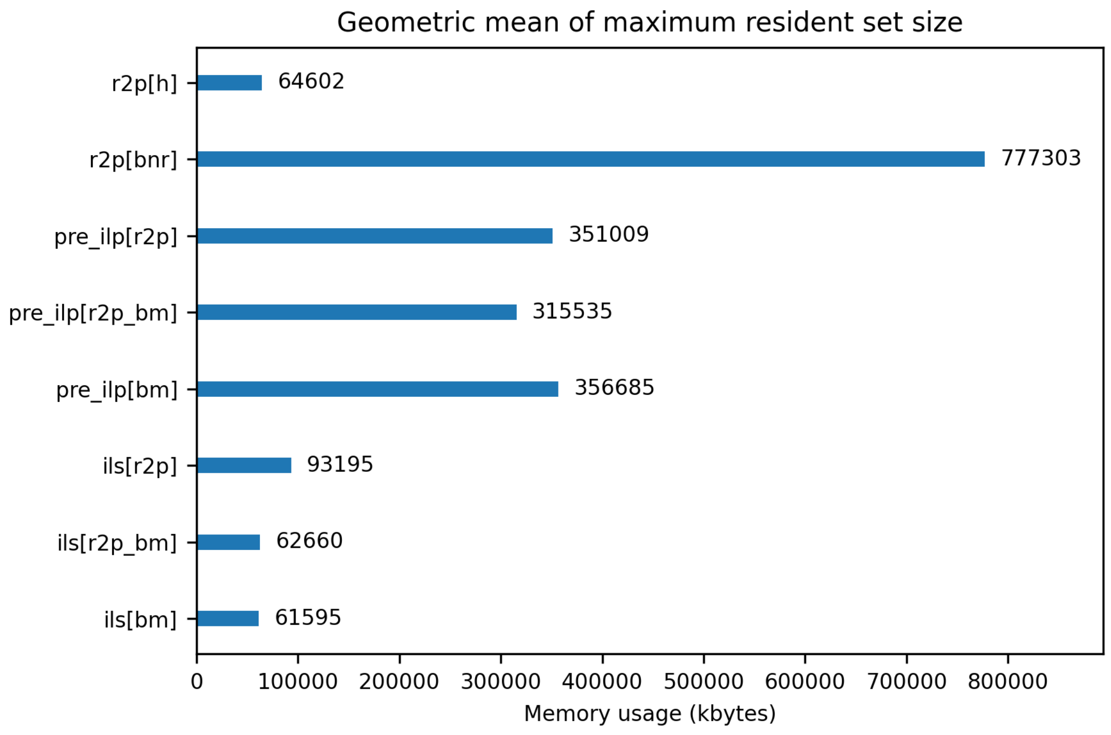

Figure 5.12: Geometric mean of maximum resident set size

CHAPTER 6

Discussion

6.1 Conclusion

In conclusion, the evaluation of our reduction variants demonstrates that both `r2p` and `r2p_bm` provide enhanced reduction efficacy over `bm`. However, it should be noted that only 1-2 reduction rules contributed significantly to the overall reduction of the (hyper-)graph. Additionally, only the `red2pack` reductions successfully reduced the outerplanar graphs, while our methods contributed minimal reductions, highlighting potential areas for further improvement in hypergraph reduction techniques.

Examining the performance profiles, we observe that the reduction variants generally performed similarly, suggesting that the specific choice of reductions had a limited impact on overall performance. When comparing the performance of our algorithms with `red2pack`, the exact implementation of `red2pack` clearly outperformed ours in running time. Only for the outerplanar graphs, where all exact algorithms exceeded the time limit, could we provide better solution sizes with our exact algorithm than `red2pack`. In contrast, our heuristic algorithm `ils` demonstrated significantly faster running times. Despite this speed advantage, the `ils` solutions sometimes reached only 88% of the optimal solution size achieved by `red2pack`. Therefore, our heuristic algorithm is possibly better suited for faster execution on larger and more complex graphs. On the other hand, `red2pack` offers better solution sizes on most instances but at the cost of a much longer running time. In terms of memory usage, our algorithms lead the field, though the `red2pack` heuristic algorithm was surprisingly competitive, performing more efficiently than anticipated.

6.2 Future Work

As we could see, the outerplanar graphs posed as a huge problem not only for our algorithms and reductions rules but also for `red2pack`. The design of reduction rules which also work properly on graph classes like the outerplanar graphs is definitely a point of interest. Furthermore, there still is room for improvements on our algorithms especially considering solution quality. On a side note, it would also be interesting to extend the concept of finding 2-packing sets using hypergraph matching to k-packing sets in general.

Zusammenfassung

Ein 2-packing set ist eine Teilmenge von Knoten eines Graphen, bei der keine zwei Knoten einen Abstand von zwei oder weniger zueinander haben. In dieser Arbeit versuchen wir ein 2-packing set zu finden, indem wir den Graphen zunächst in eine geeignete Hypergraph-Repräsentation transformieren. Das Finden eines hypergraph matchings auf dem Hypergraphen liefert dann eine äquivalente Lösung zum 2-packing set. Wir fokussieren uns außerdem auf Vorverarbeitungstechniken, indem wir die Effizienz bestehender Datenreduktionsregeln auf dem (Hyper-)Graphen untersuchen. Außerdem führen wir auch unsere eigenen Reduktionsregeln ein, welche von bereits bestehenden inspiriert sind. Unsere experimentellen Ergebnisse zeigen, dass einige der Reduktionen vergleichsweise effektiv sind. Weiterhin erreichen wir mit unserem heuristischen Algorithmus zwar teilweise nur eine Lösungsqualität von 88%, bei bestimmten Graphklassen weisen wir dafür jedoch eine wesentlich bessere Laufzeit auf. Zudem benötigt unser Algorithmus auch vergleichsweise wenig Speicherplatz.

Bibliography

[1] Diogo Vieira Andrade, Mauricio G. C. Resende, and Renato Fonseca F. Werneck. Fast local search for the maximum independent set problem. *J. Heuristics*, 18(4):525–547, 2012.

[2] Georg Anegg, Haris Angelidakis, and Rico Zenklusen. Simpler and stronger approaches for non-uniform hypergraph matching and the füredi, kahn, and seymour conjecture. In Hung Viet Le and Valerie King, editors, *4th Symposium on Simplicity in Algorithms, SOSA 2021, Virtual Conference, January 11-12, 2021*, pages 196–203. SIAM, 2021.

[3] David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In Reda Alhajj and Jon G. Rokne, editors, *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*. Springer, 2018.

[4] Jannick Borowitz, Ernestine Großmann, Christian Schulz, and Dominik Schweisgut. Finding optimal 2-packing sets on arbitrary graphs at scale. *CoRR*, abs/2308.15515, 2023.

[5] Marek Cygan. Improved approximation for 3-dimensional matching via bounded pathwidth local search. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 509–518. IEEE Computer Society, 2013.

[6] Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. Accelerating local search for the maximum independent set problem. In *15th International Symposium on Experimental Algorithms SEA*, volume 9685 of *Lecture Notes in Computer Science*, pages 118–133. Springer, 2016.

[7] Yihua Ding, James Zijun Wang, and Pradip K. Srimani. Self-stabilizing algorithm for maximal 2-packing with safe convergence in an arbitrary graph. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014*, pages 747–754. IEEE Computer Society, 2014.

[8] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2):201–213, 2002.

[9] 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, SEA^2 2019, Kalamata, Greece, June 24-29, 2019, Revised Selected Papers*, volume 11544 of *Lecture Notes in Computer Science*, pages 248–264. Springer, 2019.

[10] Alejandro Flores-Lamas, José Alberto Fernández-Zepeda, and Joel Antonio Trejo-Sánchez. Algorithm to find a maximum 2-packing set in a cactus. *Theor. Comput. Sci.*, 725:31–51, 2018.

[11] Alejandro Flores-Lamas, José Alberto Fernández-Zepeda, and Joel Antonio Trejo-Sánchez. A distributed algorithm for a maximal 2-packing set in halin graphs. *J. Parallel Distributed Comput.*, 142:62–76, 2020.

[12] Martin Fürer and Huiwen Yu. Approximating the k -set packing problem by local improvements. In Pierre Fouilhoux, Luis Eduardo Neves Gouveia, Ali Ridha Mahjoub, and Vangelis Th. Paschos, editors, *Combinatorial Optimization - Third International Symposium, ISCO 2014, Lisbon, Portugal, March 5-7, 2014, Revised Selected Papers*, volume 8596 of *Lecture Notes in Computer Science*, pages 408–420. Springer, 2014.

[13] Ernestine Großmann, Felix Joos, Henrik Reinstädtler, and Christian Schulz. Engineering hypergraph b-matching algorithms. *CoRR*, abs/2408.06924, 2024.

[14] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.

[15] Johan Håstad. Clique is hard to approximate within n$^{\text{1-epsilon}}$. *Electron. Colloquium Comput. Complex.*, TR97-038, 1997.

[16] Elad Hazan, Shmuel Safra, and Oded Schwartz. On the complexity of approximating k-set packing. *Comput. Complex.*, 15(1):20–39, 2006.

[17] Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. Exactly solving the maximum weight independent set problem on large real-world graphs. In Stephen G. Kobourov and Henning Meyerhenke, editors, *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*, pages 144–158. SIAM, 2019.

[18] Morten Mjedle. k-packing and k-domination on treegraphs. Master's thesis, University of Bergen, 2004.

[19] Zhengnan Shi. A self-stabilizing algorithm to maximal 2-packing with improved complexity. *Inf. Process. Lett.*, 112(13):525–531, 2012.

[20] Richard Sinkhorn and Paul Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21:343–348, 1967.

[21] Joel Antonio Trejo-Sánchez, Daniel Fajardo-Delgado, and José Octavio Gutiérrez García. A genetic algorithm for the maximum 2-packing set problem. *Int. J. Appl. Math. Comput. Sci.*, 30(1):173–184, 2020.

[22] Joel Antonio Trejo-Sánchez and José Alberto Fernández-Zepeda. Distributed algorithm for the maximal 2-packing in geometric outerplanar graphs. *J. Parallel Distributed Comput.*, 74(3):2193–2202, 2014.

[23] Joel Antonio Trejo-Sánchez, José Alberto Fernández-Zepeda, and Julio Cesar Ramírez Pacheco. A self-stabilizing algorithm for a maximal 2-packing in a cactus graph under any scheduler. *Int. J. Found. Comput. Sci.*, 28(8):1021–1046, 2017.

[24] Joel Antonio Trejo-Sánchez, Francisco Alejandro Madera-Ramírez, José Alberto Fernández-Zepeda, José Luis López-Martínez, and Alejandro Flores-Lamas. A fast approximation algorithm for the maximum 2-packing set problem on planar graphs. *Optim. Lett.*, 17(6):1435–1454, 2023.