Engineering Algorithms for the Weighted Maximum Clique Problem

Roman Erhardt

November 18, 2022

3658006

Master Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:

Prof. Dr. Christian Schulz

Co Supervisors:

Dr. Darren Strash
Dr. Nils Morten Kriege
Dr. Kathrin Hanauer

Acknowledgments

I would like to express my gratitude to my supervisor Prof. Dr. Christian Schulz for giving me the opportunity to write my master thesis about algorithms in graph theory, a field that I am especially interested in. I thank him and my co-supervisors Dr. Darren Strash, Dr. Nils Kriege and Dr. Kathrin Hanauer for mentoring and guiding me throughout this work. It was an enriching experience and a pleasure to work on this project together and to be a part of your research team. I would also like to thank the Algorithm Engineering Group Heidelberg as well as the colleagues at the University of Vienna for allowing me to use their resources for my experiments. Last but not least, I am thankful to my parents for the opportunities they gave me and to my family and friends for their continuous support throughout my life.

Bei der eingereichten Arbeit zu dem Thema Engineering Algorithms for the Maximum Weighted Clique Problem handelt es sich um meine eigenständig erbrachte Leistung. Ich habe nur die angegebenen Quellen und Hilfsmittel benutzt und mich keiner unzulässigen Hilfe Dritter bedient. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten vollständig übereinstimmt.

November 18, 2022

Roman Erhardt

Abstract

The maximum weighted clique problem (MWC) is a well-known problem in graph theory with many applications. In this work, both exact and heuristic algorithms, which interleave successful techniques from related work with novel graph reduction rules are proposed. While graph reductions based on upper bounds have been used for MWC in the past, we present reduction rules, that make use of local graph structures in order to identify and remove vertices and edges without reducing the optimal solution. A set of exact reduction rules is employed in an exact algorithm called MWCRedu, while heuristic reduction techniques based on machine learning models such as MLP, Deepset and GNN are explored in the heuristic framework MWCPeel. Experiments on a broad range of graphs show, that MWCRedu outperforms the current state-of-the-art exact solver TSM-MWC [25] for most inputs. Specifically for naturally weighted, medium-sized street network graphs and random hyperbolic graphs, which are considered to model real world graphs well, MWCRedu is faster by orders of magnitudes. The heuristic solver MWCPeel also outperforms its competitors FastWCLq [12] and SCCWalk4l [47] on these instances, but is slightly less effective on extremely dense or large instances.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Finding a clique in a graph is a classic problem in graph theory with applications in any field where some form of commonality can be used to formulate the problem. One of the earliest and most well-known applications can be found in social networks, where actors are modeled as nodes in a graph and the ties between them as edges. A clique is then a group of actors who all know each other and can be identified by looking for a group of nodes in the graph that are adjacent to each other. Finding cliques in social networks can help explain (and predict) various social and psychological observations, like homogeneity in decision making among groups or the forming of group standards in general [49]. Other applications can be found in biochemistry, where finding cliques can be used to research the interaction between rigid molecules, giving important information that may be used, e.g. in drug discovery [37]. Here, the nodes of a graph model possible matches, i.e. an assignment of one molecule's feature to the others that would allow docking at this point. Edges are used to model the compatibility of matches, where two matches are compatible if the distance between the respective features is within some tolerance. A clique then represents a group of matches that can be formed simultaneously. To get the binding mode that is most likely to be observed in experiments, finding the clique of maximum cardinality can give a good approximation.
To model more complex scenarios, the nodes in a graph are often assigned different weights, which gives rise to the more complex generalization of maximum clique (MC), the maximum weighted clique problem (MWC). One application of MWC has been discovered by Zhang et al. [55] in video object co-segmentation. Their method allows robust tracking of objects in a video despite noise or occlusion, by exploiting the commonality of same objects in different frames. This is achieved, by representing video objects from each frame as nodes and weighing them according to a function over some criteria, such as boundary definition, difference from surroundings and optical flow gradient. Two nodes

(from two different frames) are then connected by an edge, if their similarity is above some threshold. A (regulated) maximum weighted clique in this graph then identifies the object that is most consistent across different frames and best fulfils the scoring criteria. More applications can be found, e.g. in coding theory [56], combinatorial auctions [51] and genomics [9].

There has been extensive research on solving the various formulations of the clique problem using both heuristic and exact approaches. Despite this, state-of-the-art algorithms still have problems finding the solution to specific graph instances in reasonable time, which is no surprise as the problem has been proven to be NP-hard [28]. One powerful technique to tackle NP-hard problems in graph theory are graph reduction rules, which search the graph for specific local structures in polynomial time and classify the contained vertices accordingly, effectively reducing the input instance to an equivalent, smaller instance [1]. Such reductions are often seen in related work for the equivalent maximum weighted independent set problem (MWIS) and minimum weighted vertex cover problem (MWVC), where they achieve great results [32]. Besides exact reductions, which are guaranteed not to reduce the solution quality, heuristic reductions that make use of machine learning techniques have been shown to be effective as well [44, 33]. Inspired by these ideas, this work is focused on engineering a set of exact reduction rules and several heuristic graph reduction schemes, aimed at reducing the number of vertices and edges in the input graph instance. While removing edges is mostly useful for freeing up the reduction space for vertex reductions, reducing the number of vertices that are not part of the solution according to an exact reduction, a heuristic rule or a machine learning algorithm is guaranteed to speed up any existing solver that is applied on the reduced graph.

1.2 Contributions

The main contributions of this work are both exact and heuristic algorithms for the MWC problem, that combine effective techniques from state-of-the-art solvers with novel vertex- and edge reduction rules. To the best of our knowledge, this is the first attempt at engineering reduction rules that exploit local graph structures for the MWC problem and the first attempt at conceiving a reduction solely aimed at reducing the number of edges in a graph. For the purpose of applying the reductions, a reduce-and-peel framework as seen in recent algorithms for the MWIS problem is engineered. The framework describes how to apply each reduction, as well when to stop applying reductions and switch to the exact solver. For many input graph instances, the resulting solver is able to greatly reduce the graph size and outperform state-of-the-art solvers.

1.3 Organization

The remainder of this work is organized as follows. In Chapter 2 fundamentals, such as graph notations and relevant problems in graph theory are given. Furthermore, several techniques that are used later on, such as hashing, branch-and-bound and machine learning are explained. Chapter 3 covers related work, starting with classic algorithms for MC up to current state-of-the-art algorithms for MWC, as well as techniques from the related minimum vertex cover and maximum independent set problem. The contributions of this work are presented in Chapter 4, including the novel reductions rules and how to integrate them in a holistic MWC solver. Chapter 5 then discusses the implementation of the techniques from a practical point of view. The performance of the presented techniques is evaluated and compared with state-of-the-art solvers on a broad range of graphs in Chapter 6. We then conclude this work in Chapter 7 and discuss possible future work.

CHAPTER 2

Fundamentals

2.1 Graph Preliminaries

A graph G is a mathematical structure that is used to model pairwise relationships between objects. Objects are represented as a finite set of vertices $V = \{v_1, ..., v_{|V|}\}$, where each vertex $v_i \in V$ is associated with a weight $w(v_i) \in \mathbb{N}$. An unweighted graph is a special case where $\forall v_i \in V, w(v_i) = 1$. The relationships between the vertices are represented by a set of edges $E = \{e_1, ..., e_{|E|}\}$. The graph density ρ is computed as $\rho = \frac{2|E|}{|V|(|V|-1)}$. In an undirected graph, two vertices v_i and v_j are considered adjacent, if $\{v_i, v_j\} \in E$. The set of adjacent vertices of a vertex v_i makes up its neighborhood $N(v) = \{v_j \mid \{v_i, v_j\} \in E\}$, or $N[v]$, if v is included. The number of neighbors of a vertex v is given by the degree of the vertex $d(v) = |N(v)|$. The graph containing vertices V and edges E is written as $G = (V, E, w)$. The sub-graph induced by the subset $V' \subseteq V$ is denoted by $G[V']$ and only contains the edges $E' = \{\{v_i, v_j\} \in E \mid v_i, v_j \in V'\}$. The maximum weight of V' is denoted by $w^*(V')$ and can be computed as $\max_{v_i \in V'}(w(v_i))$. A subset $C \subseteq V$ is a clique, if $G[C]$ is a complete graph. The weight of a clique is computed as $\sum_{v_i \in C} w(v_i)$. The most simple way to represent a graph is an adjacency matrix A, i.e. a boolean $|V| \times |V|$ matrix, where an element A_{ij} evaluates to $true$, if $\{i, j\} \in E$. Another way of representation is an adjacency list A_{list}. This data structure uses a list with N elements, denoting the vertices in the graph, where each element points to another list containing the vertices neighbors, i.e. $A_{list}[v] = N(v)$.

2.2 Related Problems

The maximum weighted clique problem (MWC) is a generalization of the maximum clique problem (MC), which is to find a clique of maximum cardinality in an unweighted graph. Its decision version is among the first 21 NP-complete problems presented in Karp's

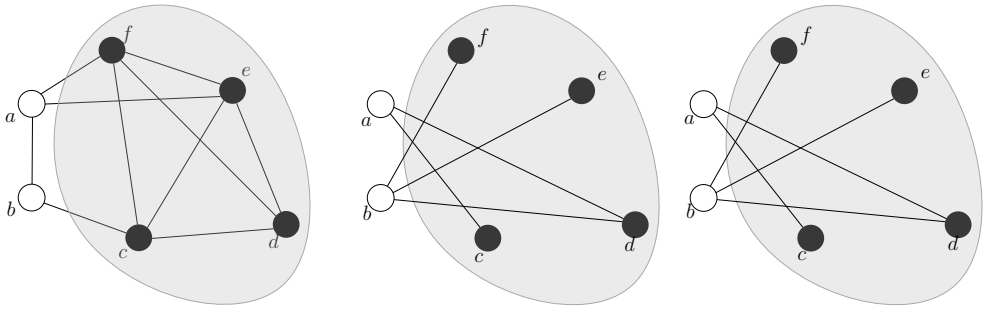

Figure 2.1: An illustration of the relation between MC, MIS and MVC. Given the initial graph G with $V = \{a, b, c, d, e, f\}$ (left) and its complementary graph $\overline{G}$ (right), the set of vertices $\{c, d, e, f\}$ is an MC of G and an MIS of $\overline{G}$, whereas $\{a, b\}$ forms an MVC [50].

seminal paper on computational complexity [28]. The difference between the two is that for MWC, vertex weights can take arbitrary values. The problem is then to find a clique with the largest combined vertex weight. MC is dual to the maximum independent set problem (MIS) and the minimum vertex cover problem (MVC), another two well known combinatorial optimization problems. The MIS problem is to determine the largest set of vertices in a graph such that none of the vertices contained in the set are adjacent. It is easy to see, that solving MIS on the complement graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{v_i, v_j \mid v_i, v_j \in V \wedge \{v_i, v_j\} \notin E\}$, directly gives a solution to MVC in $\overline{G}$ and MC in G (Figure 2.1). By extension, MWIS and MWVC are dual to MWC. MIS is also closely related to the vertex coloring problem (VC), which is to assign a label $c \in \mathbb{Z}$ to each vertex, such that all pairs of adjacent vertices are assigned a different label. The labels are typically assigned starting from 1, where each value represents one color class. Since all vertices with the same label are non-adjacent, VC partitions the vertices in the graph into independent sets. The minimum number of labels is also called the chromatic number of the graph.

Another famous optimization problem that is interesting for solving MC is the maximum satisfiability problem (MaxSAT), which is to determine the maximum number of clauses in a Boolean formula in conjunctive normal form that evaluate to $true$ at the same time by some assignment of values to the variables of the formula. The problem is NP-hard and APX-complete, meaning that there cannot be a polynomial time approximation scheme (PTAS) unless $P = NP$. A variation of the MaxSAT problem called partial MaxSAT can also be used to encode MC. Partial MaxSAT differentiates between soft clauses, which are relaxable, and hard clauses, which must evaluate to $true$. The objective here is, to find a variable assignment that satisfies all hard clauses and the maximum number of soft clauses. The encoding is done by introducing a unit soft clause for each vertex, indicating whether it is in the maximum clique or not, and a hard clause $\bar{x_i} \vee \bar{x_j}$ for each non-connected pair of vertices. The optimal solution to the partial MaxSAT instance then yields the maximum clique of the graph [34].

Algorithm 1 Branch and Bound

```
function MAIN
    x̂ ← ∅
    branch_bound(∅, V)
end function
function branch_bound(x, C)
    if f(x) > f(x̂) then
        x̂ ← x
    end if
    for v ∈ C do
        C′ ← C \ {v}
        if upper_bound(C′) + f(x) > f(x̂) then
            branch_bound(x, C′)
        end if
        x′ ← x ∪ {v}
        if upper_bound(C′) + f(x′) > f(x̂) then
            branch_bound(x′, C′)
        end if
    end for
end function
```

2.3 The Branch and Bound Paradigm

Branch and Bound (B&B) is an algorithm design paradigm for solving optimization problems exactly by systematically searching the solution space. A general algorithm is given in Algorithm 1. During the search procedure, the graph nodes V are enumerated according to a specific predetermined ordering. At each node in the search tree, the search path branches in two, with one path including the current node in the solution and one excluding it. The search space is thus a binary tree where the leaves represent all possible solutions to the problem. At each node of the tree, the current best solution $\hat{x}$ and its value $f(\hat{x})$ are known and mark the lower bound for the solution quality. Furthermore, an upper bound can be computed for the subtree rooted at the current node. Before branching on the node, it is first checked if the upper bound is higher than the lower bound. If not, the subtree cannot contain a solution that is better than the one which has already been found and the subtree can be discarded. While B&B algorithms still have exponential running time in the worst case, they are much faster than brute force search in practice since a large part of solution candidates can be excluded early thanks to the bounding scheme. How fast the algorithm will converge to the optimal solution is typically dependent on the tightness and efficiency of the computed upper bounds.

2.4 Hashing

Hashing denotes the process of mapping any value from an arbitrarily sized set to a smaller set of specific size via a hash function. A common application of hashing is to speed up the search for specific elements in a list, making it an important asset when checking a graph represented by an adjacency list for specific edges efficiently. This is done by computing the hash value of that element and using it as an index for the hash table, potentially giving its position in constant time if the hash value is unique, or in $\mathcal{O}(n)$ otherwise. Since the hash set is smaller than the input set, the hash function may map two different values from the input set to the same value of the output set. For this reason some form of collision handling must be implemented when using the hash value as an index. The hash function should be chosen, such that the number of collisions and the following, often expensive collision handling procedures are minimized. A process known as universal hashing guarantees this in expectation, by selecting the hash function at random from a family of hash functions:

$$h_a = a \cdot x \mod m \tag{2.1}$$

where $x = (x_1, ..., x_k)$ is the element, which the hash should be computed for, $a = (a_1, ..., a_k) \in \{0, ..., m-1\}^k$ is a randomly chosen factor to ensure an equal distribution over the entire solution space and m is the chosen output set size [36].

2.4.1 Bloom Filter

A Bloom filter is a probabilistic data structure that makes use of hashing to check whether an item is in a given set or not. It was designed by Burton H. Bloom [8] to efficiently trade off space requirements and performance given a tolerated error margin. The key element is the lookup function, which has a time complexity of $\mathcal{O}(1)$ and is guaranteed to return no false negatives. This is done by first computing the hash value of an element before adding it to the list and using it as an index to set a bit in an internal hash set. Whenever an element is looked up via the same process, the function returns $true$ if the bit is set, or $false$ if it is not. While one can be certain, that the element is not contained in the list if the function returns $false$, a positive result gives no information as to whether the bit in the hash set has been set by this element or another element that maps to the same hash value. In this case, the list would have to be searched for the item using different methods.
The parameters of the Bloom filter are interdependent and can be chosen to achieve a specific likelihood of false positives. To decrease the chance of a false positive, both a larger hash table or a higher number of hash functions can be used. If the number of items n in the list is known and the optimal number of hash functions is chosen, then the size m of the internal array can be computed as

$$m = \left\lceil \frac{n \cdot \log(\frac{1}{p})}{\log^2(2)} \right\rceil \tag{2.2}$$

where p is the probability of false positives [8]. The optimal number of hashes k is computed as

$$k = \left\lfloor \frac{m \cdot \log(2)}{n} \right\rceil . \tag{2.3}$$

However, a higher number of hash functions also means more hashes need to be computed and more bits need to be read during a lookup or written when adding an item. Overall k clearly dominates the runtime of the Bloom filter data structure, which is why it is sometimes beneficial to choose a less than optimal number of hash functions.

2.5 Machine Learning

Machine learning (ML) denotes a field in computer science, where data is leveraged to improve the performance on some task. ML is a subfield of artificial intelligence, since a machine learning model typically learns new information such as patterns and correlations from the given data. Most machine learning approaches can be categorized as either supervised learning, unsupervised learning or reinforcement learning. The latter two have in common, that no labels, i.e. solutions to a subset of the data, are required. In unsupervised learning, a loss function that depends only on the input data is minimized. The goal here is typically to find patterns or cluster the data. Reinforcement learning trains the model by guiding its actions via a reward function. The model therefore tries to learn a sequence of actions that maximizes its rewards. However, given the possibility of generating labels for large amounts of data, supervised learning is usually the most effective method and will be the approach discussed in the remainder of this section. Especially neural networks (NN) have proven versatile and efficient ML tools in recent years. For training, the NN model first receives a data sample as an input and uses its current state of parameters to infer a prediction. Next, a loss function is employed to measure the distance between the predicted values and the given label. To learn parameters that result in a prediction close to the label, this loss function is minimized. This is done during a process called back-propagation, where the gradients of each parameter w.r.t. the loss are computed and adjusted accordingly. Finishing such a cycle for all given training samples is called an epoch. In this work we evaluate the following three NN variants: Multi-layer Perceptrons (MLP), DeepSets and Graph Neural Networks (GNN).

2.5.1 Multi-Layer Perceptron

Artificial neural networks are inspired by the functionality of neurons in the biological brain, which interpret and relay chemical and electrical input signals as an essential part of the decision making process. They were first conceived by Rosenblatt [42], who proposed to model a neuron as a simple linear function

$$y = w \cdot x + b \tag{2.4}$$

where x is the input signal, y the output signal and w and b are trainable parameters of the model. A single model contains many neurons, which are distributed along the input layer, the hidden layer and the output layer. While the number of neurons in the input and output layers are necessarily the number of inputs and outputs respectively, the number of neurons in the hidden layers as well as the number of hidden layers itself can be arbitrarily chosen. Each layer is followed by a non-linear activation function, such that the MLP does not degenerate to a linear regression model and is capable of learning even complex functions. One such activation function is ReLU [2], which only allows positive values to pass on to the next layer. Neural networks that do not have connections which form a cycle and are fully connected are called MLPs.

2.5.2 DeepSets

One drawback of the original neural network model, is that the input is fixed regarding dimension and ordering. Zaheer et al. [54] find a way around this problem in an architecture called DeepSets, in which the model input parameters are allowed arbitrary dimensions. The main idea is to first apply an inner neural network ϕ on each input x_m and to then sum up the outputs and apply another neural network ρ on the sum to compute the final prediction:

$$f(X) = \rho\left(\sum_{x \in X} \phi(x)\right). \tag{2.5}$$

With this model, any set functions can be learned and applied on arbitrarily sized inputs. In order to still be able to perform efficient batching during training and inference, input sets are zero-padded to a certain dimension.

2.5.3 Graph Neural Networks

Different from MLP and DeepSets, which directly infer a prediction on the target value from the given input, GNN compute a d-dimensional real vector for each input. Being specifically engineered to operate on graph-type data, a GNN layer takes an $|V| \times |V|$ adjacency matrix A and a $|V| \times d$ vector $H^{(0)}$ containing d features per node as input. To avoid memory issues for larger graphs, the adjacency matrix is stored in Coordinate list (COO) format. COO is a sparse matrix format that stores only non-zero entries explicitly. For this, each entry is represented by the two-tuple $(row, column)$. Similarly to how deep sets handle arbitrarily sized inputs, each layer updates the given features for each node by computing a weighted sum over the node features in their neighborhood. GNN were first presented by Kipf and Welling [29] with a layer called GCN, which implements this procedure as follows:

$$H^{(t)} = \sigma\left(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}H^{(t-1)}W^{(t-1)}\right) \tag{2.6}$$

where σ is a non-linear activation function, such as ReLU, $\widetilde{A} = A + I_N$, I_N is the identity matrix, $\widetilde{D}_{ii} = \sum_j \widetilde{A}_{ij}$ and $W^{(t-1)}$ is a layer-specific trainable weight matrix. As shown by Gilmer et al. [21], different kinds of permutation-invariant differentiable functions $f_{aggr}^{W_2}$ to aggregate information about the neighbor features, and functions $f_{merge}^{W_1}$ to merge the vertex feature of the last iteration with the aggregated neighborhood information can be used, resulting in the following more general formulation:

$$f^{(t)}(v) = f_{merge}^{W_1}\left(f^{(t-1)}(v), f_{aggr}^{W_2}(\{f^{(t-1)}(w) \mid w \in N(v)\})\right) \tag{2.7}$$

More recently, Xu et al. [53] present a GNN framework called Graph Isomorphism Network (GIN), which they prove to be at least as effective as the Weisfeiler-Lehman graph isomorphism test, a famous test for identifying non-isomorphic graphs by computing signatures for each node based on their neighborhood. The proof extends the idea of Zaheer et al. from sets to multi-sets. Specifically, they show that there must be a function $f : \mathcal{X} \to \mathbb{R}^n$, s.t. $h(X) = \sum_{x \in X} f(x)$ is unique for each multi-set $X \subset \mathcal{X}$ of bounded size. Furthermore any multi-set function g can be obtained from $h(X)$, by applying some function Φ on it. Both functions f and Φ can be learned by MLP models, which is why the final function for updating node representations looks as follows:

$$h_v^{(t)} = MLP^{(t)}\left((1 + \epsilon^{(t)}) \cdot h_v^{(t-1)} + \sum_{u \in N(v)} h_u^{(t-1)}\right) \tag{2.8}$$

where ϵ is either a learnable parameter or a constant.

CHAPTER 3

Related Work

A lot of research has been done for MC, the unweighted variant of the MWC. Algorithms that aim at finding the exact solution almost exclusively make use of the branch and bound (B&B) paradigm (Section 2.3), whereas heuristics are typically based on local search strategies. Both exact and heuristic algorithms leading up to the respective state-of-the-art solvers are summarized in this section. A more detailed review on approaches to solve MC has been published in 2015 by Wu and Hao [50]. As the MWIS and MWVC solvers can be directly applied to solve MWC instances, some algorithms and techniques for these problems are looked at as well.

3.1 Exact Solvers for the Maximum Clique Problem

The foundation for most exact algorithms these days was laid by CP [13], a B&B based solver for the MC problem (Algorithm 2). On each including branch, the clique property can be used to reduce the candidate set P, by keeping only the intersection of the current candidate set with the neighborhood of the newly included vertex. This is possible since only vertices that are adjacent to all vertices in the current clique can be added. The size of the current clique $|C|$ and the current candidate set $|P|$ combined always provide an upper bound on the current search space, while the lower bound is given by the size of the best clique that was found at that point, $|\hat{C}|$.
Most exact solvers that follow the CP algorithm use the same framework and focus on computing a tighter upper bound. One successful technique is vertex coloring (Section 2.2): If a graph can be colored with k colors, then the number of vertices in a clique must be $|C| \leq k$. This holds because every color class can only contribute up to one vertex to a clique since vertices of the same color must be non-adjacent. With this, a much better upper bound can be achieved than by only using the candidate set size. However, since VC is NP-hard, computing a good coloring can be quite expensive in itself, which is why fast heuristics are preferably employed. A popular heuristic works by iterating the candidate vertices and

Algorithm 2 CP algorithm [13]

```
function MAIN
    Ĉ ← ∅
    Clique(∅, V)
    return Ĉ
end function
function Clique(set C, set P)
    if |C| > |Ĉ| then
        Ĉ ← C
    end if
    if |C| + |P| > |Ĉ| then
        for all p ∈ P in predetermined order do
            P ← P \ {p}
            C' ← C ∪ {p}
            P' ← P ∩ N(p)
            Clique(C', P')
        end for
    end if
end function
```

assigning the smallest color class to each vertex that is unique among its neighbors. If all existing color classes are already used in a neighborhood, an additional class is introduced until all vertices are colored.

Early classic algorithms to employ VC are e.g. BT [4] and MCQ [45], where the coloring of $G[P]$ is computed at every node in the search tree s.t. $|P| > |\hat{C}|$, pruning the subtree whenever the candidate set could be colored with less or equal than $|\hat{C}| - 1$ colors. MCQ additionally introduces the idea of using the coloring as a branching order by branching on the highest color vertices first, the intuition being that vertices belonging to a high color class must have a lot of interconnected neighbors and thus have a high chance of being in a large clique. Konc and Janezic [30] improve the coloring based upper bound further with MaxCliqueDyn by presenting the vertices to the greedy coloring heuristic in a non-increasing order of their degrees. They dynamically recompute the vertex degrees and reorder the vertices especially near the first few branches and only if its color number $Color_j$ is below the threshold $|\hat{C}| - |C| + 1$. Another idea of improving the bound is the idea of recoloring as presented in MCS [46]. That is, changing the color of a vertex to a color class below $k_{min} = |\hat{C}| - |C|$ in order to decrease the total number of vertices to be searched. In order to recolor a vertex v, a vertex $w \in Color_j$ has to be found, that is the only neighbor of v in that color class. Furthermore, there must be some color class that does not contain a neighbor of w. If that is the case, v can be moved to $Color_j$ with $j < k_{min}$. Li et al. [35] improve the upper bound UB_{Color} obtained by VC further, by applying MaxSAT reasoning in their algorithm MaxCLQ. They do this by transforming the

Algorithm Name	Publishing Date	Description
CP [13]	1990	Fundamental B&B algorithm.
BT [4]	1990	First algorithm to use vertex coloring to improve upper bounds.
MCQ [45]	2003	Introduced the idea of using the coloring as an ordering for B&B.
MaxCliqueDyn [30]	2007	Improved the coloring heuristic.
MCS [46]	2010	Proposed the idea of decreasing the set of branching vertices by recoloring.
MaxCLQ [35]	2010	First algorithm to use MaxSAT reasoning for reducing the search space.

Table 3.1: Overview of exact solvers for MC.

graph into a MaxSAT instance, where they add ISs as soft clauses instead of the vertices itself. To improve UB_{Color}, they repeatedly look for a set of disjoint conflicting soft clauses. Each time such a set has been found, UB_{Color} can be reduced by 1, since at least one color class cannot contribute a vertex to the clique. The set is then relaxed, by adding an additional variable to each clause, so that further conflicting soft clauses can be searched. Table 3.1 gives an overview of the exact algorithms for MC.

3.2 Exact Solvers for the Maximum Weighted Clique Problem

The MWC has significantly less solvers available compared to MC, which may be due to its higher complexity and the fact that some ideas from MC solvers are not applicable here. Ideas that have been extended to MWC are the upper bounds based on VC and MaxSAT. The extension of VC was first proposed by Kumlander [31] as follows: Given a valid vertex coloring of $G[P]$ into the color classes $\Pi = \{D_1, D_2, ..., D_{|\Pi|}\}$, the upper bound can be computed as $\sum_{j=1}^{|\Pi|} w^*(D_j)$. This holds, since each color class can contribute up to one vertex $v \in D_j$ of weight $w(v) \leq w^*(D_j)$ to the same clique. MWCLQ [17] is the first to implement the idea of MaxSAT reasoning introduced by the MC solver MaxCLQ [35]. It does so, by encoding the graph into a MaxSAT instance as before, but associating a weight with each literal in a soft clause. When detecting a set of disjoint conflicting soft clauses, the set is split by weight into the conflicting subset, and a subset that can be searched further, similar to MaxCLQ. WLMC [26] also relies on MaxSAT reasoning and contributes an efficient preprocessing step, that computes an initial clique $\hat{C}$, as well as a vertex branching ordering. It furthermore computes a simple upper bound on the maximum weight clique a vertex can be part of as $w(N[v])$, and removes vertices s.t. $w(N[v]) \leq w(\hat{C})$. TSM-MWC [25] refines the approach further with a two-stage MaxSAT reasoning approach, that applies less

Algorithm Name	Publishing Date	Description
MWCLQ [17]	2016	First algorithm to extend the MaxSAT reasoning approach to MWC.
WLMC [26]	2017	Extended MWCLQ by an efficient preprocessing step.
TSM-MWC [25]	2018	Refined the MaxSAT reasoning approach used in the previous solvers. Represents the state-of-the-art.

Table 3.2: Overview of exact solvers for MWC.

expensive MaxSAT techniques to reduce the number of branching vertices, before looking for disjoint conflicting soft clauses exhaustively. TSM-MWC currently achieves the best results for a wide spectrum of graph instances, most notably massive real-world graph instances, and can be seen as the current state-of-the-art. Table 3.2 gives an overview of the exact algorithms for MWC.

3.3 Heuristic Solvers for the Maximum Clique Problem

The most successful heuristic framework for the MC is local search. In a local search algorithm, a clique is constructed by inserting a single starting vertex into the growing clique C and repeatedly adding vertices that are adjacent to all vertices in C using some evaluation function. The set of candidate vertices is denoted as PA. Once no more add operations can be performed, some vertices must be removed before a larger clique can be constructed. In order to avoid loops, local search algorithms typically implement a prohibition rule that restricts which vertices can be removed or added back into the solution. In 1993, Gendrau et al. [20] propose two algorithms, DT and PT, implementing this procedure. DT follows a deterministic scheme by adding the vertex with the highest degree first. When no more vertex can be added, the vertex that results in the largest PA is removed. As a prohibition rule they simply disallow removed vertices to be added again for some time. The second algorithm PT randomly selects which vertex to add to the current solution. Battiti & Tecchiolli build on these ideas in their algorithm RLS [5]. The algorithm works similar for the most part, but puts more emphasis on diversification by choosing the prohibition period dynamically and adding a restart mechanism, which ensures that all vertices are included in a clique at some point. Later, PLS [40] is among the first to include the swap operator in the main search procedure. This operator looks for a vertex in OM, a set of vertices that are connected to all but one vertex of C. In PLS, the operator is applied once no more add operations are possible. If neither operators are applicable, the clique is perturbed by adding a random vertex and removing all non-adjacent vertices from the clique. The algorithm uses several vertex selection rules for different types of graphs, which are later improved by CLS [41]. Wu and Hao show in their algorithm MN/TS [52],

Algorithm Name	Publishing Date	Description
PT & DT[20]	1993	Two early heuristic algorithms implementing tabu search.
RLS [5]	1994	Improved the previous tabu search approach by dynamic rules and a restart mechanism.
PLS [40]	2006	Introduced the swap operator to tabu search.
CLS [41]	2011	Improved vertex selection rules of PLS.
MN/TS [52]	2012	Improved previous algorithms by a proposing a more liberal usage of the swap operator.
BLS [7]	2013	Introduced dynamic perturbations to improve the global performance.

Table 3.3: Overview of heuristic solvers for MC.

that it is beneficial to regard neighboring solutions presented by the different operators at all times. Lastly, Benlic and Hao [7] introduced the idea of entirely changing the search area upon stagnation in BLS. Table 3.3 gives an overview of the heuristic algorithms for MC.

3.4 Heuristic Solvers for the Maximum Weighted Clique Problem

Some of the available solvers for the MC have been extended to the MWC as well. Such is the case for PLS [39], where instead of adding a random vertex among PA, the vertex is randomly chosen only among the highest weight vertices. MN/TS [52] is also applicable on the MWC and even outperforms previous approaches. As the authors point out, it is especially important for the MWC to combine the search spaces of the three operators add, swap and drop, since add is not strictly the better choice over swap or swap over drop respectively here. LSCC [48] improves MN/TS by adding a prohibition rule based on configuration checking and altering it to fit MWC. In configuration checking, all vertices initially have a configuration label 1, indicating that they are allowed to be added to the solution. However they may be disallowed, once their configuration changes to 0. Strong configuration checking (SCC) uses this tool to encourage or disencourage adding the neighbors of a vertex v to the clique, based on the operation used on v. For example, SCC encourages adding neighbors of an added vertex, but not neighbors of a dropped vertex. The second algorithm LSCC+BMS further improves LSCC by using Best from Multiples Selection (BMS), a strategy used to decide which vertex from the candidate set to add next [10]. BMS works by randomly sampling k different candidate vertices and choosing the best vertex with respect to some benefit estimation function. Cai and Lin [11] interleave local search with BMS and graph reductions in FastWCLq. The reductions they use compute

upper bounds for each vertex, including the one used in WLMC, and remove the vertex if one of the computed upper bounds is lower than the weight of the current best clique. Every time an improved solution is found by local search, the reductions are reapplied, in turn improving the odds of local search finding the optimal solution. SCCWalk4l [47] adopts the previously seen successful strategies SCC, BMS and the graph reductions. They furthermore introduce a technique called walk perturbation, which, similar to PLS [40], adds a random vertex to the solution when the search stagnates and removes all vertices that become invalid by this perturbation from PA. Finally, FastWCLq has been further improved to also apply a reduction and hill climbing method based on vertex coloring [12]. SCCWalk4l and FastWCLq present the current state-of-the-art for heuristic MWC solvers, with the former being especially dominant in small dense networks, such as graphs from the DIMACS and BHOSLIB challenge [47], and the latter showing the best results in massive real-world networks [12].
Besides local search, other approaches have been tried as well to improve existing state-of-the-art solvers. Using Machine Learning for Problem Reduction (MLPR), Sun et al. [44] are able to speed up existing solvers for some problem instances. Their method uses Support Vector Machines (SVM), a supervised machine learning approach, to learn an evaluation function on the quality of vertices. They apply this model to remove vertices that receive a sufficiently low prediction of being in the optimal solution before applying TSM-MWC or FastWCLq on the reduced graph. Besides basic vertex and graph metrics, their model uses two probabilistic features, that, while proving effective, don't scale well to large graph instances. Very recently, another similar approach has been presented for the equivalent MWVC by Langedal et al. [33]. Instead of SVM, they employ a graph neural network model (Section 2.5.3) in their solver GNN & LS to classify whether a vertex is in the solution or not. The algorithm starts by applying MWVC reductions on the input graph exhaustively and solving small sub-graphs exactly by a branch-and-reduce. After that, an ML architecture combining GNN with MLP layers is applied on the remaining graph to classify the vertices with the highest probability of being in or out of the solution. The solution is then optimized using local search. Overall their approach outperforms competing state-of-the-art heuristics both in speed and solution quality.

3.5 Maximum Weighted IS Reductions

Being closely related to MWC, ideas from MWIS solvers can often be used to make progress in solving MWC and vice versa. The MWIS reductions that inspired some of the novel MWC reductions shown later are outlined in this section. From the numerous reductions that have been engineered for the MWIS, the three exact reduction rules twin, domination and isolated vertex removal [32], as well as an inexact reduction introduced by Dahlum et al. [15] are of particular interest.

Algorithm Name	Publishing Date	Description
PLS [39]	2008	Extended PLS to MWC.
MN/TS [52]	2012	The same as MN/TS for MC.
LSCC & LSCC+BMS [48]	2016	Two local search algorithms that improved MN/TS by implementing SCC and BMS.
FastWCLq [11, 12]	2016/2021	Local search algorithm using BMS and branching techniques in tandem with graph reductions. Represents the state-of-the-art.
SCCWalk4l [47]	2020	Combined SCC, BMS, graph reductions and walk perturbation. Represents the state-of-the-art.
MLPR [44]	2019	Employed an SVM model to remove vertices that are unlikely to be in the solution.
GNN&LS [33]	2022	MWVC solver using a GNN model to remove vertices that are unlikely to be in the solution.

Table 3.4: Overview of heuristic solvers for MWC.

3.5.1 Twin Reduction

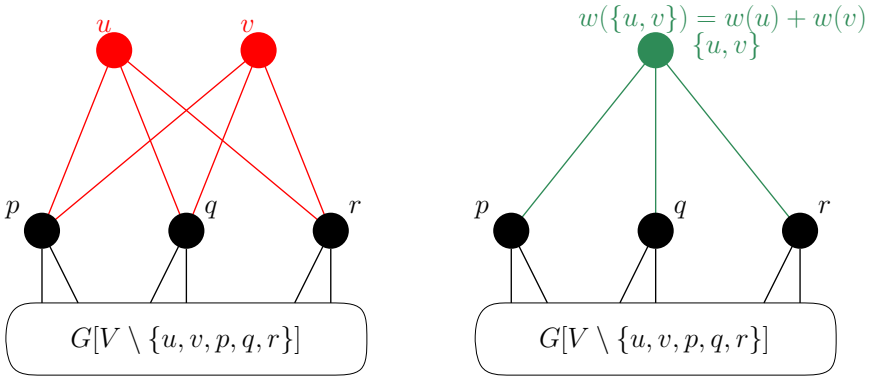

Figure 3.1: Twin Reduction for MWIS [32]

The twin reduction can be applied on two vertices $v, u \in V$, if $\{v, u\} \notin E$ and $N(u) = N(v)$. Depending on the weight distribution, either u and v or their neighborhood will be in the MWIS. u and v can thus be simplified to a single vertex $\{u, v\}$, as illustrated in Figure 3.1. Depending on the weight distribution, one of two further reductions may be applicable:

(i) if $w(\{u, v\}) \geq w(N(\{u, v\}))$, u and v are part of some maximum IS and G can be reduced to $G' = V \setminus N(\{u, v\})$

(ii) if $w(\{u,v\}) < w(N(\{u,v\}))$, but $w(\{u,v\}) > w(N(\{u,v\})) - \min_i(w(x_i))$, where $x_i \in N(\{u,v\})$, $\{u,v\}$ and $N(\{u,v\})$ can be folded to a new vertex v' with weight $w(v') = w(N(\{u,v\})) - w(\{u,v\})$. If v' is in the final MWIS I' of the reduced graph G', $N(\{u,v\})$ are in I, otherwise u and v are in I.

3.5.2 Domination Reduction

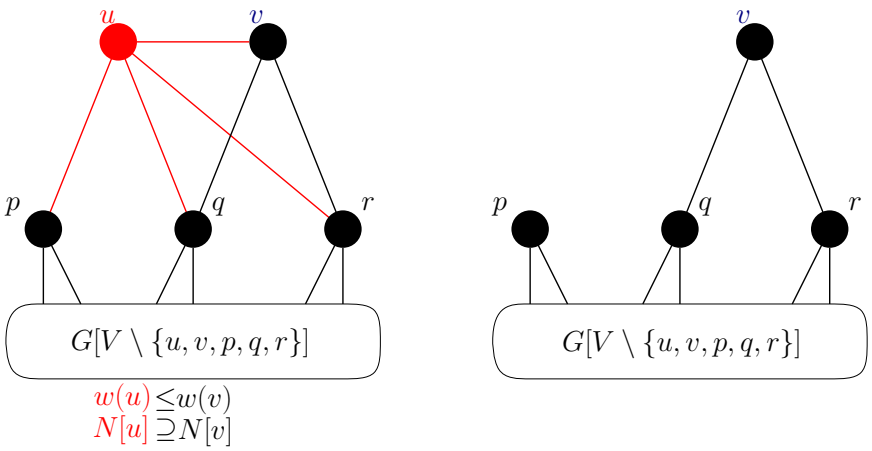

Figure 3.2: Domination Reduction for MWIS [32]

For the domination reduction, two vertices $v, u \in V$ have to be identified, for which $\{u,v\} \in E$, $N[u] \supseteq [v]$ and $w(u) \leq w(v)$ holds, as illustrated in Figure 3.2. In this case, an IS including u could always be enlarged by excluding u and including v, which is why u can safely be removed from the graph without reducing the maximum solution weight.

3.5.3 Simplicial Vertex Removal

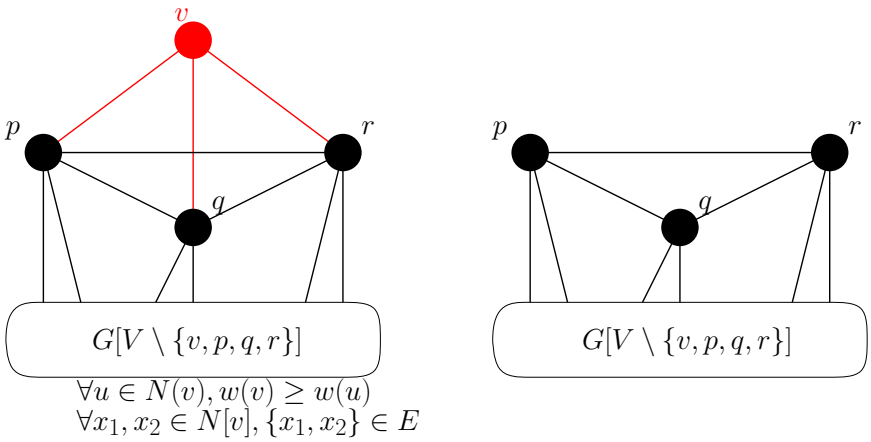

Figure 3.3: Simplicial vertex removal reduction for MWIS [32]

Simplicial vertices are vertices v whose neighborhood forms a clique C, i.e. $\forall x_1, x_2 \in N[v], \{x_1, x_2\} \in E$. Since v has no neighbor outside the clique, there must be some MWIS that includes v, if it has the highest weight of all the vertices in the clique.

3.5.4 Increasing Transformations

Another idea that was first applied on the MWIS by Gellner et al. [19], is to transform the graph G to G' by removing specific vertices, and then adding vertices and edges such that the graphs G and G' are equivalent. While the transformations may result in a larger graph initially, they open up the reduction space for other reductions, often leading to a smaller graph in the end.

3.5.5 Vertex Peeling Reduction

Besides exact reductions that are not always applicable and sometimes take a long time to compute, some research has been done on heuristic reduction rules as well. First conceived by Dahlum et al. [15] for MIS, the idea is to cut high-degree vertices from the graph in order to speed up local search, a procedure that was later called vertex peeling by Chang et al. [14]. The rule is highly intuitive as high-degree vertices exclude more vertices from an IS than low-degree vertices and are thus less likely to be in the solution. Furthermore, removing these vertices and all edges connected to them sparsifies the graph, often opening up exact reduction rules that could not be applied before.

3.5.6 Applying the Reductions

A straightforward way of computing a reduced graph given a set of Reduction Rules $\{r_1, ..., r_j\}$ used by Akiba and Iwata [3], is to iterate over all reductions and apply each rule r_i to all vertices. Whenever a reduction rule successfully reduces the graph, they reset to the first rule, such that once the last rule r_j does not reduce the graph, the kernel has been computed. As trying the reductions on all vertices in each iteration is expensive and most likely redundant, more targeted reduction schemes are researched as well. For instance, Chang et al. [14] compute and maintain a triangle count $\delta(u, v) \forall \{u, v\} \in E$ to decide whether to apply the domination reduction or not, since a vertex u only dominates its neighbor v, if $\delta(u, v) = d(u) - 1$. A lot of redundant computation can be avoided by only applying the domination reduction on v, if the triangle count in $N[v]$ changes.
Hespe et al. [24] conceive a more general strategy for targeted reductions called dependency checking. The idea of dependency checking is, that vertices whose neighborhood has not changed since last applying a reduction do not need to be rechecked by that same reduction. To avoid redundant computations, the authors therefore keep a set of viable candidate vertices D, which they initially set to $D = V$ and whenever a vertex is removed, update to $D = D \cup N(v)$. The reductions are finished once D is empty. They also propose an additional technique called reduction tracking, which is motivated by the fact that applying reductions exhaustively typically yields diminishing returns. For that reason, it is sometimes beneficial to stop applying a rule once its effect becomes small relative to the previously sampled rate of vertex removals to make room for more efficient reductions.
An important question is when to apply inexact reductions, as these have a high potential of speeding up the algorithm but at the same time a high risk of reducing the solution quality. For their vertex peeling reduction, Dahlum et al. [15] consider both removing vertices by absolute degree, i.e. vertices with a degree higher than a specific threshold, and by relative degree, where the highest-degree vertices are removed iteratively. In their solvers, the reduction is applied after a subset of the exact reductions, removing the top 1% of high-degree vertices and then running local search on the resulting graph. Chang et al. [14] set a higher priority on exact reductions, by only applying vertex peeling when no more exact reduction rules can be applied. They furthermore propose to try reapplying the exact reductions after a heuristic vertex removal.
After applying the graph reduction rules, one option is to switch to an optimal solver such as B&B to compute the final solution [32]. Another effective method is to interleave the reductions with a local search or memetic algorithm, where reductions are applied both initially and whenever more vertices are classified by the respective solver, since this likely opens up new reductions [22].

CHAPTER 4

Algorithms

This chapter covers both exact and heuristic algorithms for tackling the MWC problem. The exact algorithm applies a set of novel exact reduction rules to reduce the graph before applying a B&B solver to obtain the solution. The heuristic approach additionally applies vertex peeling using heuristic rules and machine learning to further speed up the computation. For the theoretical analysis, we assume that the graph is represented by an adjacency list A_{list} and adjacency lookups are possible in $\mathcal{O}(1)$ expected time using a hash map.

4.1 Exact Approaches

This section describes the techniques used in the exact algorithm, including the exact reduction rules, how to apply them, and how to solve the reduced graph.

4.1.1 Exact Reduction Rules

The number of reductions for MWC so far is very limited. Much more effort has been put into finding reduction rules for MWIS, some of which are highlighted Section 3. In the following subsections, novel reduction rules for the MWC that take inspiration from the previously seen MWIS reductions are presented and employed in the exact solver MWCRedu.

Neighborhood Weight Reduction

A simple but effective reduction often seen in literature [11, 12, 26, 25, 47] is based on the upper bound of $v \in V$, which is given as $w(N[v])$.

Reduction Rule 1. *Let $v \in V$ have $w(N[v]) \leq w(\hat{C})$, where $\hat{C}$ is the largest weight clique found so far. Then v cannot be part of a clique with a weight larger than that of $\hat{C}$ and v can safely be removed from the graph without reducing the maximum solution weight.*

The rule can be applied on a vertex $v \in V$ in $\mathcal{O}(1)$, given that the neighborhood weight is stored and maintained throughout the reductions.

Largest Neighbor Reduction

Another reduction rule introduced by Cai et al. [11] tightens Rule 1 by either including or excluding the highest weight vertex n^* in the Neighborhood.

Reduction Rule 2. *Let $v \in V$ and its highest weight neighbor n^*, s.t. $\max(w(N[v]) - w(n^*), w(v) + w(n^*) + w(N(v) \cap N(n^*))) \leq w(\hat{C})$. Then v cannot be part of a clique with a weight larger than that of $\hat{C}$ and v can safely be removed from the graph without reducing the maximum solution weight.*

For applying the rule on a vertex $v \in V$, first its highest weight neighbor is identified in $\mathcal{O}(d(v))$ and then the intersection of their neighborhood is computed in $\mathcal{O}(\min(d(v), d(n^*))$, resulting in overall $\mathcal{O}(d(v))$.

Twin Reduction

The twin reduction for MWIS (Section 3.5.1) works by the argument, that the two vertices u and v are either both or neither in the solution. For MWIS, this requires the vertices to be non-adjacent, since they could otherwise not be in the same solution. In order to be applicable to the MWC, the reduction is altered slightly, while keeping the notion of twins intact. Specifically, v and u are required to be adjacent and to share the same neighborhood, such that if either one of them is in the solution, the other one may always be added as well. An illustration of the reduction for the MC is shown in Figure 4.1.

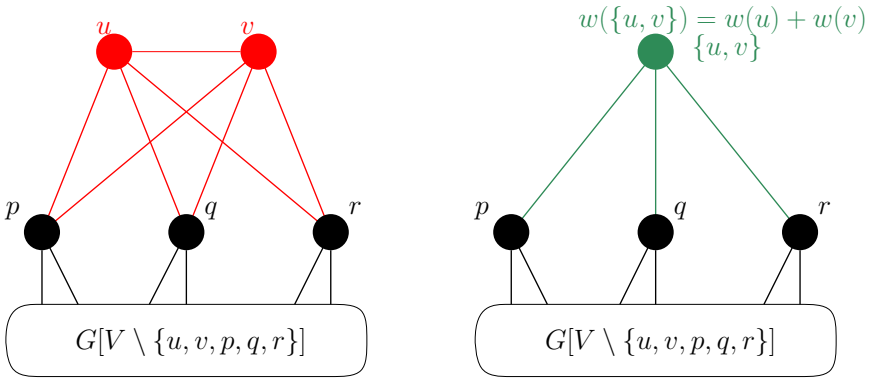

Figure 4.1: Twin reduction for MWC

Reduction Rule 3. *Given two vertices v and u, s.t. $\{v, u\} \in E$ and $N(u) = N(v)$, the vertices can be contracted to a new vertex v' with weight $w(v') = w(v) + w(u)$ and $N(v') = N(v) = N(u)$ without reducing the maximum solution weight.*

Proof. Suppose there is an optimal solution C^* that, without loss of generality, contains u, but not v. Then it is always possible to add v to the solution, as it is connected to all neighbors of u, resulting in a higher weight solution. There can therefore be no optimal solution that contains u and not v and vice versa. □

The only computation that is needed to evaluate two vertices $v, u \in V$ is the intersection of their neighborhoods, which can be computed in $\mathcal{O}(d(v))$. Note that the computation can be skipped if $d(v) \neq d(u)$.

Domination Reduction

Given the case that $N(v) \subseteq N(u)$ for $v, u \in V$, the domination reduction for MWIS (Section 3.5.2) removes the dominating vertex u if it has a weight $w(u) \leq w(v)$, as it excludes strictly more vertices if it is included in the solution than v. In the context of MWC on the other hand, a maximal clique containing u with $w(u) \geq w(v)$, would have a weight greater or equal to one including v. Given the two cases, where v and u are either adjacent or non-adjacent, two different reduction rules can be applied. In the case, that they are non-adjacent and the dominating vertex u has weight $w(u) \geq w(v)$, it is possible to remove v.

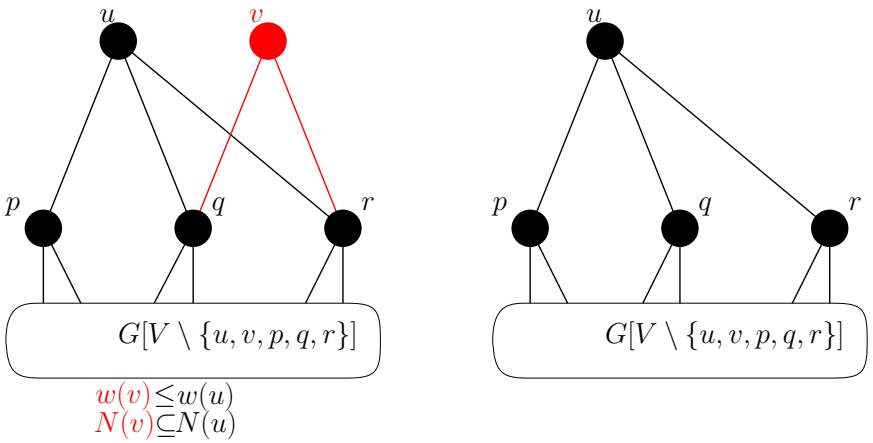

Figure 4.2: Domination reduction (Case 1) for MWC

Reduction Rule 4. *Given two vertices v and u, s.t. $\{v, u\} \notin E$, $N(v) \subseteq N(u)$ and $w(u) \geq w(v)$, vertex v can be removed from the graph without reducing the maximum solution weight.*

Proof. Suppose there is an optimal solution C^* that, without loss of generality, contains v, but not u. Then it is always possible to substitute v with u in the solution, as u is connected to all neighbors of v, resulting in a solution with at least the same weight. There is therefore at least one optimal solution that does not contain v. □

As the vertices v and u are non-adjacent, u is initially not known. Therefore, the common intersection of $N(v)$ is computed, in order to find candidates for u, where only vertices w s.t. $w \neq v$, $d(w) \geq d(v)$, $w(w) \geq w(v)$ and $\{w, v\} \notin E$ are considered as candidates. This can be done in $\mathcal{O}(d(v)|V|)$, since each neighbor could be connected to all other vertices in the worst case. For each candidate, the neighborhood intersection with v is computed in $\mathcal{O}(d(v))$. This computation can be stopped prematurely, if a vertex w s.t. $w \in N(v)$ and $w \notin N(u)$ is found. Overall, the time complexity of applying the reduction on a vertex v is $\mathcal{O}(d(v)|V|)$.

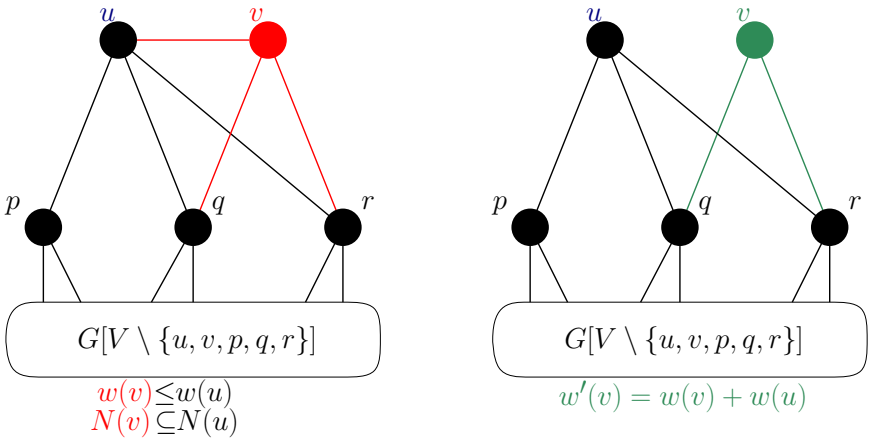

Figure 4.3: Domination reduction (Case 2) for MWC

In the case that they are adjacent, simply removing v is not possible, since it may be part of a clique containing u. It is however possible, to add the weight of u to v and then remove the edge $\{u, v\}$, preserving the best solution achievable by v and u being in the same clique while reducing the graph at the same time.

Reduction Rule 5. *Given two vertices v and u, s.t. $\{v, u\} \in E$ and $N(v) \subseteq N(u)$, the edge $\{u, v\}$ can be removed from the graph after updating $w(v)$ to $w'(v) = w(v) + w(u)$ without reducing the maximum solution weight.*

Proof. Equivalence must be shown for two cases. For the first case, without loss of generality, suppose the optimal solution C^* in the original graph contains both u and v. Then $w(C^*) \leq w(u) + w(v) + w(N(v))$, which is identical to $w'(v) + w(N(v))$ in the reduced graph. If C^* in the original graph contains u but not v, its weight is bounded by $w(u) + w(N(u))$, which is identical in the reduced graph. The case that C^* in the original graph contains v but not u cannot occur, since any solution containing only v can always be enlarged by adding u, since u is connected to all neighbors of v. □

Similar to the twin reduction, only the intersection of the neighborhoods of v and u is required to evaluate the reduction for the two vertices, resulting in a time complexity of $\mathcal{O}(d(v))$. Two vertices are only evaluated, if $d(v) < d(u)$, and the computation can be stopped, as soon as a vertex w s.t. $w \in N(v)$ and $w \notin N(u)$ is found.

Simplicial Vertex Removal Reduction

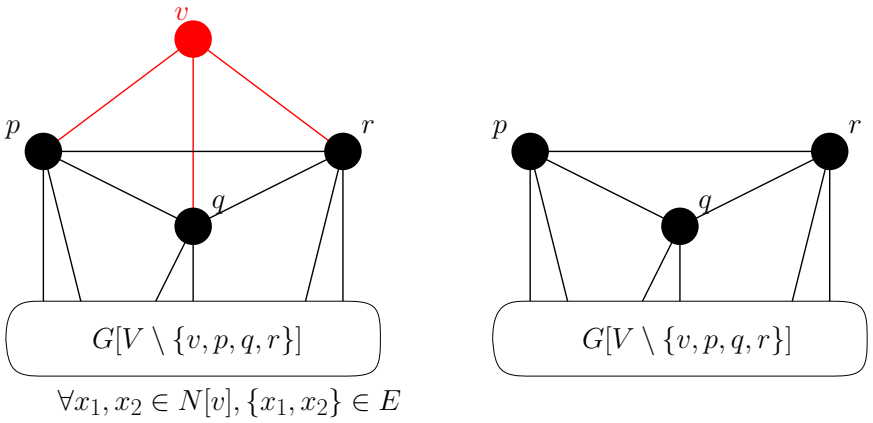

Figure 4.4: Simplicial vertex removal reduction for MWC

As is the case for MWIS (Section 3.5.3), simplicial vertices may be removed before applying the MWC solver as well. This can be done, since once a simplex v has been identified, the largest clique C it can be part of is known and can be evaluated as $w(C) = w(N[v])$. If the weight is larger than the current largest known clique, the lower bound is updated.

Reduction Rule 6. *Let $v \in V$ s.t. $\forall x_1, x_2 \in N[v], \{x_1, x_2\} \in E$. Then if $w(N[v]) > w(\hat{C})$, the solution is updated to $\hat{C} = N[v]$. After that, v can be removed from the graph without reducing the maximum solution weight.*

Proof. Correctness must be shown for two cases. Either the clique formed by $N[v]$ is an improved solution, i.e. $w(N[v]) > w(\hat{C})$, or it is not, i.e. $w(N[v]) \leq w(\hat{C})$. In the first case, the solution is updated to $\hat{C} = N[v]$. $w(N[v]) \leq w(\hat{C})$ now holds in both cases and indicates, that v cannot be part of an improved solution. □

Since the adjacency between each pair of vertices in the neighborhood of v needs to be confirmed, the computation takes $\mathcal{O}(d(v)^2)$. The computation is stopped, as soon as a vertex $w \in N(v)$ that is not connected to all other vertices in $N(v)$ is found.

Edge Bounding Reduction

The Edge Bounding Reduction is a natural extension to Reduction Rule 2, using the computed bounds not only to decide whether a vertex can be removed, but also the edge connecting it with its highest weight neighbor. Given a vertex $v \in V$ and its highest weight neighbor $n^* \in N(v)$, let ub_{inc} denote the including upper bound $w(v) + w(n^*) + w(N(v) \cap N(n^*))$ and ub_{ex} the excluding upper bound $w(N[v]) - w(n^*)$. The reduction implemented in Rule 2 states, that v can be removed if both $ub_{inc} \leq w(\hat{C})$ and $ub_{ex} \leq w(\hat{C})$ hold. The extension provided by the edge bounding reduction is based

Reduction Rule	Time Complexity	Application
1	$\mathcal{O}(1)$	$v \in V$
2	$\mathcal{O}(d(v))$	$v \in V$
3	$\mathcal{O}(d(v))$	$v \in V$ and $u \in N(v)$, s.t. $d(v) = d(u)$
4	$\mathcal{O}(d(v)\|V\|)$	$v \in V$
5	$\mathcal{O}(d(v))$	$v \in V$ and $u \in N(v)$, s.t. $d(v) < d(u)$
6	$\mathcal{O}(d(v)^2)$	$v \in V$
7	$\mathcal{O}(d(v))$	$v \in V$
7(Extended)	$\mathcal{O}(\min(d(v), d(u)))$	$v \in V$ and $u \in N(v)$

Table 4.1: Overview of the time complexity of exact reduction rules for MWC.

on the observation, that in the case that $ub_{ex} > w(\hat{C})$ but $ub_{inc} \leq w(\hat{C})$, it is possible to remove the edge connecting n^* and v.

Reduction Rule 7. *Given the vertices v and n^*, s.t. $\{v, n^*\} \in E$, if $ub_{inc} \leq w(\hat{C})$, the edge $\{v, n^*\}$ can be removed from the graph without reducing the maximum solution weight.*

Proof. ub_{inc} is an upper bound on the weight of any clique containing both v and n^*. If a clique $\hat{C}$ with weight $w(\hat{C}) \geq ub_{inc}$ is known, then there is at least one optimal solution C^* that does not contain both v and n^*. The edge $\{v, n^*\}$ is thus irrelevant in the search for a higher weight solution. □

The time complexity is identical to that of Rule 2: $\mathcal{O}(d(v))$. Another extension can be made by branching not only on the highest weight neighbor, but on all neighbors. The proof still holds since the branching procedure does not depend on specific weight configurations. The time complexity for applying the reduction on a pair of vertices v and u then becomes $\mathcal{O}(\min(d(v), d(u)))$.
Table 4.1 gives an overview of the exact reduction rules for MWC, where the first column indicates the reduction number, the second column the time complexity of applying the reduction once and the third column the set of eligible vertices.

4.1.2 The Exact Solver

The exact solver presented in this work is denoted as MWCRedu. It works in two stages: First, the set of exact reduction rules from Section 4.1.1 is used to reduce the graph. Once the reductions terminate, the reduced graph is passed to an exact B&B solver to compute the final solution.

Algorithm 3 Computing the lower bound

```
U ← V
for i := 1 to |U| do
    v_i ← min_{v∈U}(d(v))
    if d(v_i) = |U| − 1 then
        C_0 ← U
        return C_0
    end if
    U ← U \ v_i
    for u ∈ N(v_i) do
        d(u) ← d(u) − 1
    end for
end for
```

Computing a Lower Bound

Reduction Rules 1, 2 and 7 compute an upper bound and thus depend on an initial solution to remove vertices. For computing bounds, fast heuristics are generally preferred, since spending more time on improving the initial solution typically gives diminishing returns. A well suited heuristic for computing an initial lower bound is the one employed in WLMC [26]. As shown in Algorithm 3, the algorithm works by repeatedly removing the vertex with the smallest vertex degree from the graph, until after $i-1$ vertices have been removed, all remaining vertices are adjacent to each other and form the initial clique C_0. The initial lower bound is then given as $w(C_0)$.
During the reduction phase, the initial solution is continuously improved both by Rule 6 and the local search algorithm from FastWCLq [12], the latter being applied on the reduced graph in between checking each reduction rule. The local search algorithm starts the clique construction by adding a single vertex from a set of starting vertices, which is regulated such that each vertex is used at least once, and enlarges the clique using BMS. Specifically, a number of random vertices from the set of candidates are evaluated and the best one according to the following benefit estimation function is added to the clique.

$$\hat{b}(v) = \frac{2\,w(v) + w(N(v) \cap CandSet)}{2} \tag{4.1}$$

After no more vertices are in the candidate set, further improvements are attempted on the given clique C using a lightweight B&B algorithm (Section 2.3): For each vertex $v \in C$, the algorithm computes the maximum weight clique C'_v among the new candidates of $C \setminus v$. If $w(C'_v) > w(v)$, an improved solution is found and C is updated to C'_v accordingly. The B&B algorithm is implemented using a coloring-based upper bound. Given a valid vertex coloring of $G[P]$ into the color classes $\Pi = \{D_1, D_2, ..., D_{|\Pi|}\}$, the upper bound can be computed as $\sum_{j=1}^{|\Pi|} w^*(D_j)$. The coloring is computed by iterating the candidate vertices in descending order of weight, tie-breaking by degree, and assigning the smallest color class

to each vertex that is unique among its neighbors. If all existing color classes are already used in a neighborhood, an additional class is introduced until all candidate vertices are colored [12].
The best solution found by the initial heuristic, local search or the simplex reduction is then used as a lower bound in the reduction rules 1, 2 and 7. It is furthermore used as an initial solution for the solver that is applied on the reduced graph.

Applying the Reductions

For applying the exact reduction rules proposed in Section 4.1.1, an adapted version of the strategy from Hespe et al. [24] (Section 3.5.6) that entails both dependency checking and reduction tracking is used. Specifically, the reductions $\{r_1, ..., r_7\}$ are iterated with each rule r_i being tried on its set of viable vertices D_i, which is initially set to $D_i = V$. Every time a rule fails to reduce a vertex, that vertex is removed from the set of viable candidates D_i, whereas otherwise, the sets of all rules are updated to $D_i = D_i \cup N(v)$ for $i = \{1, ..., 7\}$ and the applicable vertices or edges are removed from the graph. This way redundant computations are minimized without affecting the final kernel size [24].
Slightly different from the original strategy, reduction tracking is implemented by pausing a reduction, once it fails to achieve a reduction rate of at least 1 % of the current number of vertices or edges per second, until another reduction reduces the graph by that amount. Reduction tracking is checked both in-between applying different reduction rules and periodically during the iteration of candidate vertices, in order to prevent single reductions to delay the solver and allow either more efficient reductions or the exact solver to take over. Another addition to the strategy by Hespe et al. is to set a dynamic limitation on the degree of vertices that are tried in the reductions. The limit is set to 10 % of the highest degree initially and is increased by 10 % whenever the reductions have been exhaustively applied in the previous level. This guarantees, that reductions applicable on low degree vertices, which are typically more efficient, are applied first. The loop terminates once the degree is no longer limited and all reductions are paused, at which point the reduced graph is passed to the exact solver to compute the final result.

Solving the Reduced Graph

The reduced graph is solved using the B&B paradigm, which is guaranteed to output the optimal solution given enough time, by considering all valid combinations of vertices as a solution. As the procedure has exponential time complexity, it is important to choose a good ordering and to reduce the set of branching vertices by computing tight upper bounds. The ordering is computed as in WLMC as $v_1 < v_2 < ... < v_{|V|}$, where v_1 has the smallest vertex degree, v_2 has the smallest vertex degree after v_1 is removed, etc [26].
For computing tight upper bounds and reducing the set of branching vertices, efficient IS- and MaxSAT-based approaches from related work are applied throughout the search. As in the B&B solver from Section 4.1.2, the VC heuristic is used to obtain an upper bound

$ub = \sum_{j=1}^{|\Pi|} w^*(D_j)$, where the color classes $\Pi = \{D_1, D_2, ..., D_{|\Pi|}\}$ form the ISs [26]. The set of branching vertices is then further reduced by using the two-stage MaxSAT reasoning approach from TSM-MWC [25]. In the first stage, which the authors refer to as binary MaxSAT reasoning, the set of branching vertices is reduced by inserting as many vertices as possible into the ISs s.t. $\sum_{j=1}^{|\Pi|} w^*(D_j) \leq w(\hat{C})$. As these vertices cannot form a clique with a weight larger than $w(\hat{C})$ by themselves, they can be removed from the set of branching vertices. If a vertex $v_i \in V$ has neighbors in all existing ISs but $ub + w(v_i) \leq w(\hat{C})$ holds, it is inserted as a new IS. Otherwise it is attempted to split its weight among ISs that do not contain any of its neighbors, by adding v_i with weight $w^*(S_j)$ into IS S_j and updating the weight to $w(v_i) = w(v_i) - w^*(S_j)$ for $j = 1, 2, ..., k$, until its remaining weight is given as $\delta = w(v_i) - \sum_{j=1}^{k} w^*(S_j)$. If $\delta > 0$ and $ub + \delta \leq w(\hat{C})$, v_i is inserted as a new IS with weight δ, otherwise the weight splitting procedure is undone and v_i is kept in the set of branching vertices. In the second stage, called ordered MaxSAT reasoning, the set of branching vertices is reduced further, by detecting disjoint conflicting subsets of ISs. Firstly, the weight of a branching vertex v_i is again split among the ISs $\{S_1, S_2, ..., S_k\}$ that do not contain any of its neighbors, resulting in the remaining weight $w(v_i) = \delta > 0$, since the vertex was not removed from the set of branching vertices in the first stage. After that, the algorithm tries to find a set of ISs $\{U_1, U_2, ..., U_r\}$ that each contain exactly one neighbor u of v_i. It then looks for an IS D_q s.t. $D_q \cap N(v_i) \cap N(u) = \emptyset$ for any U_j, proving that the sets $\{\{v_i\}, U_j, D_q\}$ are conflicting. In this case, ub can be further improved to $ub + \delta - \beta$, where $\beta = \min(\delta, w^*(U_j), w^*(D_q))$ [25].
Finally, if after considering all $U_j \in \{U_1, U_2, ..., U_r\}$ ub is still higher than the lower bound, ub is reduced by identifying conflicting subsets via unit propagation as first implemented for MWC in MWCLQ [17]. Unit propagation works from the idea that clauses with more literals are more likely to be satisfied and are thus considered *weaker* clauses. A unit clause is thus the strongest clause since it only has one possibility of evaluating to $true$. The algorithm repeatedly satisfies such a clause, removing all occurrences of the contained literal from the other clauses. If an empty clause remains, the set of clauses is identified as conflicting. Each time a set of conflicting clauses $\{S_0, S_1, ..., S_r\}$ is identified, the upper bound can be reduced by $\delta = \min(w^*(S_1), ..., w^*(S_r))$. To tighten the bound further, each S_j ($0 \leq j \leq r$) is split into S'_j and S''_j so that $w^*(S'_j) = \delta$ and $w^*(S''_j) = w^*(S_j) - \delta$. S'_j then represents the conflicting subset found so far, whereas further conflicts can be deduced from S''_j [17].
The procedure is run at every branch of the solver in order to reduce the amount of work to be done. The algorithm terminates, when all branches are either explored or pruned, i.e. the optimal solution is found, or when the time limit is reached, in which case the best solution found at that point is reported.

4.2 Heuristic Approaches

As will be evident in Section 6, there are graph instances for which the exact reduction rules by themselves are not sufficiently effective. Furthermore, in real world scenarios where the optimal solution is not known, it is often the case that getting a good approximation fast is preferable over taking a long time to find the optimal solution. For this reason, we investigate vertex peeling techniques similar to the ones employed by Dahlum et al. [15] for MIS (Section 3.5.5). Vertex peeling denotes the technique of removing the vertices from the graph, that are assigned the lowest scores by some heuristic rule. This rule must therefore capture the likelihood of a vertex belonging to the solution as well as possible. As for MIS, using the vertex degree is the obvious choice for MC as well, since a vertex with a high degree is more likely to form a large clique. Specifically $d(v)$ gives an upper bound on the size of the clique v can be part of. For the measure to remain an upper bound in the context of MWC, the weight of the neighborhood of each vertex is taken into account. The resulting simple and intuitive scoring measure $w(N[v])$ is used in the first peeling rule, Peel_{UB}.
Another promising approach is to train a machine learning model (Section 2.5) to predict the likelihood of a vertex belonging to the solution based on some input features. Graphs for training the model are openly available in online graph repositories from real world problems [16] or from random graph generators such as KaGen [18]. Since labels can be obtained by solving easier graphs to optimality using an exact algorithm such as TSM-MWC [25], supervised learning algorithms are the obvious choice. In order for any ML model to infer a score representing the potential of a vertex to be in the optimal solution, both information about the weight distribution in the neighborhood and about the neighborhood connectivity are required. Since the main goal of a heuristic is to find a good solution fast, the effect of each feature on the accuracy of the model needs to be balanced with its computational cost.

4.2.1 Vertex and Graph Features

Obvious choices for model features as seen in previous work [44] are the vertex weight, degree and neighborhood weight, as these features can be aggregated while reading the graph and maintained in linear time in case of a vertex removal. A feature that is more expensive to compute but gives valuable information about the neighborhood connectivity, is the local clustering coefficient (LCC). The LCC is defined as the number of edges in the neighborhood of a vertex divided by the total possible number of edges and thus gives a measure of how close the neighborhood is to being a clique. It can be computed for each vertex $v \in V$ in $\mathcal{O}(d(v)^2)$, by checking the adjacency between each pair of vertices in $N(v)$ and applying Equation 4.2.

$$LCC(v) = \frac{2 \mid \{u, w \in N(v) \mid \{u, w\} \in E\} \mid}{d(v)(d(v) - 1)} \tag{4.2}$$

Another measure to quantify communities in a graph can be obtained from running a semi-supervised clustering algorithm such as label propagation on the input graph. Label propagation works by first initializing the labels such that each vertex has a unique label. The algorithm is then executed either until termination, or until ℓ passes have been made. In each pass, the vertices are iterated and the label of the current vertex is updated to the label occurring most frequently in its neighborhood. After a few iterations of the algorithm, the labels of the vertices should give information on their connectivity, since vertices with more common labels are more likely to be in large communities. The running time is $\mathcal{O}(\ell|E|)$.
Given a selection of vertex features, it is important for the model to generalize to different types of graphs. If two vertices differ significantly in value for specific features, e.g. because of different weight distributions, the model will likely over-fit on the higher valued features as a result. To tackle this issue, all input features of a given graph are divided by the respective maximum occurring value, except for the LCC, which is already constrained to [0,1]. Furthermore, graph features can be used to relativize the given vertex feature values. Such features include the graph density and the graph clustering coefficient (GCC). The GCC measures the graph connectivity by computing the number of triangles divided by the number of all triplets, meaning sets of three nodes that are connected by two or three edges. It can be computed as the average of all LCCs in the graph in $\mathcal{O}(|V|)$, if the LCCs are already known. Optimally, this should lead the ML model to not only interpret the given vertex features as absolute values, but to evaluate them in the context of the entire graph, such that a vertex in a dense graph needs a higher local connectivity than a vertex in a sparse graph to be assigned a high score.

4.2.2 Machine Learning Model Architectures

From the wide range of existing supervised machine learning models, four models are trained and evaluated in this work: Two MLP models, a DeepSet-based approach and a GNN model.

Multi-Layer Perceptron Architecture

The first model $\text{Peel}_{\text{MLPfast}}$ uses MLP layers (Section 2.5.1) and only basic features, i.e. vertex weight, vertex degree, vertex neighborhood weight and graph density, making it possible to run inference frequently and on large graph instances. The model is visualized in Figure 4.5. A second model $\text{Peel}_{\text{MLPfull}}$ uses the LCC, cluster size and GCC in addition to the features employed in $\text{Peel}_{\text{MLPfast}}$. Both models are made up of the input layer, followed by three hidden layers with 300, 100 and 30 neurons respectively, and the output layer. Non-linearity is applied on the first three layers using the ReLU function and on the last layer using the sigmoid function to allow the output to be interpreted as a probability.

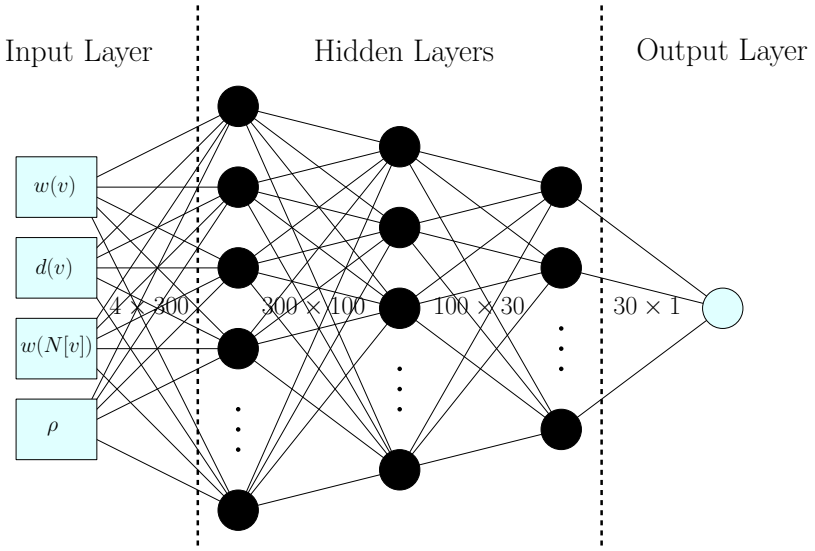

Figure 4.5: MLP model

DeepSet Architecture

The DeepSet architecture [54] (Section 2.5.2) is interesting in this application, since it enables the use of arbitrarily sized input sets as features for an ML model. The model $\text{Peel}_{\text{DeepSet}}$ makes use of this, by using the weights of the inclusive neighborhood of each vertex as the main input feature. First, given some vertex $v \in V$, the inner MLP ϕ is applied on $w(v)$ and $w(u)\ \forall\ u \in N(v)$. The outputs are then summed up and used as an input feature for the outer MLP ρ. The idea is, that the model should be able to learn any function on the weight distribution of each vertex this way. As the weights itself do not give information on the connectivity, the vertex degree, cluster size, LCC, GCC and graph density are added as input to ρ as well. To allow efficient batching techniques without requiring too much RAM space, the input set of ϕ is padded or trimmed to a fixed size of 100. Both neural networks ϕ and ρ follow the same structure as the one employed in $\text{Peel}_{\text{MLPfast}}$ and $\text{Peel}_{\text{MLPfull}}$.

Graph Neural Network Architecture

Another natural choice for addressing graph optimization problems is GNN (Section 2.5.3), an ML framework designed for graph-like structures. GNN layers have access to the adjacency matrix, which they use to propagate messages, i.e. vertex features, within neighborhoods. Since information about the graph connectivity is already contained in the adjacency matrix, the only feature that should be required is the vertex weight. From the various types of GNN layers in literature, GIN [53] is chosen, as it is best suited for identifying specific structures in the neighborhood of a vertex, which intuitively should be an advantage when looking for large interconnected clusters.

The message passing is implemented as

$$h_v^{(t)} = MLP^{(t)} \left(h_v^{(t-1)} + \sum_{u \in N(v)} h_u^{(t-1)} \right) \tag{4.3}$$

where $h_v^{(t)}$ is the feature vector of $v \in V$ at time t and the MLP follows the same structure as in the MLP-based strategies. The ML model is made up of one GIN layer, since the most relevant information should be contained in the 1-neighborhood of each vertex. The resulting strategy is denoted as Peel_{GNN}.

4.2.3 Training the Machine Learning Model

For training the models, 220 graphs from `openstreetmap.org`, the second DIMACS challenge [27] and the SuiteSparse matrix collection [16] are chosen, that can be solved within less than a second by TSM-MWC. While training on harder instances would likely be more fruitful, obtaining high quality labels for these would be difficult. The idea is, that the model learns to identify high quality vertices based on a local graph structure, which should generalize to harder instances as well. After labeling the graphs such that the label represents the potential value of each vertex in a weighted clique, the models are trained and implemented into the solver.

Procuring the Labels

While GNN & LS [33] achieves a high accuracy of around 80% for MWVC by simply assigning binary labels to the vertices depending on whether they are in the solution or not and then applying classification, this approach is less promising for MWC. That is because for MWVC, the number of vertices in and out of the solution is approximately equal, assuming a typical power-law distribution w.r.t. the degrees. For MWC however, being applied on the complement graph, the same labeling strategy would result in a highly unbalanced class distribution, which is hard to learn by a supervised ML model. For this reason, an alternative labeling strategy is proposed as shown in Algorithm 4. Each vertex is assigned a value in $[0, 1]$, denoting the highest weight clique it can potentially be part of divided by the weight of the optimal solution. The largest weight clique containing a specific vertex v can be computed with any exact MWC solver, by first increasing the weight of the respective vertex by a sufficiently large constant to $w'(v) = w(v) + c$, computing the optimal solution (which is guaranteed to contain v due to the increased weight) and finally reverting the weight augmentation. The labels are then obtained after running the solver n times as $label(v) = \frac{w'(\hat{C}_v)}{w'(\hat{C})}$, where $w(\hat{C}) = \max_{v \in V}(w(\hat{C}_v))$. The advantage of this labeling strategy is not only that the class distribution is more balanced, as every vertex receives a label greater than zero, but also that vertices contained in high weight cliques, that would have been assigned the label 0 otherwise, are also assigned a high label. This property is

Algorithm 4 Computing the labeling

for $v \in V$ **do**
  $w(v) \leftarrow w(v) + c$
  $\hat{C}_v \leftarrow$TSM-MWC($G[V]$)
  $label[v] \leftarrow w(\hat{C}_v) - c$
  $w(\hat{C}) \leftarrow \max(w(\hat{C}), w(\hat{C}_v) - c)$
  $w(v) \leftarrow w(v) - c$
end for
for $v \in V$ **do**
  $label[v] \leftarrow \frac{label[v]}{w(\hat{C})}$
end for
return $label$

especially useful for heuristic algorithms, since they aim to find any high quality solution quickly.

Training Procedure

The training itself is done in a similar manner for each model. First, the labeled graphs are separated by a 90%-10% split into training- and validation data, where the model only directly learns from the training data and the validation data serves the purpose of evaluating how well the model generalizes to data it has not seen before, as will be the case in practice. The models are then trained for 25 epochs, where one epoch is finished after all batches of the training set have been passed to the model once. The batch size is chosen as 50,000 vertices for the MLP and DeepSet models and 10 graphs for the GNN models. After each epoch, the average score the model assigns to vertices with the label 1 in the validation set is compared with the average score assigned to all other vertices in the validation set to indicate the performance of the model. The models achieving the highest difference between the two values are taken into consideration for inference later on.

4.2.4 The Heuristic Solver

In this section, the heuristic MWC solver MWCPeel is introduced. The solver works similar to MWCRedu described in Section 4.1.2, but implements the peeling reduction on top of the previously introduced exact reductions. The peeling rule that is used for the heuristic reduction is determined in Section 6, by comparing the different peeling rules and choosing the best one. The remainder of this section discusses how to use the scores assigned to each vertex to reduce the graph and when to stop the heuristic reductions and pass the reduced graph to the B&B solver.

Peeling Strategy

A straightforward approach is to run inference on the entire graph and remove the vertices with the lowest score. Similarly to the approach used by Dahlum et al. [15], a percentage of the currently remaining number of vertices is removed in each step, in order to minimize the overhead of running inference on the entire graph. The number of vertices to be removed in one step num is dynamically determined as follows:

$$num = \begin{cases} 10\,\% \cdot |V|, & \text{if } |V| > 50{,}000 \\ \max(1\,\%, 10\,\% \cdot \frac{|V|}{50{,}000}), & \text{otherwise.} \end{cases}$$

The differentiation between larger and smaller graphs is made since the exact reductions would otherwise often be reapplied on many vertices, which would significantly slow down the solver. Furthermore, as the degree often follows a power-law distribution w.r.t. the degrees in real-world graph instances [23], the size of the optimal solution makes up a smaller portion of the graph for large graphs. After each peeling step, the viable candidate sets are updated and exact reductions are re-evaluated.

Stopping Criteria

Another important decision is when to stop applying the peeling reduction; stopping too early could result in a much higher running time for the solver applied on the reduced graph, whereas stopping late might negatively impact the solution quality. Since the optimal amount of vertices to reduce is highly dependent on the graph structure, a static stopping criterion is unlikely a good strategy. For this reason, a dynamic strategy is employed, that works by comparing the current computed score with previously computed scores. The first stopping criterion is the deterioration of the maximum score value below a certain threshold relative to the total maximum score value, as this indicates that the peeling reduction begins to reduce the maximum solution. A second stopping criterion takes effect, when the difference between the minimum and maximum score shrinks below a certain threshold, as this shows, that the scoring model can no longer clearly distinguish high quality vertices from low quality vertices. Both thresholds are chosen as 90% to achieve a good balance between speed-up and solution quality. As a fail-safe, a backup of the current graph state is created before applying the heuristic reduction, which can be reloaded in the case the graph is reduced to zero. Following the reduction procedure, the B&B solver is applied on the reduced graph to obtain the final result.

CHAPTER 5

Implementation

While the previous chapter focused on the theoretical description and the general idea of the algorithms, this chapter describes the implementation from a practical point of view.

5.1 Implementing the Exact Approaches

As the solver should be able to process even huge graphs, the space requirement has to be taken into account when choosing the data structure for representing the graph. The minimum space requirement of $\mathcal{O}(|V| + |E|)$ can be achieved by using a simple adjacency list A_{list}. Using sequential search, checking whether two vertices are adjacent takes $\mathcal{O}(\min(d(v), d(u)))$. This poses a challenge for computing common neighborhoods (Rule 7), dominating neighborhoods (Rule 3,4,5) and interconnectivity (Rule 6) efficiently. The naive algorithm to compute the intersection of two neighborhoods, which simply checks the adjacency of v with each of its neighbors u, is quadratic in the degree, making it infeasible for most larger graph instances. One way to speed up these computations would be to make sure the adjacency list is always sorted, which would lead to a time complexity that is logarithmic in the degree. This is however not an option since the deletion of vertices itself would become quite costly. The following approaches give a much better runtime for the given reductions.

5.1.1 Efficiently Computing Common Neighborhoods

An efficient way to compute the common neighborhood of two adjacent vertices v and u, used by Cai et al. [11] in FastWCLq to check Reduction Rule 2, uses a boolean indicator list of size $|V|$. In a first iteration, all neighbors of v are marked as $false$. After that, the neighbors of u are marked as $true$. Finally, all vertices $w \in N(v)$ where the indicator list evaluates to $true$ form the common neighborhood of v and u. Assuming, that the list is

Algorithm 5 Computing the common neighborhood of v and u, where $d(v) \leq d(u)$.

```
function COMMON_NEIGHBORHOOD(v, u)
    intersection = { }
    for w ∈ N(v) do
        indicator[w] ← false
    end for
    for w ∈ N(u) do
        indicator[w] ← true
    end for
    for w ∈ N(v) do
        if indicator[w] then
            intersection ← intersection ∪ {w}
        end if
    end for
    return intersection
end function
```

Reduction Rule	Time Complexity	Application
1	$\mathcal{O}(1)$	$v \in V$
2	$\mathcal{O}(d(v) + d(n^*))$	$v \in V$
3	$\mathcal{O}(d(v) + d(u))$	$v \in V$ and $u \in N(v)$, s.t. $d(v) = d(u)$
4	$\mathcal{O}(d(v)\|V\|)$	$v \in V$
5	$\mathcal{O}(d(v) + d(u))$	$v \in V$ and $u \in N(v)$, s.t. $d(v) < d(u)$
6	$\mathcal{O}(d(v)^2)$	$v \in V$
7	$\mathcal{O}(d(v) + d(n^*))$	$v \in V$
7(Extended)	$\mathcal{O}(d(v) + d(u))$	$v \in V$ and $u \in N(v)$

Table 5.1: Overview of the time complexity of exact reduction rules for MWC.

already instantiated, the algorithm (Algorithm 5) achieves a runtime of $\mathcal{O}(d(v) + d(u))$, where v is the vertex with the smaller degree [11].
The time complexities of the reduction rules are updated as shown in Table 5.1.

5.1.2 Efficiently Computing Dominating Neighborhoods

As mentioned in the beginning of Chapter 4, adjacency lookups may be improved to $\mathcal{O}(1)$ in expectation by using a hash-table. Since the intersection can already be computed efficiently however, a data-structure specialized in improving the performance on computing dominating neighborhoods and interconnectivity is sufficient. While the dominating neighborhoods can of course be computed by Algorithm 5 as well, this would not make good use of the easier nature of the problem: That the computation for a pair of vertices may be

Algorithm 6 Computing, whether v is dominated by u.

```
function DOMINATING_NEIGHBORHOOD(u, v)
    for w ∈ N(v) do
        if u ≠ w and not bloom_filter.lookup(u, hash(w))
         or u ≠ w and not bloom_filter.lookup(w, hash(u)) then
            return false
        end if
        indicator[w] ← false
    end for
    for w ∈ N(u) do
        indicator[w] ← true
    end for
    for w ∈ N(v) do
        if not indicator[w] then
            return false
        end if
    end for
    return true
end function
```

stopped early, as soon as it is clear, that neither vertex dominates the other. The Bloom filter (Section 2.4.1) is a natural extension for speeding up this computation. It is implemented by assigning a fixed sized hash table to each vertex. For each edge of the graph, the hashes of both vertices are computed and the corresponding flags are set in the hash-table of the respective other vertex. While checking whether two vertices v and u are adjacent, it is first checked whether $bloom\_filter.lookup(u, hash(v))$ or $bloom\_filter.lookup(v, hash(u))$ evaluate to $false$. In that case, the vertices are guaranteed to be non-adjacent and the computation can be stopped prematurely. The simple universal hash family shown in Section 2.4 is used as a hash function, since it is fast to compute and gives an even distribution. The size of the hash-tables is chosen relative to the respective vertex degree, such that the hash-table is filled to about 50%, which gives a good balance of space and time efficiency. Algorithm 6 shows how to compute whether v is dominated by its neighbor u using the Bloom filter. While the Bloom filter does not improve the theoretical running time of the reduction rules, it does accelerate them significantly in practice. Especially for Reduction Rule 6, which has a high time complexity of $\mathcal{O}(d(v)^2)$, the Bloom filter is able to stop most computations early, making Rule 6 very fast in practice.

5.1.3 Applying the Reductions

While optimally all reductions would be applied on the input graph instance exhaustively, this is not always the best approach in practice, as some reduction rules take a lot of time

for large graphs. It is therefore important to first reduce the graph by using the fastest and most effective reduction rules and only resort to slow reductions when the former have been applied exhaustively and the graph size has been sufficiently reduced. In some cases, where the graph size is still too large, it is even better to pass the reduced graph to the exact solver early instead.
Among the reductions presented in Section 4.1.1, the fastest reduction is clearly Rule 1 (Bounding Rule 1), since it runs in $\mathcal{O}(1)$. It is followed by Rule 3 (Twin) and Rule 6 (Simplex), which run in $\mathcal{O}(d(v) + d(u))$ and $\mathcal{O}(d(v)^2)$ respectively, but are much faster in practice, given that it is often possible to stop the computation early. For Rule 7 (Edge Bounding Rule), the entire intersection needs to be computed, meaning that the computation cannot be sped up by the Bloom Filter. For this reason, only the highest weight neighbor of each vertex is considered above a vertex count of 50,000. For smaller graphs, or once the graph has been sufficiently reduced, the intersection is computed for all neighbors. Rule 4 (Domination 1) has the worst computational complexity and is thus only executed if the graph contains less than 50,000 vertices. The same limitation is set for Rule 5 (Domination 2), as it has the same time complexity for removing one edge that other rules have for removing a vertex along with all its adjacent edges. The reductions are therefore applied in the following order:

1. Rule 1: Bounding Rule 1
2. Rule 3: Twin
3. Rule 6: Simplicial Vertex Removal
4. Rule 7: Bounding Rule 2 & Edge Bounding Rule
5. Rule 4: Domination 1
6. Rule 5: Domination 2

5.2 Implementing the Heuristic Approaches

As the heuristic solver MWCPeel is an extension of the exact solver MWCRedu, the implementation specifications described in Section 5.1 mostly apply here as well. Some optimizations are made in the reduction strategy and the model features, in order to shift the focus of the solver more towards speed rather than solution quality. After that, implementation details of the machine learning models are given.

5.2.1 Applying the Reductions

The adjustments concern only the least time efficient reductions, i.e. Rules 4, 5 and 7. Specifically, Rules 4 and 7 are not run at all, since they take a disproportional amount of

time to process and are the least effective in reducing the graph. Rule 5 is limited to the fast variation, which only looks at the highest weight neighbor, regardless of graph size. This is because the highest weight neighbor is most likely to give a tight bound, that not only allows the removal of an edge, but of the vertex and all its adjacent edges, making it more efficient than the extended version of this reduction rule.

5.2.2 Computing the Features

Another optimization targets the vertex feature LCC, since computing it for a single vertex takes $\mathcal{O}(d(v)^3)$ with adjacency lookups taking $\mathcal{O}(\min(d(v), d(u)))$, which would also effect the computation of the GCC. The LCC is therefore computed using the approximation algorithm by Becchetti et al. [6]. The algorithm first assigns a random label to each vertex and identifies for each vertex the smallest label occurring in its neighborhood. It then checks for each edge $\{v, u\}$ if the minimum label at the endpoints is equal, and if so, increases the counters Z_v and Z_u. This procedure is repeated l times, where l can be varied to trade off time and solution quality. The estimation of the triangle count of a vertex v can finally be computed as $\frac{Z_v}{3l}$, which can be plugged into Equation 4.2 to compute the LCCs. The algorithm can therefore compute the LCC for all vertices in $\mathcal{O}(l|E|)$. The authors show, that the algorithm achieves a Pearson correlation coefficient of greater than 0.9 between the approximate- and the exact solution for $l \geq 20$.

5.2.3 Implementing the Machine Learning Models

The models presented in Section 4.2.2 are trained using PyTorch, a python library that provides implementations of most commonly used ML components, such as neural network layers and activation functions. The GNN layer is implemented using the python library PyTorch Geometric. After training a machine learning model in python, the model is serialized using the PyTorch function $torch.jit.script$, which compiles the model source code as TorchScript and returns a ScriptModule. In order to run inference directly from C++, the PyTorch C++ frontend LibTorch is used, which allows a model saved in the ScriptModule format to be de-serialized and deployed in C++ code. For GNN models, the torch-scatter and torch-sparse C++ APIs are required additionally. The score is finally obtained by computing the respective features for the current graph and passing them to the model.

CHAPTER 6

Experimental Evaluation

In this section, the effectiveness of the proposed techniques is shown on a broad selection of graph instances and compared with state-of-the-art MWC solvers.

6.1 Algorithms

The solvers presented in this work are the exact solver MWCRedu, which employs the reduction strategy described in Section 4.1 and the heuristic solver MWCPeel, which additionally employs vertex peeling as described in Section 4.2.4. For comparison, three solvers that can each be regarded as state-of-the-art are included in the experiments: TSM-MWC [25] (exact), FastWCLq [12] (heuristic) and SCCWalk4l [47] (heuristic). The hyper-parameters are chosen as recommended by the authors in the original papers: For FastWCLq, the minimum and maximum number of candidates selected for BMS are chosen as $t_0 = 4$ and $t_{max} = 64$. For SCCWalk4l the search depth l, the number of unsuccessful operations before walk perturbation r and the number of candidates for BMS k are set to $l = 4000$, $r = 500$ and $k = 100$ respectively. Exact solvers are being assigned a time limit of 3,600 seconds and heuristic solvers a time limit of 1,000 seconds, since they are expected to compute a result more quickly.

6.2 Graph Instances

The algorithms are evaluated on a broad selection of graphs, covering different sizes, densities, weightings and application areas. Some of the graphs are originally unweighted and thus have to be assigned weights artificially. To cover as many applications as possible, multiple random weighting schemes are considered in the experiments. For each unweighted graph, weights are drawn in the range $[1, 200]$ from uniform (uni), power-law (pow) and exponential (exp) distributions. For more general results, three different random

seeds are used for each distribution and the average results are reported. Four datasets are highlighted in this work:

Network data repository graphs. 10 real-world graphs are taken from the network data repository (REP) [43], ranging from biological applications to social network statistics. Most of these graphs are very sparse (0.01 to 0.00001) but huge, some having over a million vertices.

OpenStreetMap graphs. 12 real-world instances are taken from `openstreetmap.org`. These graphs are generally smaller but slightly more dense and have the advantage of having non-artificial weights.

Random hyperbolic graphs. Random Hyperbolic Graphs (RHG) are randomly generated graphs, such that the vertex degrees follow a power-law distribution, i.e., the number of vertices with degree i is proportional to $i^{-\beta}$, where β is the power-law exponent. RHGs are assumed to model real world graphs very well, since these usually follow a power-law distribution as well [23]. For more information on random graph generation techniques we refer to a recently published survey by Penschuck et al. [38]. 13 RHGs with between 250,000 and 750,000 vertices are generated using the random graph generator KaGen [18]. The power-law exponent is varied between 1.75 and 2.25 and the average degree is chosen between 100 and 500.

DIMACS graphs. In the second DIMACS implementation challenge [27], many graphs specifically intended for comparing different algorithms for clique-related problems were made available. As they are also used in most relevant literature, they are a natural choice for comparing with other state-of-the-art solvers. Chosen from the dataset are 23 graphs that are relatively small but have high densities ranging from 0.5 to 0.99.

For more detailed information on the graph instances, see Tables 7.1-7.14 in the Appendix.

6.3 Experimental Setup

Experiments are performed on a Intel Xeon Silver 4216 CPU @ 2.10GHz with 16 cores under Linux with 95 GB of RAM. All solvers are implemented in C/C++ and compiled using GNU gcc -O3. For computing the results, each solver solves up to 16 graph instances in parallel; by running the solver exclusively on the machine, there is no relevant difference to solving the graph instances sequentially. For all solvers, the solution quality $w(\hat{C})$, and the time to find that solution t_{sol} are reported. For exact solvers, the time to prove its optimality t_{prv} is measured additionally. Solvers that use random number generation (RNG) are run five times with different seeds and report their average solutions, in order to better capture their general performance. Furthermore, for the graphs that were assigned

artificial weights, the results for the different weightings are averaged, such that each solver reports one result tuple per graph for each weighting scheme. The solution weights are rounded to integer numbers for better interpretability in this case.
The results are presented for each weighting scheme separately, in order to get a better idea of each solvers individual strengths and weaknesses. For better interpretability, the best values are marked as bold in each line. If an exact algorithm is not able to prove one of the three graphs in time, t_{prv} is marked as $-$. As exact- and heuristic solvers follow different objectives, the discussion focuses on comparing them among each other. An algorithm is generally considered to outperform another algorithm, if it reaches a higher solution weight, or if it reaches the same solution weight faster. In the case of heuristic solvers however, a solver computing a slightly lower weight solution significantly faster can also be considered superior, depending on whether speed or quality is more important for the use-case.

6.4 Comparing the Exact Algorithms

First, the exact solver MWCRedu is compared with the state-of-the-art solver TSM-MWC on each dataset, starting with the OpenStreetMap graph instances.

6.4.1 OpenStreetMap graphs

The results for all exact solvers on the OSM dataset are shown in Table 6.1. Between TSM-MWC and MWCRedu, the latter clearly dominates, both when it comes to the time to find the solution t_{sol}, and the time to prove its optimality t_{prv}. In two cases TSM-MWC even fails to find the optimal solution within the time limitation of 3,600 seconds.

Graph	t_{sol}		t_{prv}		$w(\hat{C})$	
	TSM-MWC	MWCRedu	TSM-MWC	MWCRedu	TSM-MWC	MWCRedu
district-of-columbia-AM2	0.80	**0.15**	0.88	**0.21**	**235,777**	**235,777**
district-of-columbia-AM3	1,937.83	**5.64**	1,938.06	**6.52**	**545,969**	**545,969**
greenland-AM3	10.31	**0.82**	11.91	**3.30**	**604,575**	**604,575**
hawaii-AM3	3,598.87	**30.88**	-	**54.68**	1,110,978	**1,229,741**
idaho-AM3	218.66	**4.47**	220.42	**5.55**	**1,101,721**	**1,101,721**
kentucky-AM3	3,580.19	**114.09**	-	**144.51**	1,808,419	**1,860,308**
massachusetts-AM3	0.81	**0.03**	1.01	**0.13**	**115,636**	**115,636**
oregon-AM3	8.93	**1.70**	11.07	**2.47**	**557,634**	**557,634**
rhode-island-AM3	81.30	**11.03**	93.69	**18.40**	**1,162,925**	**1,162,925**
vermont-AM3	4.57	**0.49**	4.90	**0.56**	**604,213**	**604,213**
virginia-AM3	0.13	**0.06**	0.21	**0.06**	**207,457**	**207,457**
washington-AM3	12.22	**1.01**	12.75	**1.06**	**356,314**	**356,314**
Geometric mean						
	27.62	**1.55**	31.01	**2.43**	537,149	**542,993**

Table 6.1: OSM exact results

6.4.2 DIMACS Graphs

Table 6.2 shows the results of the exact solvers for the DIMACS instances. There is no big difference in performance between the two solvers here, since none of the exact reductions employed in MWCRedu are able to remove vertices or edges for any DIMACS instance, and the solver thus quickly proceeds to apply the B&B solver on the unreduced graph, which uses the same techniques as TSM-MWC. The overhead from applying the reduction rules is only notable for the easier instances. MWCRedu even performs slightly better on average, which is likely due to better initial solutions obtained from running local search during the reduction phase.

Graph	t_{sol} TSM-MWC	t_{sol} MWCRedu	t_{prv} TSM-MWC	t_{prv} MWCRedu	$w(\hat{C})$ TSM-MWC	$w(\hat{C})$ MWCRedu
Uniform weighted						
brock800_1	**1,122.28**	1,191.32	**2,667.43**	2,820.98	**3,006**	**3,006**
brock800_2	**2,601.19**	2,752.57	**3,111.08**	3,285.93	**3,074**	**3,074**
brock800_3	**1,318.78**	1,400.10	**2,735.77**	2,896.13	**2,984**	**2,984**
brock800_4	**2,019.67**	2,137.40	-	-	**3,059**	**3,059**
C1000.9	2,889.65	**1,429.16**	-	-	7,338	**7,459**
C2000.5	**1,867.49**	1,908.40	-	-	**2,395**	**2,395**
C2000.9	2,248.93	**697.11**	-	-	7,898	**8,284**
C4000.5	2,804.98	**2,488.93**	-	-	**2,460**	2,437
C500.9	2,716.53	**2,611.48**	-	-	**6,789**	**6,789**
gen400_p0.9_55	**2,359.72**	2,378.30	-	-	**6,654**	**6,654**
gen400_p0.9_65	**2,768.34**	2,772.78	-	-	**6,535**	**6,535**
gen400_p0.9_75	1,420.53	**1,388.95**	2,522.75	**2,511.52**	**7,492**	**7,492**
hamming10-4	3,110.88	**3,018.87**	-	-	**5,205**	5,125
johnson32-2-4	**1,983.31**	2,070.46	-	-	**2,935**	**2,935**
keller5	**2,801.07**	2,821.09	-	-	3,807	**3,827**
keller6	2,569.15	**2,301.13**	-	-	5,617	**6,175**
MANN_a27	2.67	**2.09**	2.81	**2.14**	**17,866**	**17,866**
MANN_a45	121.72	**59.03**	126.40	**63.31**	**49,459**	**49,459**
MANN_a81	3,599.91	**1,922.82**	-	**3,116.00**	118,898	**161,903**
p_hat1000-3	2,177.09	**2,068.98**	-	-	**8,261**	**8,261**
p_hat1500-2	952.79	**902.21**	2,491.91	**2,371.21**	**7,556**	**7,556**
p_hat1500-3	1,212.92	**1,111.24**	-	-	**10,796**	**10,796**
sanr400_0.7	**11.86**	12.69	**26.63**	28.43	**2,926**	**2,926**
Powerlaw weighted						
brock800_1	**2.13**	2.38	**2.78**	3.03	**1,249**	**1,249**
brock800_2	**2.23**	2.48	**2.58**	2.83	**1,329**	**1,329**
brock800_3	**1.71**	1.95	**2.12**	2.35	**1,427**	**1,427**
brock800_4	**1.00**	1.30	**2.34**	2.59	**1,347**	**1,347**
C1000.9	**1,276.17**	1,957.97	-	-	3,181	**3,226**
C2000.5	**14.67**	16.13	**30.77**	31.50	**1,284**	**1,284**
C2000.9	2,221.50	**1,926.19**	-	-	3,407	**3,409**

C4000.5	2,426.53	**2,309.13**	-	**3,235.54**	**1,521**	**1,521**
C500.9	**2.11**	2.15	**2.98**	3.04	**2,432**	**2,432**
gen400_p0.9_55	**0.48**	0.50	**0.64**	0.66	**2,110**	**2,110**
gen400_p0.9_65	**0.48**	0.52	**0.69**	0.73	**2,043**	**2,043**
gen400_p0.9_75	**0.55**	0.58	**0.73**	0.77	**2,075**	**2,075**
hamming10-4	**7.11**	7.78	**10.85**	11.38	**2,232**	**2,232**
johnson32-2-4	**0.72**	0.95	**0.94**	1.16	**1,666**	**1,666**
keller5	**1.79**	2.19	**2.46**	2.68	**1,591**	**1,591**
keller6	**2,080.84**	2,670.39	-	-	3,370	**3,407**
MANN_a27	2.84	**0.22**	2.89	**0.59**	**4,049**	**4,049**
MANN_a45	144.45	**6.06**	145.28	**11.54**	**10,617**	**10,617**
MANN_a81	3,599.86	**751.85**	-	**915.33**	21,120	**35,092**
p_hat1000-3	**3.81**	4.33	**6.43**	6.81	**2,199**	**2,199**
p_hat1500-2	**0.85**	2.41	**6.69**	7.91	**2,031**	**2,031**
p_hat1500-3	**17.18**	18.43	**39.87**	40.15	**2,793**	**2,793**
sanr400_0.7	**0.18**	0.23	**0.26**	0.31	**1,135**	**1,135**
Exponential weighted						
brock800_1	29.47	**29.27**	100.94	**100.37**	**509**	**509**
brock800_2	54.97	**54.65**	97.37	**97.06**	**516**	**516**
brock800_3	139.08	**138.13**	143.18	**142.29**	**502**	**502**
brock800_4	37.68	**37.63**	71.59	**70.75**	**533**	**533**
C1000.9	**1,768.54**	2,121.71	-	-	1,087	**1,100**
C2000.5	462.88	**429.93**	1,096.22	**1,033.57**	**481**	**481**
C2000.9	714.16	**652.71**	-	-	1,191	**1,295**
C4000.5	2,728.03	**2,628.18**	-	-	**507**	**507**
C500.9	1,307.10	**1,139.58**	-	-	**1,005**	**1,005**
gen400_p0.9_55	192.14	**187.58**	284.62	**278.19**	**949**	**949**
gen400_p0.9_65	950.94	**930.35**	1,091.49	**1,067.74**	**945**	**945**
gen400_p0.9_75	509.76	**501.27**	634.89	**624.50**	**987**	**987**
hamming10-4	1,451.38	**734.07**	-	-	**900**	890
johnson32-2-4	**2.93**	3.50	**3.32**	4.00	**643**	**643**
keller5	**50.26**	51.46	**65.47**	66.55	**618**	**618**
keller6	**1,628.11**	1,915.29	-	-	1,007	**1,009**
MANN_a27	1.33	**0.34**	1.40	**0.74**	**2,257**	**2,257**
MANN_a45	65.69	**11.41**	66.52	**13.92**	**6,285**	**6,285**
MANN_a81	3,599.61	**739.94**	-	**767.71**	17,249	**20,926**
p_hat1000-3	1,534.97	**1,423.81**	-	**2,083.65**	**1,025**	**1,025**
p_hat1500-2	61.17	**57.27**	96.95	**90.46**	**1,020**	**1,020**
p_hat1500-3	1,366.75	**1,273.09**	-	-	**1,419**	**1,419**
sanr400_0.7	**2.05**	2.17	**2.28**	2.40	**449**	**449**
Geometric mean						
	128.36	**106.57**	204.09	**179.36**	2,485	**2,531**

Table 6.2: DIMACS exact results

6.4.3 Network Data Repository Graphs

The results of the REP instances are shown in Table 6.3. MWCRedu and TSM-MWC both outperform the respective other for specific instances. TSM-MWC proves very efficient for large instances with more than 1,000,000 vertices, whereas MWCRedu clearly outperforms TSM-MWC for the smaller, more dense biology graphs. While TSM-MWC is faster on average, it fails to prove a solutions optimality five times, three of which actually being suboptimal.

Graph	t_{sol} TSM-MWC	t_{sol} MWCRedu	t_{prv} TSM-MWC	t_{prv} MWCRedu	$w(\hat{C})$ TSM-MWC	$w(\hat{C})$ MWCRedu
Uniform weighted						
aff-digg	**16.55**	45.40	**244.04**	273.67	**3,829**	**3,829**
bio-human-gene1	3,598.98	**2,010.82**	-	**3,327.65**	136,325	**136,692**
bio-human-gene2	2,298.36	**474.49**	-	**1,380.66**	**131,904**	**131,904**
bio-mouse-gene	1,789.70	**198.47**	-	**240.97**	50,785	**59,476**
sc-TSOPF-RS-b2383	**9.70**	27.59	**9.70**	365.22	**913**	**913**
soc-flickr-und	**35.10**	133.73	**74.52**	162.42	**10,847**	**10,847**
soc-orkut	**56.42**	135.79	**67.48**	144.89	**5,832**	**5,832**
soc-orkut-dir	**46.26**	116.30	**60.22**	128.33	**5,261**	**5,261**
web-wikipedia_link_it	170.90	**45.99**	171.08	**46.15**	**87,175**	**87,175**
web-wikipedia-growth	**13.09**	84.77	**17.30**	88.57	**3,334**	**3,334**
Powerlaw weighted						
aff-digg	**10.03**	165.91	**61.06**	212.16	**1,369**	**1,369**
bio-human-gene1	3,599.34	**445.04**	-	**611.50**	21,278	**21,689**
bio-human-gene2	2,174.21	**330.54**	2,289.48	**430.11**	**19,341**	**19,341**
bio-mouse-gene	168.31	**123.91**	277.04	**154.11**	**9,252**	**9,252**
sc-TSOPF-RS-b2383	**3.14**	301.66	**5.59**	618.56	**565**	**565**
soc-flickr-und	**5.39**	124.55	**28.66**	137.52	**2,530**	**2,530**
soc-orkut	**43.33**	176.24	**43.69**	176.59	**1,420**	**1,420**
soc-orkut-dir	**36.38**	150.88	**36.93**	151.23	**1,322**	**1,322**
web-wikipedia_link_it	165.36	**47.57**	165.43	**47.69**	**13,005**	**13,005**
web-wikipedia-growth	**7.58**	46.50	**9.36**	46.68	**921**	**921**
Exponential weighted						
aff-digg	**8.94**	30.44	**99.98**	117.26	**581**	**581**
bio-human-gene1	2,797.16	**1,135.33**	-	**1,847.39**	**15,258**	**15,258**
bio-human-gene2	604.93	**241.30**	930.09	**627.22**	**14,903**	**14,903**
bio-mouse-gene	267.89	**161.45**	383.23	**190.15**	**6,429**	**6,429**
sc-TSOPF-RS-b2383	**2.62**	189.53	**5.89**	512.73	**222**	**222**
soc-flickr-und	**16.11**	126.65	**50.38**	152.17	**1,198**	**1,198**
soc-orkut	**52.04**	152.53	**59.15**	157.67	**735**	**735**
soc-orkut-dir	**39.85**	136.78	**51.93**	145.51	**675**	**675**
web-wikipedia_link_it	**24.95**	48.80	**25.01**	48.89	**9,616**	**9,616**
web-wikipedia-growth	**10.76**	75.80	**13.96**	79.01	**414**	**414**
Geometric mean						
	69.78	141.39	**117.80**	215.87	4,420	**4,446**

Table 6.3: REP exact results

6.4.4 Random Hyperbolic Graphs

The results of each solver on the RHG instances are shown in Table 6.4. Here MWCRedu outperforms its competitor TSM-MWC clearly. The reason for its good performance is likely the structure of RHGs, which allows it to remove most vertices quickly using very efficient reductions. For most of the instances, the graph is reduced to 0, such that there is no need to apply B&B on the reduced graph. MWCRedu reaches the optimal solution faster than TSM-MWC 35 out of 39 times, often by magnitudes, with TSM-MWC failing to find the optimal solution twice.

Graph	t_{sol} TSM-MWC	t_{sol} MWCRedu	t_{prv} TSM-MWC	t_{prv} MWCRedu	$w(\hat{C})$ TSM-MWC	$w(\hat{C})$ MWCRedu
Uniform weighted						
rhg_250000_100_1.75	94.71	**2.26**	94.91	**2.72**	**99,839**	**99,839**
rhg_250000_100_2.25	2.53	**2.02**	2.75	**2.10**	**37,947**	**37,947**
rhg_250000_250_1.75	101.48	**4.74**	107.14	**4.76**	**112,769**	**112,769**
rhg_250000_250_2.25	21.87	**4.53**	22.60	**4.66**	**71,001**	**71,001**
rhg_250000_500_1.75	1,079.84	**35.47**	1,092.32	**38.43**	**137,234**	**137,234**
rhg_250000_500_2.25	51.02	**9.05**	52.12	**9.25**	**102,364**	**102,364**
rhg_500000_250_1.75	3,093.33	**37.09**	-	**37.70**	130,973	**131,559**
rhg_500000_250_2.25	88.77	**11.36**	89.79	**11.71**	**88,512**	**88,512**
rhg_500000_500_2.25	43.49	**15.11**	47.06	**21.09**	**122,781**	**122,781**
rhg_750000_250_1.75	3,599.25	**33.80**	-	**55.44**	150,676	**160,845**
rhg_750000_250_2.25	132.25	**18.15**	132.43	**18.33**	**96,362**	**96,362**
rhg_750000_500_1.75	**9.74**	311.73	**226.52**	1,173.96	**207,197**	**207,197**
rhg_750000_500_2.25	35.91	**34.13**	49.39	**34.19**	**119,936**	**119,936**
Powerlaw weighted						
rhg_250000_100_1.75	111.00	**3.29**	111.11	**3.62**	**14,183**	**14,183**
rhg_250000_100_2.25	13.12	**2.06**	13.18	**2.07**	**5,801**	**5,801**
rhg_250000_250_1.75	152.12	**4.82**	153.23	**5.14**	**16,771**	**16,771**
rhg_250000_250_2.25	37.22	**4.31**	37.38	**4.32**	**10,541**	**10,541**
rhg_250000_500_1.75	873.87	**32.70**	881.02	**32.74**	**20,188**	**20,188**
rhg_250000_500_2.25	189.67	**12.36**	190.08	**12.75**	**14,720**	**14,720**
rhg_500000_250_1.75	1,043.61	**34.01**	1,051.70	**34.51**	**19,356**	**19,356**
rhg_500000_250_2.25	137.13	**12.11**	137.44	**12.27**	**12,997**	**12,997**
rhg_500000_500_2.25	360.74	**21.86**	360.99	**23.01**	**18,100**	**18,100**
rhg_750000_250_1.75	1,014.10	**40.73**	1,027.87	**53.30**	**22,720**	**22,720**
rhg_750000_250_2.25	125.19	**16.95**	125.22	**17.51**	**14,014**	**14,014**
rhg_750000_500_1.75	**406.34**	547.85	**595.45**	798.33	**29,071**	**29,071**
rhg_750000_500_2.25	249.15	**35.45**	257.06	**38.04**	**17,746**	**17,746**
Exponential weighted						
rhg_250000_100_1.75	14.45	**1.92**	14.58	**2.04**	**10,808**	**10,808**
rhg_250000_100_2.25	4.15	**2.04**	4.26	**2.09**	**4,270**	**4,270**
rhg_250000_250_1.75	32.84	**4.43**	37.15	**4.44**	**12,368**	**12,368**
rhg_250000_250_2.25	12.28	**4.93**	12.59	**4.98**	**7,930**	**7,930**
rhg_250000_500_1.75	332.80	**36.26**	345.84	**38.85**	**15,189**	**15,189**
rhg_250000_500_2.25	51.83	**11.02**	52.34	**11.12**	**11,225**	**11,225**
rhg_500000_250_1.75	261.02	**33.81**	278.67	**39.47**	**14,518**	**14,518**
rhg_500000_250_2.25	27.72	**11.38**	28.07	**11.40**	**9,675**	**9,675**

rhg_500000_500_2.25	45.58	**23.03**	45.93	**23.31**	**13,894**	**13,894**
rhg_750000_250_1.75	639.05	**54.61**	654.30	**62.65**	**17,391**	**17,391**
rhg_750000_250_2.25	24.35	**17.75**	24.60	**17.84**	**10,849**	**10,849**
rhg_750000_500_1.75	**62.11**	379.38	**244.05**	554.79	**22,404**	**22,404**
rhg_750000_500_2.25	**7.43**	33.59	**18.56**	33.64	**13,387**	**13,387**
Geometric mean						
	92.71	**15.60**	110.56	**17.49**	26,932	**26,980**

Table 6.4: RHG exact results

Figure 6.1 shows the solution quality achieved by MWCRedu and TSM-MWC for each dataset, where the value 0 indicates that the solver found the better solution, and values above 0 give the percentage the found solution was below the best solution. For the instances where no bar is visible, both algorithms compute the same solution weight and are thus assigned the value 0. Otherwise, MWCRedu mostly finds a higher weight solution, with TSM-MWC sometimes reaching a significantly lower solution weight.
Figure 6.2 shows for both exact solvers the time to get to the solution relative to the best time among the solvers on a logarithmic scale. A value of 100 for example indicates, that the solver took 100 times longer to get to its solution. The graphic illustrates the strength of each solver well, with TSM-MWC being faster for most REP instances, and MWCRedu taking the lead for most OSM and RHG instances.

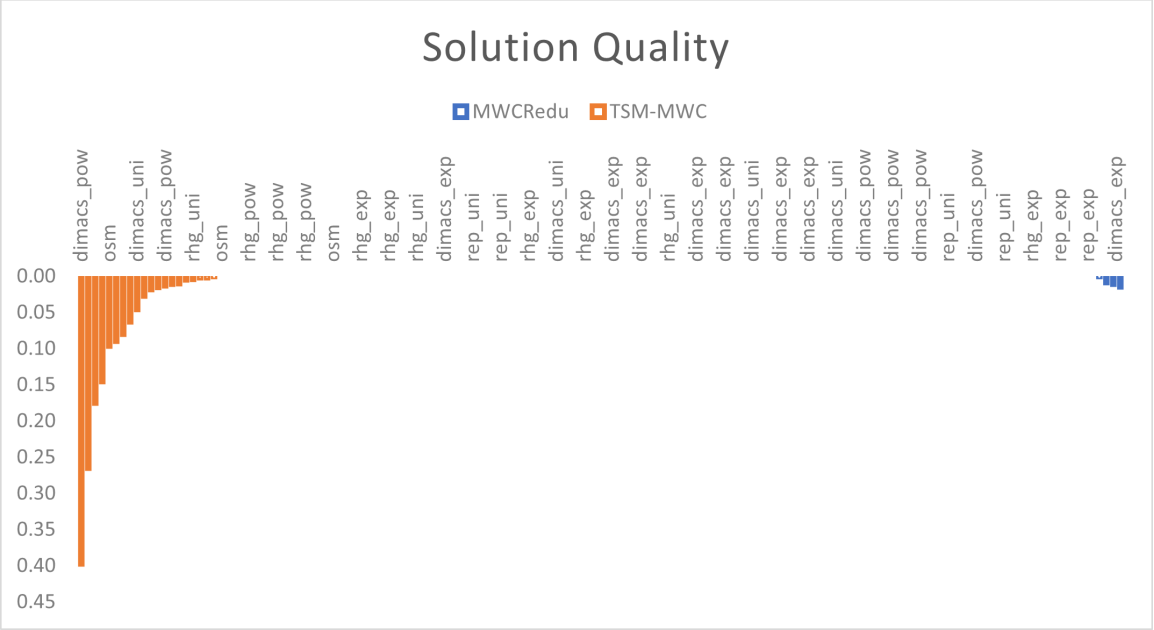

Figure 6.1: Solution quality for exact solvers.

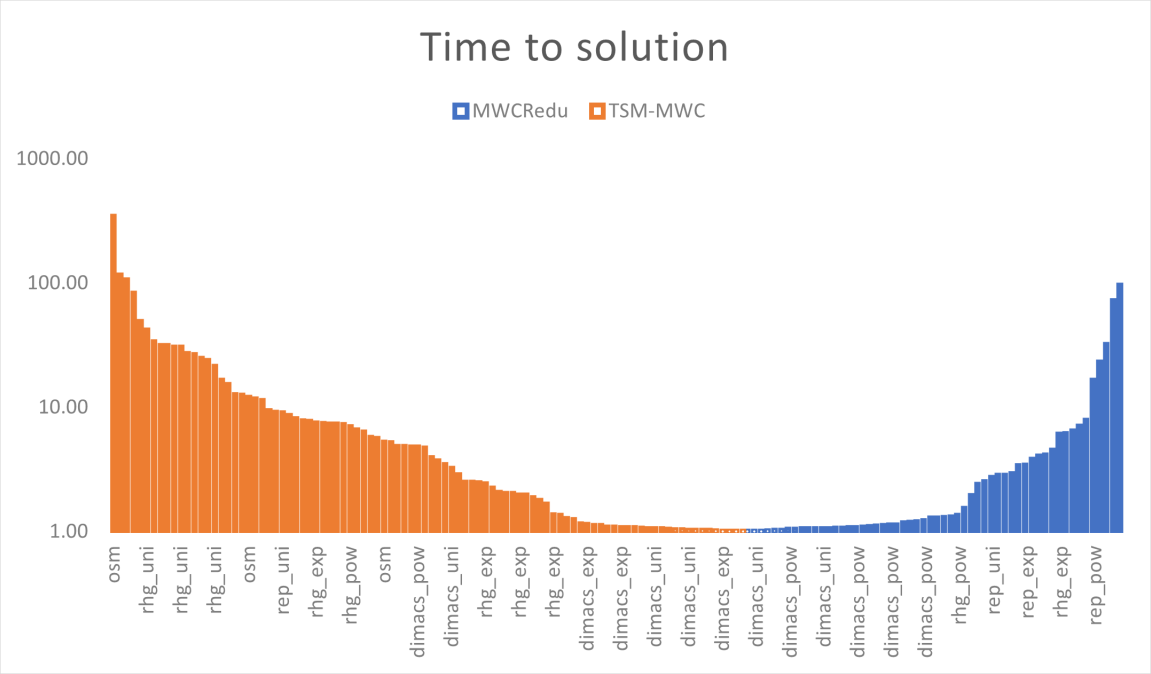

Figure 6.2: Time to solution for exact solvers.

6.5 Comparing the Peeling Rules

Before the heuristic solvers are compared amongst each other, first the peeling rule that should be employed in MWCPeel has to be decided. One indicator of the quality of each peeling rule are the performance scores after training the models, where the score is the difference between the average score assigned to solution vertices and the average score assigned to other vertices (Section 4.2.3). The performance measures for each approach are listed in Table 6.5. As indicated by the low values of 1 % and 2 %, both $\text{Peel}_{\text{MLPfast}}$ and $\text{Peel}_{\text{MLPfull}}$ are not capable of learning a pattern that lets them reliably distinguish solution vertices from ordinary vertices. Both strategies are therefore not further investigated in this work, though it cannot be ruled out that a different MLP architecture or additional

Approach	Performance
$\text{Peel}_{\text{MLPfast}}$	1 %
$\text{Peel}_{\text{MLPfull}}$	2 %
$\text{Peel}_{\text{DeepSet}}$	45 %
Peel_{GNN}	54 %

Table 6.5: The highest achieved difference between the average prediction for solution vertices vs. other vertices on the validation set during training by each peeling approach.

features may improve the performance. The DeepSet and GNN approach both show a good performance on the validation data, assigning a prediction score to solution vertices that is on average around 50 % higher than the prediction for other vertices. These models will therefore be compared along with the heuristic rule Peel_{UB}, which computes the score for each $v \in V$ as $w(N[v])$ (Section 4.2), in order to determine the best peeling strategy for employment in the heuristic solver. For the comparison, a preliminary experiment is conducted, where each solver uses the same set of exact reduction rules and reduction strategy. Each algorithm is run with five different random seeds and a time limit of 1,000 s.

6.5.1 OpenStreetMap graphs

For the OSM instances all peeling rules achieve similar results (Table 6.6), since the graphs can be reduced by the exact reduction rules for the most part. While all peeling rules lead to the optimal solutions, Peel_{UB} is slightly faster than its competitors.

6.5.2 DIMACS Graphs

For the DIMACS instances, shown in Table 6.7, both the time and quality of the solutions differ notably between the peeling rules. Out of the 69 instances, Peel_{UB} achieves the best solution among the three 43 times and the best speed 42 times. It furthermore achieves a lower time and higher solution weight on average, proving to be the best general DIMACS solver out of the three.

Graph	t_{sol}			$w(\hat{C})$		
	Peel_{UB}	$\text{Peel}_{\text{DeepSet}}$	Peel_{GNN}	Peel_{UB}	$\text{Peel}_{\text{DeepSet}}$	Peel_{GNN}
district-of-columbia-AM2	**0.16**	0.21	0.18	**235,777**	**235,777**	**235,777**
district-of-columbia-AM3	**5.26**	5.59	5.46	**545,969**	**545,969**	**545,969**
greenland-AM3	**0.85**	0.99	**0.85**	**604,575**	**604,575**	**604,575**
hawaii-AM3	**29.82**	31.38	31.90	**1,229,741**	**1,229,741**	**1,229,741**
idaho-AM3	**4.29**	4.72	4.36	**1,101,721**	**1,101,721**	**1,101,721**
kentucky-AM3	**102.17**	109.32	112.18	**1,860,308**	**1,860,308**	**1,860,308**
massachusetts-AM3	**0.03**	0.04	0.04	**115,636**	**115,636**	**115,636**
oregon-AM3	**1.72**	1.90	1.73	**557,634**	**557,634**	**557,634**
rhode-island-AM3	**9.34**	10.04	9.66	**1,162,925**	**1,162,925**	**1,162,925**
vermont-AM3	**0.39**	0.50	0.42	**604,213**	**604,213**	**604,213**
virginia-AM3	**0.05**	0.07	0.06	**207,457**	**207,457**	**207,457**
washington-AM3	**0.96**	1.14	1.03	**356,314**	**356,314**	**356,314**
Geometric mean						
	1.45	1.67	1.55	**542,993**	**542,993**	**542,993**

Table 6.6: OSM peeling results

Graph	t_{sol} Peel$_{\text{UB}}$	Peel$_{\text{DeepSet}}$	Peel$_{\text{GNN}}$	$w(\hat{C})$ Peel$_{\text{UB}}$	Peel$_{\text{DeepSet}}$	Peel$_{\text{GNN}}$
Uniform weighted						
brock800_1	**45.67**	296.95	123.71	2,886	**2,911**	2,689
brock800_2	**59.84**	280.14	60.74	2,935	**2,949**	2,696
brock800_3	**37.69**	295.42	56.04	**2,912**	2,870	2,756
brock800_4	**48.01**	253.46	65.69	**2,887**	2,864	2,661
C1000.9	262.85	476.10	**196.86**	7,779	**8,116**	7,473
C2000.5	536.78	**388.36**	559.53	**2,390**	2,379	2,348
C2000.9	123.18	223.43	**57.84**	8,603	**9,249**	8,234
C4000.5	638.30	**572.47**	722.99	2,472	**2,504**	2,392
C500.9	474.51	152.05	**81.93**	**6,964**	6,832	6,853
gen400_p0.9_55	50.16	114.81	**24.39**	**6,614**	6,301	6,230
gen400_p0.9_65	59.97	100.00	**59.04**	**6,654**	6,420	6,305
gen400_p0.9_75	10.78	91.87	**8.96**	**7,261**	6,763	6,863
hamming10-4	527.12	**281.45**	512.84	5,279	**5,562**	5,093
johnson32-2-4	537.90	334.79	**103.21**	3,020	**3,042**	3,022
keller5	326.39	**268.56**	787.34	3,545	**3,700**	3,657
keller6	400.16	528.21	**353.12**	6,103	**6,680**	6,054
MANN_a27	**1.63**	12.19	6.46	**17,710**	17,694	**17,710**
MANN_a45	93.78	55.44	**54.00**	49,312	49,221	**49,345**
MANN_a81	**7.17**	7.78	7.62	**161,648**	**161,648**	**161,648**
p_hat1000-3	404.89	**289.30**	358.53	**8,223**	8,052	8,184
p_hat1500-2	475.25	506.69	**287.15**	**7,546**	7,540	7,411
p_hat1500-3	**206.37**	340.88	257.59	**10,801**	10,640	10,740
sanr400_0.7	**0.91**	158.29	34.69	2,874	**2,877**	2,741
Powerlaw weighted						
brock800_1	**0.87**	472.33	95.75	**1,242**	1,205	**1,242**
brock800_2	**0.84**	392.52	94.97	1,274	**1,286**	1,280
brock800_3	**0.78**	479.05	92.87	**1,415**	1,398	**1,415**
brock800_4	**0.81**	421.78	106.22	**1,335**	1,319	**1,335**
C1000.9	**4.41**	341.67	141.83	**3,276**	3,077	3,259
C2000.5	**3.58**	597.93	177.25	**1,275**	1,190	1,207
C2000.9	592.43	504.53	**206.57**	**4,129**	3,806	3,973
C4000.5	**27.26**	465.50	239.89	**1,494**	1,358	1,377
C500.9	**0.57**	235.04	93.65	2,387	2,385	**2,419**
gen400_p0.9_55	**0.36**	210.98	68.19	2,093	2,061	**2,097**
gen400_p0.9_65	**0.38**	191.70	58.81	2,005	1,966	**2,027**
gen400_p0.9_75	**0.37**	203.90	57.89	2,025	2,016	**2,047**
hamming10-4	**2.32**	219.49	85.25	**2,214**	2,047	**2,214**
johnson32-2-4	**0.81**	389.37	75.59	**1,666**	1,638	**1,666**
keller5	**0.95**	388.03	47.00	**1,539**	1,536	**1,539**
keller6	521.83	291.14	**142.58**	**3,249**	2,932	3,078
MANN_a27	**2.76**	3.98	3.52	4,048	4,045	**4,049**

MANN_a45	35.96	**16.36**	18.19	**10,616**	10,592	**10,616**
MANN_a81	62.98	7.73	**7.58**	**34,937**	34,867	34,867
p_hat1000-3	**2.40**	305.36	59.17	**2,198**	2,051	2,188
p_hat1500-2	**2.33**	374.30	126.81	**2,031**	1,873	**2,031**
p_hat1500-3	**10.13**	367.65	132.96	**2,788**	2,517	2,770
sanr400_0.7	**0.23**	326.94	73.13	**1,131**	1,102	**1,131**
Exponential weighted						
brock800_1	**2.33**	292.66	137.05	**504**	491	496
brock800_2	**2.57**	240.58	140.50	**502**	485	497
brock800_3	**3.47**	316.01	137.44	**495**	478	488
brock800_4	**2.73**	343.82	140.67	**525**	512	522
C1000.9	499.37	232.02	**156.05**	1,225	1,209	**1,259**
C2000.5	**20.92**	292.28	175.98	**480**	452	462
C2000.9	469.85	320.14	**181.93**	1,416	1,453	**1,533**
C4000.5	483.52	**221.37**	363.51	**518**	501	476
C500.9	**23.52**	172.45	106.23	**1,043**	988	1,037
gen400_p0.9_55	**0.50**	137.73	82.33	914	887	**917**
gen400_p0.9_65	**0.65**	139.43	82.78	**935**	887	918
gen400_p0.9_75	**0.51**	93.08	60.30	**975**	930	960
hamming10-4	445.21	246.87	**128.07**	886	851	**900**
johnson32-2-4	**0.95**	333.90	108.72	**643**	615	**643**
keller5	**5.05**	299.65	98.06	**600**	574	588
keller6	599.11	**394.02**	683.51	1,038	**1,158**	1,014
MANN_a27	**1.38**	9.39	2.48	2,249	**2,250**	2,249
MANN_a45	122.03	**0.23**	49.71	6,275	6,252	**6,280**
MANN_a81	**7.07**	7.76	7.60	**20,810**	**20,810**	**20,810**
p_hat1000-3	240.07	211.71	**79.32**	**1,023**	979	1,006
p_hat1500-2	**58.95**	299.75	87.60	**1,020**	1,003	993
p_hat1500-3	488.84	172.07	**124.61**	1,414	1,332	**1,421**
sanr400_0.7	**0.23**	202.82	79.50	438	437	**439**
Geometric mean						
	16.28	171.11	86.07	**2,527**	2,479	2,488

Table 6.7: DIMACS peeling results

6.5.3 Network Data Repository Graphs

The experimental results for the peeling rules on REP graphs are presented in Table 6.8. Again, Peel_{UB} achieves the highest weight solution while taking the least time on average, closely followed by Peel_{GNN}.

Graph	t_{sol}			$w(\hat{C})$		
	Peel_{UB}	$\text{Peel}_{\text{DeepSet}}$	Peel_{GNN}	Peel_{UB}	$\text{Peel}_{\text{DeepSet}}$	Peel_{GNN}
Uniform weighted						
aff-digg	**47.68**	118.74	63.92	**3,829**	**3,829**	**3,829**
bio-human-gene1	**493.36**	521.23	719.66	**136,713**	136,708	136,661
bio-human-gene2	**89.59**	156.39	283.12	**131,904**	131,901	**131,904**
bio-mouse-gene	13.43	5.26	**4.82**	**59,146**	58,839	58,839
sc-TSOPF-RS-b2383	**1.43**	1.67	1.50	**870**	**870**	**870**
soc-flickr-und	**44.40**	57.08	65.27	**10,847**	9,829	**10,847**
soc-orkut	183.51	**89.51**	219.76	**5,582**	3,870	4,715
soc-orkut-dir	185.02	**73.54**	259.54	**5,116**	3,338	4,501
web-wikipedia_link_it	**36.59**	43.17	41.91	87,175	87,175	**104,319**
web-wikipedia-growth	66.06	164.50	**58.48**	**3,136**	2,999	3,065
Powerlaw weighted						
aff-digg	**31.39**	553.41	264.93	**1,369**	1,183	1,334
bio-human-gene1	**142.75**	241.61	393.90	**21,689**	21,688	**21,689**
bio-human-gene2	**97.32**	167.18	287.44	19,300	19,316	**19,341**
bio-mouse-gene	**27.07**	43.39	135.55	9,210	9,102	**9,233**
sc-TSOPF-RS-b2383	**5.38**	54.24	7.18	525	**565**	445
soc-flickr-und	**43.09**	298.11	203.82	2,460	2,243	**2,503**
soc-orkut	**169.20**	348.77	184.20	**1,417**	1,233	1,395
soc-orkut-dir	**147.82**	256.43	149.83	**1,322**	1,244	**1,322**
web-wikipedia_link_it	45.00	54.22	**44.54**	**13,005**	**13,005**	**13,005**
web-wikipedia-growth	**48.24**	108.89	65.59	900	907	**913**
Exponential weighted						
aff-digg	**35.14**	203.74	249.47	**581**	**581**	565
bio-human-gene1	**248.47**	306.64	661.16	**15,258**	15,248	15,245
bio-human-gene2	**82.10**	149.66	173.95	**14,903**	14,898	**14,903**
bio-mouse-gene	**10.40**	18.79	51.77	**6,380**	6,357	6,376
sc-TSOPF-RS-b2383	**4.18**	41.79	11.97	195	**222**	168
soc-flickr-und	**48.66**	338.18	184.58	**1,198**	1,175	1,151
soc-orkut	173.35	**75.52**	217.32	**672**	434	574
soc-orkut-dir	165.67	**95.36**	254.82	**633**	404	549
web-wikipedia_link_it	**49.29**	78.34	56.21	**10,236**	9,614	9,615
web-wikipedia-growth	**57.94**	214.70	63.43	**384**	350	381
Geometric mean						
	51.83	98.12	95.70	**4,355**	4,041	4,226

Table 6.8: REP graphs peeling results

6.5.4 Random Hyperbolic Graphs

Lastly the results on the RHGs are presented in Table 6.9. These graphs are mostly reduced by exact reduction rules, leading to similar results for the different peeling rules. Peel_{GNN} computes slightly higher weight solutions, closely followed by Peel_{UB}.

Graph	t_{sol}			$w(\hat{C})$		
	Peel_{UB}	$\text{Peel}_{\text{DeepSet}}$	Peel_{GNN}	Peel_{UB}	$\text{Peel}_{\text{DeepSet}}$	Peel_{GNN}
Uniform weighted						
rhg_250000_100_1.75	**2.67**	2.72	2.76	**99,839**	**99,839**	**99,839**
rhg_250000_100_2.25	**2.09**	3.63	5.74	**37,947**	37,944	**37,947**
rhg_250000_250_1.75	**4.24**	12.64	15.81	112,756	112,711	**112,757**
rhg_250000_250_2.25	4.58	**4.40**	5.27	**71,001**	70,971	**71,001**
rhg_250000_500_1.75	**18.39**	37.00	45.21	136,884	136,765	**137,167**
rhg_250000_500_2.25	10.15	**8.56**	11.16	**102,364**	102,352	**102,364**
rhg_500000_250_1.75	**22.69**	24.62	35.67	131,100	130,973	**131,518**
rhg_500000_250_2.25	**11.40**	11.60	13.15	**88,512**	88,502	**88,512**
rhg_500000_500_2.25	15.33	**10.15**	20.86	**122,781**	**122,781**	**122,781**
rhg_750000_250_1.75	22.72	**22.58**	23.01	**160,845**	**160,845**	**160,845**
rhg_750000_250_2.25	17.00	**16.73**	16.77	**96,362**	96,349	**96,362**
rhg_750000_500_1.75	21.41	**4.07**	22.39	**207,197**	207,195	**207,197**
rhg_750000_500_2.25	36.00	**19.49**	51.35	**119,936**	119,891	**119,936**
Powerlaw weighted						
rhg_250000_100_1.75	3.17	3.11	**3.06**	**14,183**	**14,183**	**14,183**
rhg_250000_100_2.25	2.04	**1.93**	2.42	**5,801**	5,799	**5,801**
rhg_250000_250_1.75	**5.24**	5.75	6.58	**16,771**	16,770	**16,771**
rhg_250000_250_2.25	**4.63**	4.96	4.93	**10,541**	10,539	**10,541**
rhg_250000_500_1.75	**17.32**	25.10	35.80	20,140	20,123	**20,148**
rhg_250000_500_2.25	12.46	11.86	**11.84**	**14,720**	**14,720**	**14,720**
rhg_500000_250_1.75	**19.29**	27.25	34.34	19,212	19,265	**19,356**
rhg_500000_250_2.25	11.87	**11.53**	12.03	**12,997**	12,995	**12,997**
rhg_500000_500_2.25	22.28	22.79	**22.23**	**18,100**	**18,100**	**18,100**
rhg_750000_250_1.75	**27.64**	27.99	28.20	**22,720**	**22,720**	**22,720**
rhg_750000_250_2.25	**17.01**	17.93	17.92	**14,014**	**14,014**	**14,014**
rhg_750000_500_1.75	48.08	**42.90**	77.46	**29,071**	28,958	**29,071**
rhg_750000_500_2.25	34.75	**29.76**	40.63	**17,746**	17,727	**17,746**
rhg_250000_100_1.75	**2.67**	2.72	2.76	**99,839**	**99,839**	**99,839**
Exponential weighted						
rhg_250000_100_1.75	2.33	**2.30**	2.38	**10,808**	**10,808**	**10,808**
rhg_250000_100_2.25	**1.86**	2.08	3.32	**4,270**	4,267	**4,270**
rhg_250000_250_1.75	4.92	**4.81**	14.28	**12,368**	12,356	**12,368**
rhg_250000_250_2.25	**4.74**	5.32	5.60	**7,930**	7,927	**7,930**
rhg_250000_500_1.75	**19.47**	27.97	31.95	15,157	15,153	**15,186**
rhg_250000_500_2.25	**11.68**	12.31	12.30	**11,225**	11,224	**11,225**
rhg_500000_250_1.75	**22.81**	30.34	34.94	14,488	14,485	**14,518**

rhg_500000_250_2.25	**11.62**	11.95	13.13	**9,675**	**9,675**	**9,675**
rhg_500000_500_2.25	22.31	23.11	**21.87**	**13,894**	**13,894**	**13,894**
rhg_750000_250_1.75	28.84	**28.54**	28.77	**17,391**	**17,391**	**17,391**
rhg_750000_250_2.25	17.66	**17.59**	18.04	**10,849**	**10,849**	**10,849**
rhg_750000_500_1.75	50.97	**23.51**	87.37	**22,404**	22,368	**22,404**
rhg_750000_500_2.25	30.40	**16.81**	52.04	13,386	13,384	**13,387**
Geometric mean						
	11.62	**11.52**	15.48	26,966	26,958	**26,978**

Table 6.9: RHG peeling results

Overall, it is clear, that each peeling rule has its strengths and weaknesses. Especially Peel_{UB} and Peel_{GNN} show comparable performance on average. Peel_{UB} proves to be the more consistent of the two, achieving better results for the DIMACS and REP instances. Peel_{UB} is therefore chosen as the peeling rule employed in MWCPeel.

6.6 Comparing the Heuristic Algorithms

In this section MWCPeel, using the peeling rule Peel_{UB}, is compared with the state-of-the-art solvers FastWCLq and SCCWalk4l.

6.6.1 OpenStreetMap Graphs

As shown in Table 6.10, MWCPeel performs best for 11 out of 12 OSM instances. It is noteworthy, that both MWCPeel and FastWCLq find the optimal solution to all OSM instances despite MWCPeel using inexact reductions to remove some of the graphs vertices.

	t_{sol}			$w(\hat{C})$		
Graph	FastWCLq	SCCWalk4l	MWCPeel	FastWCLq	SCCWalk4l	MWCPeel
district-of-columbia-AM2	0.32	4.91	**0.16**	**235,777**	234,219	**235,777**
district-of-columbia-AM3	16.87	208.46	**5.26**	**545,969**	**545,969**	**545,969**
greenland-AM3	2.96	39.33	**0.85**	**604,575**	**604,575**	**604,575**
hawaii-AM3	86.47	727.42	**29.82**	**1,229,741**	1,224,690	**1,229,741**
idaho-AM3	15.76	162.51	**4.29**	**1,101,721**	1,098,044	**1,101,721**
kentucky-AM3	374.57	997.09	**102.17**	**1,860,308**	1,437,770	**1,860,308**
massachusetts-AM3	0.25	50.59	**0.03**	**115,636**	113,381	**115,636**
oregon-AM3	6.06	239.62	**1.72**	**557,634**	546,314	**557,634**
rhode-island-AM3	34.16	252.86	**9.34**	**1,162,925**	1,162,920	**1,162,925**
vermont-AM3	**0.37**	2.32	0.39	**604,213**	602,793	**604,213**
virginia-AM3	0.20	6.38	**0.05**	**207,457**	**207,457**	**207,457**
washington-AM3	1.88	23.11	**0.96**	**356,314**	**356,314**	**356,314**
Geometric mean						
	4.45	64.29	**1.45**	**542,993**	528,956	**542,993**

Table 6.10: OSM heuristic results

6.6.2 DIMACS Graphs

For the DIMACS graphs (Table 6.11), SCCWalk4l clearly dominates its competitors for all weightings schemes. Between FastWCLq and MWCPeel, FastWCLq mostly computes slightly higher weight solutions, though it takes longer to compute them. Looking at the instances where TSM-MWC fails to find the optimal solution, both FastWCLq and MWCPeel achieve higher weight solutions in a much smaller amount of time for most of them.

Graph	t_{sol} FastWCLq	SCCWalk4l	MWCPeel	$w(\hat{C})$ FastWCLq	SCCWalk4l	MWCPeel
Uniform weighted						
brock800_1	150.98	**0.33**	45.67	3,000	**3,006**	2,886
brock800_2	163.22	73.82	**59.84**	3,024	**3,074**	2,935
brock800_3	127.04	**0.36**	37.69	**2,984**	**2,984**	2,912
brock800_4	230.35	98.56	**48.01**	3,007	**3,059**	2,887
C1000.9	409.16	**1.45**	262.85	8,693	**9,058**	7,779
C2000.5	404.03	**8.25**	536.78	2,426	**2,467**	2,390
C2000.9	351.96	**105.48**	123.18	9,822	**10,874**	8,603
C4000.5	375.71	**82.45**	638.30	2,580	**2,787**	2,472
C500.9	251.22	**0.28**	474.51	7,277	**7,313**	6,964
gen400_p0.9_55	169.17	**0.09**	50.16	**6,781**	**6,781**	6,614
gen400_p0.9_65	312.92	**0.28**	59.97	6,869	**6,881**	6,654
gen400_p0.9_75	104.98	84.15	**10.78**	7,547	**7,551**	7,261
hamming10-4	287.81	**1.42**	527.12	5,727	**5,917**	5,279
johnson32-2-4	459.61	**0.06**	537.90	3,004	**3,042**	3,020
keller5	390.66	**2.14**	326.39	3,811	**3,851**	3,545
keller6	347.22	**216.10**	400.16	6,727	**8,412**	6,103
MANN_a27	16.84	168.23	**1.63**	**17,866**	17,864	17,710
MANN_a45	79.35	**43.89**	93.78	**49,459**	**49,459**	49,312
MANN_a81	180.29	376.13	**7.17**	**161,903**	161,895	161,648
p_hat1000-3	375.61	**0.51**	404.89	8,248	**8,295**	8,223
p_hat1500-2	423.40	**1.22**	475.25	7,519	**7,556**	7,546
p_hat1500-3	356.96	**4.08**	206.37	10,725	**10,926**	10,801
sanr400_0.7	6.94	**0.06**	0.91	**2,926**	**2,926**	2,874
Powerlaw weighted						
brock800_1	42.48	2.08	**0.87**	**1,249**	**1,249**	1,242
brock800_2	6.77	9.25	**0.84**	**1,329**	**1,329**	1,274
brock800_3	8.16	**0.32**	0.78	**1,427**	**1,427**	1,415
brock800_4	14.50	0.99	**0.81**	**1,347**	**1,347**	1,335
C1000.9	384.40	20.23	**4.41**	3,261	**3,291**	3,276
C2000.5	284.59	**1.12**	3.58	1,282	**1,284**	1,275
C2000.9	261.90	**151.81**	592.43	4,027	**4,262**	4,129
C4000.5	455.80	**4.88**	27.26	1,417	**1,521**	1,494
C500.9	178.03	54.30	**0.57**	**2,431**	**2,431**	2,387

gen400_p0.9_55	45.79	29.80	**0.36**	2,109	**2,110**	2,093
gen400_p0.9_65	164.23	43.52	**0.38**	**2,042**	2,039	2,005
gen400_p0.9_75	249.30	15.96	**0.37**	**2,072**	2,055	2,025
hamming10-4	416.49	30.78	**2.32**	2,194	**2,232**	2,214
johnson32-2-4	21.05	**0.11**	0.81	**1,666**	**1,666**	**1,666**
keller5	191.47	1.44	**0.95**	1,590	**1,591**	1,539
keller6	238.74	**55.31**	521.83	2,639	**3,540**	3,249
MANN_a27	0.16	**0.08**	2.76	**4,049**	**4,049**	4,048
MANN_a45	4.70	**1.11**	35.96	**10,617**	**10,617**	10,616
MANN_a81	65.02	**27.68**	62.98	35,084	**35,092**	34,937
p_hat1000-3	325.44	37.05	**2.40**	2,178	**2,199**	2,198
p_hat1500-2	192.27	**1.57**	2.33	2,020	**2,031**	**2,031**
p_hat1500-3	258.36	**7.43**	10.13	2,745	**2,793**	2,788
sanr400_0.7	0.77	2.14	**0.23**	**1,135**	**1,135**	1,131
Exponential weighted						
brock800_1	97.11	**0.49**	2.33	**509**	**509**	504
brock800_2	52.63	**1.31**	2.57	**516**	**516**	502
brock800_3	274.15	**0.74**	3.47	501	**502**	495
brock800_4	189.82	**0.44**	2.73	531	**533**	525
C1000.9	261.31	**93.17**	499.37	1,328	**1,368**	1,225
C2000.5	316.07	**0.77**	20.92	471	**481**	480
C2000.9	272.79	**32.21**	469.85	1,580	**1,731**	1,416
C4000.5	404.33	**5.31**	483.52	508	**547**	518
C500.9	285.87	**1.92**	23.52	1,059	**1,061**	1,043
gen400_p0.9_55	289.76	2.16	**0.50**	946	**949**	914
gen400_p0.9_65	125.29	2.97	**0.65**	944	**945**	935
gen400_p0.9_75	214.58	**0.48**	0.51	**987**	**987**	975
hamming10-4	374.66	**4.75**	445.21	903	**938**	886
johnson32-2-4	11.64	**0.06**	0.95	**643**	**643**	**643**
keller5	392.02	**0.81**	5.05	607	**618**	600
keller6	401.76	**12.03**	599.11	1,035	**1,396**	1,038
MANN_a27	0.50	**0.08**	1.38	**2,257**	**2,257**	2,249
MANN_a45	217.75	**2.95**	122.03	6,284	**6,285**	6,275
MANN_a81	219.77	49.30	**7.07**	20,901	**20,926**	20,810
p_hat1000-3	237.96	**12.69**	240.07	1,018	**1,025**	1,023
p_hat1500-2	287.51	**1.35**	58.95	1,013	**1,020**	**1,020**
p_hat1500-3	346.58	**18.52**	488.84	1,412	**1,443**	1,414
sanr400_0.7	0.46	18.35	**0.23**	**449**	**449**	438
Geometric mean						
	111.01	**3.96**	16.28	2,564	**2,628**	2,527

Table 6.11: DIMACS heuristic results

6.6.3 Network Repository Graphs

As shown in Table 6.12, performance on REP graphs is very competitive among the heuristic solvers. While all algorithms compute the best solution an approximately equal amount of times, the solution quality of SCCWalk4l is the lowest on average. Taking speed into account, MWCPeel shows a good performance in comparison. It should be noted however, that TSM-MWC computes even higher weight solutions faster for most of the instances.

Graph	t_{sol} FastWCLq	SCCWalk4l	MWCPeel	$w(\hat{C})$ FastWCLq	SCCWalk4l	MWCPeel
Uniform weighted						
aff-digg	240.01	**30.73**	47.68	3,514	**3,829**	**3,829**
bio-human-gene1	719.13	640.92	**493.36**	136,581	136,647	**136,713**
bio-human-gene2	457.56	534.75	**89.59**	131,763	131,862	**131,904**
bio-mouse-gene	593.17	412.13	**13.43**	59,439	**59,473**	59,146
sc-TSOPF-RS-b2383	33.02	243.56	**1.43**	**913**	900	870
soc-flickr-und	601.63	252.98	**44.40**	10,806	8,968	**10,847**
soc-orkut	**135.88**	526.16	183.51	**5,832**	4,552	5,582
soc-orkut-dir	**157.93**	521.46	185.02	**5,261**	4,080	5,116
web-wikipedia_link_it	71.06	972.97	**36.59**	**87,175**	2,903	**87,175**
web-wikipedia-growth	**44.03**	343.25	66.06	**3,334**	2,960	3,136
Powerlaw weighted						
aff-digg	260.55	203.20	**31.39**	1,166	**1,369**	**1,369**
bio-human-gene1	350.41	442.01	**142.75**	21,572	21,688	**21,689**
bio-human-gene2	284.12	357.18	**97.32**	19,223	**19,340**	19,300
bio-mouse-gene	390.77	551.27	**27.07**	9,201	**9,230**	9,210
sc-TSOPF-RS-b2383	249.42	**2.38**	5.38	**565**	**565**	525
soc-flickr-und	580.29	240.61	**43.09**	2,514	**2,530**	2,460
soc-orkut	**159.10**	387.90	169.20	**1,420**	1,278	1,417
soc-orkut-dir	**147.52**	500.74	147.82	**1,322**	1,250	**1,322**
web-wikipedia_link_it	65.98	949.06	**45.00**	**13,005**	482	**13,005**
web-wikipedia-growth	69.48	204.15	**48.24**	**921**	907	900
Exponential weighted						
aff-digg	275.54	153.36	**35.14**	491	**581**	**581**
bio-human-gene1	357.68	554.11	**248.47**	15,196	15,254	**15,258**
bio-human-gene2	546.57	496.33	**82.10**	14,856	14,899	**14,903**
bio-mouse-gene	298.26	354.75	**10.40**	6,414	**6,429**	6,380
sc-TSOPF-RS-b2383	247.86	**2.33**	4.18	**222**	**222**	195
soc-flickr-und	615.71	177.61	**48.66**	1,188	**1,198**	**1,198**
soc-orkut	**164.46**	664.54	173.35	**735**	597	672
soc-orkut-dir	466.39	428.66	**165.67**	**674**	562	633
web-wikipedia_link_it	68.53	1,032.91	**49.29**	9,616	364	**10,236**
web-wikipedia-growth	114.46	324.19	**57.94**	**414**	391	384
Geometric mean						
	218.15	265.90	**51.83**	**4,378**	3,038	4,355

Table 6.12: REP graphs heuristic results

6.6.4 Random Hyperbolic Graphs

The results for the RHGs are presented in Table 6.13. Here, MWCPeel outperforms the other solvers in 31 out of 39 instances. While FastWCLq sometimes finds a slightly higher weight solution than MWCPeel, it has a higher running time on average. SCCWalk4l is clearly outperformed both in speed and solution quality.

Graph	t_{sol}			$w(\hat{C})$		
	FastWCLq	SCCWalk4l	MWCPeel	FastWCLq	SCCWalk4l	MWCPeel
Uniform weighted						
rhg_250000_100_1.75	10.56	134.23	**2.67**	**99,839**	**99,839**	**99,839**
rhg_250000_100_2.25	2.76	59.67	**2.09**	**37,947**	**37,947**	**37,947**
rhg_250000_250_1.75	42.01	501.91	**4.24**	**112,769**	112,074	112,756
rhg_250000_250_2.25	9.82	204.35	**4.58**	**71,001**	**71,001**	**71,001**
rhg_250000_500_1.75	125.03	869.09	**18.39**	**137,234**	86,773	136,884
rhg_250000_500_2.25	36.98	624.17	**10.15**	**102,364**	100,087	**102,364**
rhg_500000_250_1.75	90.36	854.58	**22.69**	**131,559**	67,352	131,100
rhg_500000_250_2.25	22.04	647.54	**11.40**	**88,512**	85,244	**88,512**
rhg_500000_500_2.25	71.83	995.47	**15.33**	**122,781**	46,201	**122,781**
rhg_750000_250_1.75	122.79	998.01	**22.72**	**160,845**	36,006	**160,845**
rhg_750000_250_2.25	28.23	687.98	**17.00**	**96,362**	95,558	**96,362**
rhg_750000_500_1.75	375.30	1,017.89	**21.41**	**207,197**	27,079	**207,197**
rhg_750000_500_2.25	72.82	1,002.84	**36.00**	**119,936**	43,470	**119,936**
Powerlaw weighted						
rhg_250000_100_1.75	11.06	92.52	**3.17**	**14,183**	**14,183**	**14,183**
rhg_250000_100_2.25	2.11	48.48	**2.04**	**5,801**	5,783	**5,801**
rhg_250000_250_1.75	27.71	321.83	**5.24**	**16,771**	16,516	**16,771**
rhg_250000_250_2.25	8.13	142.50	**4.63**	**10,541**	10,512	**10,541**
rhg_250000_500_1.75	77.55	825.63	**17.32**	**20,188**	20,093	20,140
rhg_250000_500_2.25	23.04	455.24	**12.46**	**14,720**	14,556	**14,720**
rhg_500000_250_1.75	52.87	763.45	**19.29**	**19,356**	17,332	19,212
rhg_500000_250_2.25	17.22	484.21	**11.87**	**12,997**	12,856	**12,997**
rhg_500000_500_2.25	44.85	838.71	**22.28**	**18,100**	17,948	**18,100**
rhg_750000_250_1.75	80.25	998.05	**27.64**	**22,720**	7,428	**22,720**
rhg_750000_250_2.25	23.56	629.66	**17.01**	**14,014**	13,730	**14,014**
rhg_750000_500_1.75	280.47	1,027.30	**48.08**	**29,071**	3,897	**29,071**
rhg_750000_500_2.25	64.95	999.31	**34.75**	**17,746**	8,528	**17,746**
Exponential weighted						
rhg_250000_100_1.75	11.22	260.80	**2.33**	**10,808**	10,652	**10,808**
rhg_250000_100_2.25	2.43	54.90	**1.86**	**4,270**	**4,270**	**4,270**
rhg_250000_250_1.75	34.48	396.15	**4.92**	**12,368**	11,321	**12,368**
rhg_250000_250_2.25	9.59	245.29	**4.74**	**7,930**	7,769	**7,930**
rhg_250000_500_1.75	89.77	810.86	**19.47**	**15,189**	13,046	15,157
rhg_250000_500_2.25	26.08	501.97	**11.68**	**11,225**	11,220	**11,225**
rhg_500000_250_1.75	75.88	779.39	**22.81**	**14,518**	11,303	14,488

rhg_500000_250_2.25	21.02	535.28	**11.62**	**9,675**	**9,675**	**9,675**
rhg_500000_500_2.25	57.95	914.79	**22.31**	**13,894**	12,167	**13,894**
rhg_750000_250_1.75	101.71	999.02	**28.84**	**17,391**	4,475	**17,391**
rhg_750000_250_2.25	27.01	603.72	**17.66**	**10,849**	10,564	**10,849**
rhg_750000_500_1.75	268.55	1,034.19	**50.97**	**22,404**	2,816	**22,404**
rhg_750000_500_2.25	72.35	998.22	**30.40**	**13,387**	4,984	13,386
Geometric mean						
	34.59	479.59	**11.62**	**26,980**	17,956	26,966

Table 6.13: RHG heuristic results

Figure 6.3 shows the solution quality of the heuristic solvers relative to the respective best value. The most notable observation here is the drop in performance of SCCWalk4l for multiple instances, while both FastWCLq and MWCRedu maintain a consistent solution quality.

The time to get to the best solution is compared in Figure 6.4. It shows again, that except for DIMACS, where the exact reductions are not applicable, MWCRedu performs very well in comparison with the other solvers.

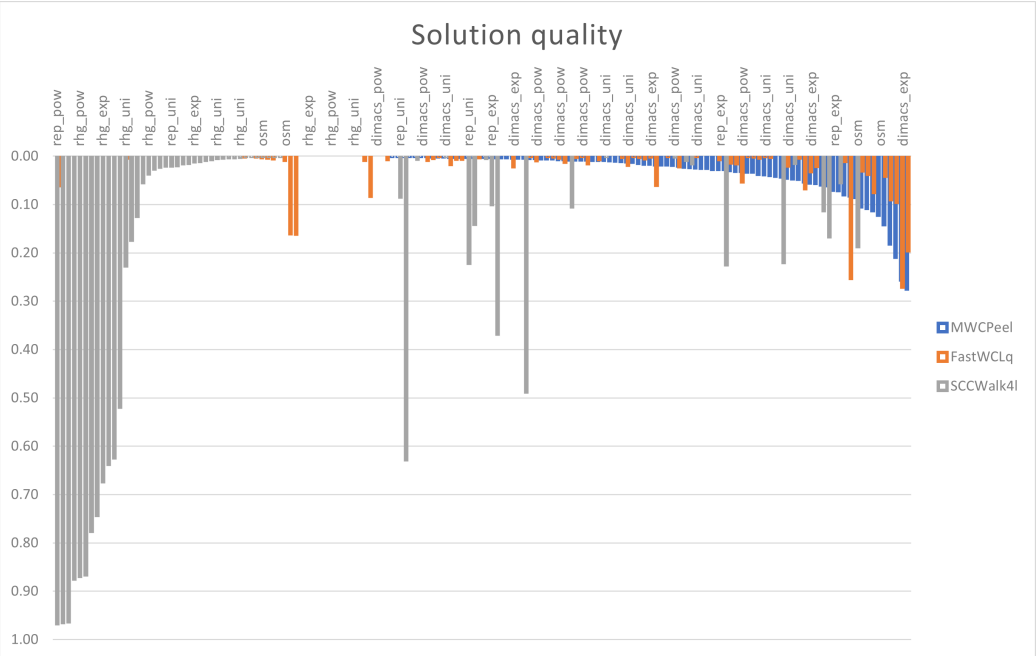

Figure 6.3: Solution quality for heuristic solvers.

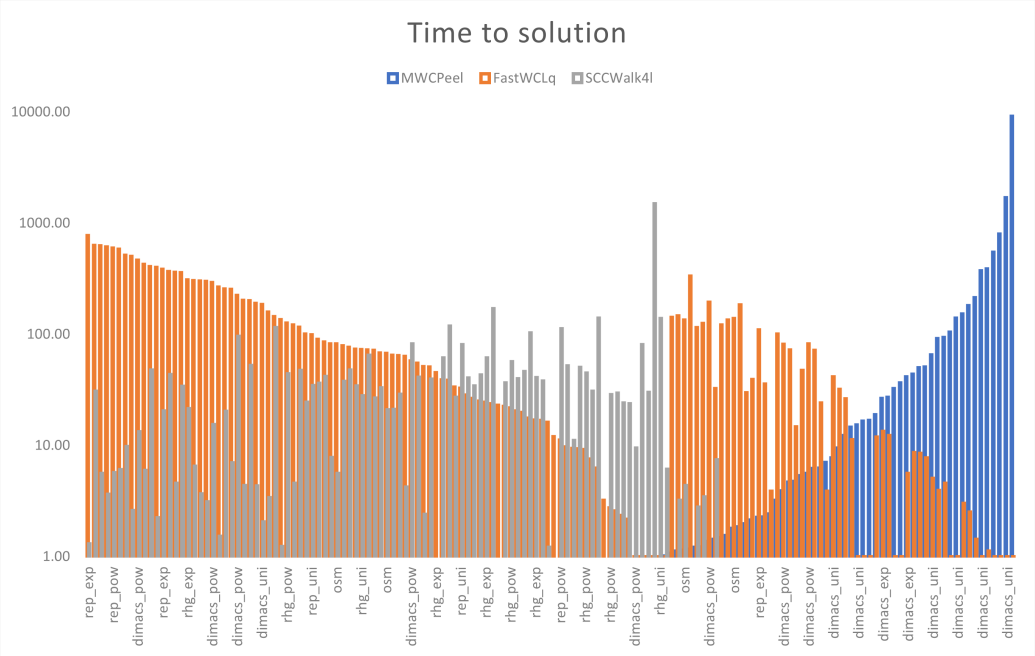

Figure 6.4: Time to solution for heuristic solvers.

CHAPTER 7

Conclusion

In this work, we proposed both exact and heuristic algorithms for the MWC problem, that combine effective techniques from previously seen solvers with novel graph reduction techniques that significantly reduce the size of the input graph for many instances. As the experiments show, especially the exact solver MWCRedu shows significant improvements over the state-of-the art solver TSM-MWC. For graphs where the reduction rules are especially effective, such as graphs resembling the structure of the RHG or OSM instances from the experiments, MWCRedu clearly outperforms TSM-MWC. Even in the case that the reduction cannot be applied on the input instance, MWCRedu still sometimes reaches higher weight solutions due to the local search algorithm. TSM-MWC only consistently reaches the optimal solution faster for very large graphs, where the reductions employed by MWCRedu are applicable but relatively inefficient. Overall, as TSM-MWC fails to compute the optimal solution several times more often than MWCRedu and takes longer on average to find and prove the optimal solution, MWCRedu appears to be the more consistent of the two.
We furthermore extend the research for heuristic algorithms for MWC, by investigating machine learning approaches, as well as a heuristic rule for reducing the graph. The heuristic solver MWCPeel, implementing both the exact reduction rules from MWCRedu and the novel heuristic reduction rule, proves to be competitive with the state-of-the-art solvers FastWCLq and SCCWalk4l. It clearly outperforms both competitors for most of the OSM and RHG instances, especially when it comes to the time to compute the solution. While it is outperformed by SCCWalk4l for the DIMACS instances, MWCPeel reaches significantly higher weight solutions on average. FastWCLq achieves higher quality solutions on average, but usually takes longer to get there. It therefore depends on the use-case which solver should be employed, depending on whether speed or solution quality is more important. The approach of scoring the vertices using machine learning models for the heuristic reductions also shows potential, but is not yet able to fully capture the complex nature of the MWC problem. We believe that given more complex models and/or features, machine learning will be a natural extension to many of the presented techniques.

7.1 Future Work

Among the ideas presented in this work, many can be extended and evolved into new techniques to further speed up the solver. The first extension that might be worth looking into concerns the heuristic vertex removal rule. Specifically, as exact edge reduction rules have proven very effective in reducing some graph instances, the peeling reduction can be extended to also remove edges according to some heuristic criterion, further reducing the input instance and speeding up solvers applied on the reduced graph. Possible choices for the criterion could be the combined neighborhood weight of the endpoints, the weight of their common neighborhood or some score assigned by a machine learning model. As the understanding and development of machine learning continues to grow, it would also be interesting to evaluate more powerful models on the MWC problem. A prediction that is correlated to the potential of a vertex to be in the optimal solution can be used not only to improve the heuristic vertex reduction rule, but also, e.g., as an evaluation function for BMS in local search algorithms, or as an ordering for exact B&B solvers. For the heuristic solver, it would also be possible to use some state-of-the-art local search algorithm, such as the one employed in FastWCLq or SCCWalk4l, to look for higher weight solutions in the reduced graph instead of applying B&B, possibly reaching a high quality solution faster. Lastly, an extension that is certain to further speed up MWCRedu and MWCPeel is parallelization. This could be done, e.g., by partitioning the graph and applying the reductions on each partition in parallel.

Appendix

Instance	$\|V\|$	$\|E\|$	ρ	d_{min}	d_{max}	d_{avg}
vermont-AM3	3,436	1,136,164	3.27×10^{-1}	1	1,608	661.33
massachusetts-AM3	3,703	551,491	1.10×10^{-1}	1	1,188	297.86
idaho-AM3	4,064	3,924,080	4.75×10^{-1}	1	3,332	1,931.14
greenland-AM3	4,986	3,652,361	1.74×10^{-2}	1	3,354	1,465.05
oregon-AM3	5,588	2,912,701	2.60×10^{-2}	1	2,906	1,042.48
virginia-AM3	6,185	665,903	3.48×10^{-2}	1	775	215.33
washington-AM3	10,022	2,346,213	4.67×10^{-2}	1	1,986	468.21
district-of-columbia-AM2	13,597	1,609,795	8.05×10^{-2}	1	1,126	236.79
rhode-island-AM3	15,124	12,622,219	1.26×10^{-1}	1	5,930	1,669.17
kentucky-AM3	19,095	59,533,630	2.94×10^{-1}	1	14,928	6,235.51
hawaii-AM3	28,006	49,444,921	1.87×10^{-1}	1	10,313	3,531.02
district-of-columbia-AM3	46,221	27,729,137	1.93×10^{-1}	1	5,940	1,199.85

Table 7.1: OSM graphs

Instance	w_{min}	w_{max}	w_{avg}
vermont-AM3	33	2,539	648.90
massachusetts-AM3	33	2,731	324.38
idaho-AM3	33	2,506	547.21
greenland-AM3	33	1,029	324.32
oregon-AM3	33	2,392	432.76
virginia-AM3	33	3,048	659.48
washington-AM3	33	3,672	523.41
district-of-columbia-AM2	33	1,389	292.83
rhode-island-AM3	33	2,939	469.78
kentucky-AM3	33	2,825	292.36
hawaii-AM3	33	3,628	328.71
district-of-columbia-AM3	33	1,389	209.55

Table 7.2: OSM graphs weight distribution

Instance	$\|V\|$	$\|E\|$	ρ	d_{min}	d_{max}	d_{avg}
MANN_a27	378	70,551	9.90×10^{-1}	364	374	373.29
sanr400_0.7	400	55,869	7.00×10^{-1}	252	310	279.35
gen400_p0.9_55	400	71,820	9.00×10^{-1}	334	375	359.10
gen400_p0.9_65	400	71,820	9.00×10^{-1}	333	378	359.10
gen400_p0.9_75	400	71,820	9.00×10^{-1}	335	380	359.10
johnson32-2-4	496	107,880	8.79×10^{-1}	435	435	435.00
C500.9	500	112,332	9.00×10^{-1}	431	468	449.33
keller5	776	225,990	7.52×10^{-1}	560	638	582.45
brock800_3	800	207,333	6.49×10^{-1}	474	558	518.33
brock800_1	800	207,505	6.49×10^{-1}	477	560	518.76
brock800_4	800	207,643	6.50×10^{-1}	481	565	519.11
brock800_2	800	208,166	6.51×10^{-1}	472	566	520.42
p_hat1000-3	1,000	371,746	7.44×10^{-1}	582	895	743.49
C1000.9	1,000	450,079	9.01×10^{-1}	868	925	900.16
hamming10-4	1,024	434,176	8.29×10^{-1}	848	848	848.00
MANN_a45	1,035	533,115	9.96×10^{-1}	1,012	1,031	1,030.17
p_hat1500-2	1,500	568,960	5.06×10^{-1}	335	1,153	758.61
p_hat1500-3	1,500	847,244	7.54×10^{-1}	912	1,330	1,129.66
C2000.5	2,000	999,836	5.00×10^{-1}	919	1,074	999.84
C2000.9	2,000	1,799,532	9.00×10^{-1}	1,751	1,848	1,799.53
MANN_a81	3,321	5,506,380	9.99×10^{-1}	3,280	3,317	3,316.10
keller6	3,361	4,619,898	8.18×10^{-1}	2,690	2,952	2,749.12
C4000.5	4,000	4,000,268	5.00×10^{-1}	1,895	2,123	2,000.13

Table 7.3: DIMACS graphs

Instance	w_{min}	w_{max}	w_{avg}
MANN_a27	1	200	98.78
sanr400_0.7	1	200	98.69
gen400_p0.9_55	1	200	98.69
gen400_p0.9_65	1	200	98.69
gen400_p0.9_75	1	200	98.69
johnson32-2-4	1	200	97.99
C500.9	1	200	98.00
keller5	1	200	98.54
brock800_3	1	200	98.62
brock800_1	1	200	98.62
brock800_4	1	200	98.62
brock800_2	1	200	98.62
p_hat1000-3	1	200	99.29
C1000.9	1	200	99.29
hamming10-4	1	200	99.41
MANN_a45	1	200	99.46
p_hat1500-2	1	200	98.68
p_hat1500-3	1	200	98.68
C2000.5	1	200	98.83
C2000.9	1	200	98.83
MANN_a81	1	200	99.64
keller6	1	200	99.68
C4000.5	1	200	99.76

Table 7.4: DIMACS graphs with uniform weighting scheme

Instance	w_{min}	w_{max}	w_{avg}
MANN_a27	1	189	14.43
sanr400_0.7	1	189	14.82
gen400_p0.9_55	1	189	14.82
gen400_p0.9_65	1	189	14.82
gen400_p0.9_75	1	189	14.82
johnson32-2-4	1	190	14.66
C500.9	1	190	14.59
keller5	1	196	14.31
brock800_3	1	196	14.32
brock800_1	1	196	14.32
brock800_4	1	196	14.32
brock800_2	1	196	14.32
p_hat1000-3	1	197	14.10
C1000.9	1	197	14.10
hamming10-4	1	197	14.07
MANN_a45	1	197	14.04
p_hat1500-2	1	199	13.68
p_hat1500-3	1	199	13.68
C2000.5	1	199	13.82
C2000.9	1	199	13.82
MANN_a81	1	199	14.17
keller6	1	199	14.16
C4000.5	1	199	14.04

Table 7.5: DIMACS graphs with powerlaw weighting scheme

Instance	w_{min}	w_{max}	w_{avg}
MANN_a27	1	107	10.85
sanr400_0.7	1	107	10.77
gen400_p0.9_55	1	107	10.77
gen400_p0.9_65	1	107	10.77
gen400_p0.9_75	1	107	10.77
johnson32-2-4	1	107	10.99
C500.9	1	107	10.97
keller5	1	107	10.85
brock800_3	1	107	10.82
brock800_1	1	107	10.82
brock800_4	1	107	10.82
brock800_2	1	107	10.82
p_hat1000-3	1	107	10.78
C1000.9	1	107	10.78
hamming10-4	1	107	10.79
MANN_a45	1	107	10.80
p_hat1500-2	1	107	10.84
p_hat1500-3	1	107	10.84
C2000.5	1	107	10.80
C2000.9	1	107	10.80
MANN_a81	1	107	11.04
keller6	1	107	11.05
C4000.5	1	107	10.99

Table 7.6: DIMACS graph instances with exponential weighting scheme

Instance	$\|V\|$	$\|E\|$	ρ	d_{min}	d_{max}	d_{avg}
bio-human-gene2	14,340	9,027,024	8.78×10^{-2}	0	7,228	1,259.00
bio-human-gene1	22,283	12,323,680	4.96×10^{-2}	0	7,938	1,106.10
sc-TSOPF-RS-b2383	38,121	16,115,324	2.22×10^{-2}	0	16,353	845.48
bio-mouse-gene	45,101	14,461,095	1.42×10^{-2}	0	8,031	641.27
aff-digg	872,622	22,501,699	5.91×10^{-5}	0	75,587	51.19
soc-flickr-und	1,715,256	15,555,040	1.06×10^{-5}	0	27,236	17.78
web-wikipedia-growth	1,870,710	36,532,530	2.09×10^{-5}	0	226,073	38.98
web-wikipedia_link_it	2,936,414	86,754,663	2.01×10^{-5}	0	825,147	58.53
soc-orkut-dir	2,997,167	106,349,208	2.37×10^{-5}	0	27,466	70.85
soc-orkut	3,072,442	117,185,082	2.48×10^{-5}	0	33,313	76.17

Table 7.7: Real-world graphs

Instance	w_{min}	w_{max}	w_{avg}
bio-human-gene2	1	200	99.98
bio-human-gene1	1	200	100.08
sc-TSOPF-RS-b2383	1	200	100.31
bio-mouse-gene	1	200	100.33
aff-digg	1	200	100.48
soc-flickr-und	1	200	100.48
web-wikipedia-growth	1	200	100.48
web-wikipedia_link_it	1	200	100.43
soc-orkut-dir	1	200	100.43
soc-orkut	1	200	100.43

Table 7.8: Real-world graphs with uniform weighting scheme

Instance	w_{min}	w_{max}	w_{avg}
bio-human-gene2	1	199	14.12
bio-human-gene1	1	199	14.05
sc-TSOPF-RS-b2383	1	199	14.05
bio-mouse-gene	1	199	14.07
aff-digg	1	199	14.10
soc-flickr-und	1	199	14.09
web-wikipedia-growth	1	199	14.09
web-wikipedia_link_it	1	199	14.04
soc-orkut-dir	1	199	14.03
soc-orkut	1	199	14.03

Table 7.9: Real-world graphs with powerlaw weighting scheme

Instance	w_{min}	w_{max}	w_{avg}
bio-human-gene2	1	110	10.96
bio-human-gene1	1	110	10.94
sc-TSOPF-RS-b2383	1	122	10.98
bio-mouse-gene	1	122	10.97
aff-digg	1	179	10.99
soc-flickr-und	1	179	11.00
web-wikipedia-growth	1	179	11.00
web-wikipedia_link_it	1	179	11.00
soc-orkut-dir	1	179	11.00
soc-orkut	1	179	11.00

Table 7.10: Real-world graphs with exponential weighting scheme

Instance	$\|V\|$	$\|E\|$	ρ	d_{min}	d_{max}	d_{avg}
rhg_250000_100_1.75	250,000	7,755,473	2.48×10^{-4}	0	145,773	62.04
rhg_250000_100_2.25	250,000	10,546,938	3.38×10^{-4}	6	77,509	84.38
rhg_250000_250_1.75	250,000	17,828,988	5.71×10^{-4}	3	206,689	142.63
rhg_250000_250_2.25	250,000	24,036,880	7.69×10^{-4}	19	119,003	192.29
rhg_250000_500_1.75	250,000	35,161,098	1.13×10^{-3}	11	225,762	281.29
rhg_250000_500_2.25	250,000	47,230,197	1.51×10^{-3}	45	210,238	377.84
rhg_500000_250_1.75	500,000	35,493,799	2.84×10^{-4}	1	333,324	141.97
rhg_500000_250_2.25	500,000	49,954,694	4.00×10^{-4}	21	343,333	199.82
rhg_500000_500_2.25	500,000	92,901,492	7.43×10^{-4}	47	216,025	371.61
rhg_750000_250_1.75	750,000	53,201,080	1.89×10^{-4}	1	676,852	141.87
rhg_750000_250_2.25	750,000	73,667,026	2.62×10^{-4}	19	339,764	196.44
rhg_750000_500_1.75	750,000	102,363,505	3.64×10^{-4}	8	506,104	272.81
rhg_750000_500_2.25	750,000	139,633,569	4.96×10^{-4}	46	425,575	372.34

Table 7.11: RHG graphs

Instance	w_{min}	w_{max}	w_{avg}
rhg_250000_100_1.75	1	200	100.49
rhg_250000_100_2.25	1	200	100.49
rhg_250000_250_1.75	1	200	100.49
rhg_250000_250_2.25	1	200	100.49
rhg_250000_500_1.75	1	200	100.49
rhg_250000_500_2.25	1	200	100.49
rhg_500000_250_1.75	1	200	100.48
rhg_500000_250_2.25	1	200	100.48
rhg_500000_500_2.25	1	200	100.48
rhg_750000_250_1.75	1	200	100.48
rhg_750000_250_2.25	1	200	100.48
rhg_750000_500_1.75	1	200	100.48
rhg_750000_500_2.25	1	200	100.48

Table 7.12: RHG graphs with uniform weighting scheme

Instance	w_{min}	w_{max}	w_{avg}
rhg_250000_100_1.75	1	199	14.11
rhg_250000_100_2.25	1	199	14.11
rhg_250000_250_1.75	1	199	14.11
rhg_250000_250_2.25	1	199	14.11
rhg_250000_500_1.75	1	199	14.11
rhg_250000_500_2.25	1	199	14.11
rhg_500000_250_1.75	1	199	14.09
rhg_500000_250_2.25	1	199	14.09
rhg_500000_500_2.25	1	199	14.09
rhg_750000_250_1.75	1	199	14.10
rhg_750000_250_2.25	1	199	14.10
rhg_750000_500_1.75	1	199	14.10
rhg_750000_500_2.25	1	199	14.10

Table 7.13: RHG graphs with powerlaw weighting scheme

Instance	w_{min}	w_{max}	w_{avg}
rhg_250000_100_1.75	1	138	11.00
rhg_250000_100_2.25	1	138	11.00
rhg_250000_250_1.75	1	138	11.00
rhg_250000_250_2.25	1	138	11.00
rhg_250000_500_1.75	1	138	11.00
rhg_250000_500_2.25	1	138	11.00
rhg_500000_250_1.75	1	138	10.99
rhg_500000_250_2.25	1	138	10.99
rhg_500000_500_2.25	1	138	10.99
rhg_750000_250_1.75	1	179	10.99
rhg_750000_250_2.25	1	179	10.99
rhg_750000_500_1.75	1	179	10.99
rhg_750000_500_2.25	1	179	10.99

Table 7.14: RHG graphs with exponential weighting scheme

Abstract (German)

Das Maximum Weighted Clique Problem (MWC) ist ein bekanntes Problem der Graphentheorie mit vielen praktischen Anwendungen. In dieser Arbeit stellen wir sowohl exakte als auch heuristische Algorithmen vor, die erfolgreiche Techniken aus verwandten Arbeiten mit neuen Graphenreduktionen kombinieren. Während für MWC in der Vergangenheit ausschließlich Graphenreduktionen basierend auf oberen Schranken verwendet wurden, präsentieren wir Reduktionsregeln, die lokale Graphenstrukturen ausnutzen, um Knoten und Kanten zu identifizieren und zu entfernen, ohne die optimale Lösung zu reduzieren. Ein Satz exakter Reduktionsregeln wird in einem exakten Algorithmus namens MWCRedu verwendet, während heuristische Reduktionstechniken basierend auf maschinellen Lernmodellen wie MLP, Deepset und GNN im heuristischen Framework MWCPeel untersucht werden. Experimente mit einer breiten Palette von Graphen zeigen, dass MWCRedu den exakten Algorithmus TSM-MWC [25], welcher den aktuellen Stand der Technik repräsentiert, für die meisten Grapheninstanzen übertrifft. Insbesondere für mittelgroße, gewichtete Graphen, die Straßennetzwerke modellieren, und zufällig generierte hyperbolische Graphen, die als gute Annäherung an Graphen der realen Welt gelten, ist MWCRedu um Größenordnungen schneller. MWCPeel übertrifft seine Konkurrenten FastWCLq [12] und SCCWalk4l [47] ebenfalls auf diesen Instanzen, ist jedoch auf extrem dichten oder großen Instanzen etwas weniger effektiv.

Bibliography

[1] Faisal Abu-Khzam, Sebastian Lamm, Matthias Mnich, Alexander Noe, Christian Schulz, and Darren Strash. “Recent advances in practical data reduction”. In: *arXiv preprint arXiv:2012.12594* (2020).

[2] Abien Fred Agarap. “Deep learning using rectified linear units (relu)”. In: *arXiv preprint arXiv:1803.08375* (2018).

[3] Takuya Akiba and Yoichi Iwata. “Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover”. In: *Theoretical Computer Science* 609 (2016), pp. 211–225.

[4] Luitpold Babel and Gottfried Tinhofer. “A branch and bound algorithm for the maximum clique problem”. In: *Zeitschrift für Operations Research* 34.3 (1990), pp. 207–217.

[5] Roberto Battiti and Giampietro Tecchiolli. “The reactive tabu search”. In: *ORSA journal on computing* 6.2 (1994), pp. 126–140.

[6] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. “Efficient semi-streaming algorithms for local triangle counting in massive graphs”. In: *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2008, pp. 16–24.

[7] Una Benlic and Jin-Kao Hao. “Breakout local search for maximum clique problems”. In: *Computers & Operations Research* 40.1 (2013), pp. 192–206.

[8] Burton H Bloom. “Space/time trade-offs in hash coding with allowable errors”. In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[9] Sergiy Butenko and Wilbert E Wilhelm. “Clique-detection models in computational biochemistry and genomics”. In: *European Journal of Operational Research* 173.1 (2006), pp. 1–17.

[10] Shaowei Cai. “Balance between complexity and quality: Local search for minimum vertex cover in massive graphs”. In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015.

[11] Shaowei Cai and Jinkun Lin. “Fast Solving Maximum Weight Clique Problem in Massive Graphs.” In: *IJCAI*. 2016, pp. 568–574.

[12] Shaowei Cai, Jinkun Lin, Yiyuan Wang, and Darren Strash. “A semi-exact algorithm for quickly computing a maximum weight clique in large sparse graphs”. In: *Journal of Artificial Intelligence Research* 72 (2021), pp. 39–67.

[13] Randy Carraghan and Panos M Pardalos. “An exact algorithm for the maximum clique problem”. In: *Operations Research Letters* 9.6 (1990), pp. 375–382.

[14] Lijun Chang, Wei Li, and Wenjie Zhang. “Computing a near-maximum independent set in linear time by reducing-peeling”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 1181–1196.

[15] Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F Werneck. “Accelerating local search for the maximum independent set problem”. In: *International symposium on experimental algorithms*. Springer. 2016, pp. 118–133.

[16] Timothy A Davis and Yifan Hu. “The University of Florida sparse matrix collection”. In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–25.

[17] Zhiwen Fang, Chu-Min Li, and Ke Xu. “An exact algorithm based on maxsat reasoning for the maximum weight clique problem”. In: *Journal of Artificial Intelligence Research* 55 (2016), pp. 799–833.

[18] Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. “Communication-free massively distributed graph generation”. In: *Journal of Parallel and Distributed Computing* 131 (2019), pp. 200–217.

[19] Alexander Gellner, Sebastian Lamm, Christian Schulz, Darren Strash, and Bogdán Zaválnij. “Boosting Data Reduction for the Maximum Weight Independent Set Problem Using Increasing Transformations”. In: *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2021, pp. 128–142.

[20] Michel Gendreau, Patrick Soriano, and Louis Salvail. “Solving the maximum clique problem using a tabu search approach”. In: *Annals of operations research* 41.4 (1993), pp. 385–403.

[21] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. “Neural message passing for quantum chemistry”. In: *International conference on machine learning*. PMLR. 2017, pp. 1263–1272.

[22] Ernestine Großmann, Sebastian Lamm, Christian Schulz, and Darren Strash. “Finding Near-Optimal Weight Independent Sets at Scale”. In: *arXiv preprint arXiv:2208.13645* (2022).

[23] Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. “Random hyperbolic graphs: degree sequence and clustering”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2012, pp. 573–585.

[24] Demian Hespe, Christian Schulz, and Darren Strash. "Scalable kernelization for maximum independent sets". In: *Journal of Experimental Algorithmics (JEA)* 24 (2019), pp. 1–22.

[25] Hua Jiang, Chu-Min Li, Yanli Liu, and Felip Manya. "A two-stage maxsat reasoning approach for the maximum weight clique problem". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[26] Hua Jiang, Chu-Min Li, and Felip Manya. "An exact algorithm for the maximum weight clique problem in large graphs". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 31. 1. 2017.

[27] David S Johnson and Michael A Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*. Vol. 26. American Mathematical Soc., 1996.

[28] Richard M Karp. "Reducibility among combinatorial problems". In: *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[29] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks". In: *arXiv preprint arXiv:1609.02907* (2016).

[30] Janez Konc and Dušanka Janezic. "An improved branch and bound algorithm for the maximum clique problem". In: *proteins* 4.5 (2007), pp. 590–596.

[31] Deniss Kumlander. "A new exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring and a backtrack search". In: *Proc. 5th Int'l Conf. on Modelling, Computation and Optimization in Information Systems and Management Sciences*. Citeseer. 2004, pp. 202–208.

[32] 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: *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2019, pp. 144–158.

[33] Kenneth Langedal, Johannes Langguth, Fredrik Manne, and Daniel Thilo Schroeder. "Efficient Minimum Weight Vertex Cover Heuristics Using Graph Neural Networks". In: *20th International Symposium on Experimental Algorithms (SEA 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.

[34] Chu Min Li and Felip Manya. "MaxSAT, hard and soft constraints". In: *Handbook of satisfiability*. IOS Press, 2021, pp. 903–927.

[35] Chu-Min Li and Zhe Quan. "An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem". In: *Twenty-fourth AAAI conference on artificial intelligence*. 2010.

[36] Kurt Mehlhorn, Peter Sanders, and Peter Sanders. *Algorithms and data structures: The basic toolbox*. Vol. 55. Springer, 2008.

[37] Ingo Muegge and Matthias Rarey. “Small molecule docking and scoring”. In: *Reviews in computational chemistry* 17 (2001), pp. 1–60.

[38] Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. “Recent advances in scalable network generation”. In: *arXiv preprint arXiv:2003.00736* (2020).

[39] Wayne Pullan. “Approximating the maximum vertex/edge weighted clique using local search”. In: *Journal of Heuristics* 14.2 (2008), pp. 117–134.

[40] Wayne Pullan. “Phased local search for the maximum clique problem”. In: *Journal of Combinatorial Optimization* 12.3 (2006), pp. 303–323.

[41] Wayne Pullan, Franco Mascia, and Mauro Brunato. “Cooperating local search for the maximum clique problem”. In: *Journal of Heuristics* 17.2 (2011), pp. 181–199.

[42] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.

[43] Ryan A. Rossi and Nesreen K. Ahmed. “The Network Data Repository with Interactive Graph Analytics and Visualization”. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015.

[44] Yuan Sun, Xiaodong Li, and Andreas Ernst. “Using statistical measures and machine learning for graph reduction to solve maximum weight clique problems”. In: *IEEE transactions on pattern analysis and machine intelligence* 43.5 (2019), pp. 1746–1760.

[45] Etsuji Tomita and Tomokazu Seki. “An efficient branch-and-bound algorithm for finding a maximum clique”. In: *International conference on discrete mathematics and theoretical computer science*. Springer. 2003, pp. 278–289.

[46] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. “A simple and faster branch-and-bound algorithm for finding a maximum clique”. In: *International Workshop on Algorithms and Computation*. Springer. 2010, pp. 191–203.

[47] Yiyuan Wang, Shaowei Cai, Jiejiang Chen, and Minghao Yin. “SCCWalk: An efficient local search algorithm and its improvements for maximum weight clique problem”. In: *Artificial Intelligence* 280 (2020), p. 103230.

[48] Yiyuan Wang, Shaowei Cai, and Minghao Yin. “Two efficient local search algorithms for maximum weight clique problem”. In: *Thirtieth AAAI Conference on Artificial Intelligence*. 2016.

[49] Stanley Wasserman, Katherine Faust, et al. “Social network analysis: Methods and applications”. In: (1994).

[50] Qinghua Wu and Jin-Kao Hao. “A review on algorithms for maximum clique problems”. In: *European Journal of Operational Research* 242.3 (2015), pp. 693–709.

[51] Qinghua Wu and Jin-Kao Hao. “Solving the winner determination problem via a weighted maximum clique heuristic”. In: *Expert Systems with Applications* 42.1 (2015), pp. 355–365.

[52] Qinghua Wu, Jin-Kao Hao, and Fred Glover. “Multi-neighborhood tabu search for the maximum weight clique problem”. In: *Annals of Operations Research* 196.1 (2012), pp. 611–634.

[53] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. “How powerful are graph neural networks?” In: *arXiv preprint arXiv:1810.00826* (2018).

[54] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. “Deep sets”. In: *Advances in neural information processing systems* 30 (2017).

[55] Dong Zhang, Omar Javed, and Mubarak Shah. “Video object co-segmentation by regulated maximum weight cliques”. In: *European Conference on Computer Vision*. Springer. 2014, pp. 551–566.

[56] Hootan Zhian, Masoud Sabaei, Nastooh Taheri Javan, and Omid Tavallaie. “Increasing coding opportunities using maximum-weight clique”. In: *2013 5th Computer Science and Electronic Engineering Conference (CEEC)*. IEEE. 2013, pp. 168–173.