Technische Universität Berlin
Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Algorithmics and Computational Complexity (AKT)

On Efficient Cut-Based Data Reduction for Weighted Cluster Editing

Hjalmar Schulz

Thesis submitted in fulfillment of the requirements for the degree
"Bachelor of Science" (B. Sc.) in the field of Computer Science

11 2021

Supervisor and first reviewer: Prof. Dr. Rolf Niedermeier
Second reviewer: Prof. Dr. Sebastian Siebertz
Co-Supervisors: Dr. André Nichterlein

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

The independent and unaided completion of the thesis is affirmed by affidavit:

Berlin, _____ _____

Datum Unterschrift

Zusammenfassung

In dieser Arbeit wird sich mit dem NP-schweren Problem Weighted Cluster Editing befasst. Dieses nutzt gewichtete Graphen, bei denen ein Kantengewicht die Kosten beschreibt, die beim Löschen bzw. Einfügen dieser Kante entstehen. Weighted Cluster Editing stellt die Frage, ob es möglich ist, den Eingabegraphen durch das Einfügen und/oder Löschen von Kanten in einen Clustergraphen zu transformieren, wobei die Gesamtkosten maximal k betragen dürfen. Ein Clustergraph ist ein Graph, bei dem jede Zusammenhangskomponente eine Clique ist. Genauer wird sich in dieser Arbeit auf eine Datenreduktionsregel für Weighted Cluster Editing konzentriert, bei der bisher nur heuristische Ansätze bekannt waren, um die Regel anzuwenden. Zunächst wird die Korrektheit der Regel gezeigt, gefolgt von Eigenschaften von Graphen, basierend auf minimalen Schnitten in Graphen. Diese Eigenschaften führen zu einem rekursiven Polynomzeitalgorithmus, der die Datenreduktionsregel erschöpfend ausführt. Anschließend wird die praktische Laufzeit des Algorithmus optimiert, indem Eigenschaften von bestimmten Knotenmengen ausgenutzt werden. Zum Schluss wird die Implementierung des Algorithmus präsentiert und mit den bekannten heuristischen Ansätzen verglichen. Basierend auf den Ergebnissen werden Stärken und Schwächen des Algorithmus analysiert und diskutiert. Insgesamt lässt sich festhalten, dass die Datenreduktionsregel effektiv ist. Durchschnittlich wurde die Größe der Graphen, basierend auf den biologischen Daten in dem *PACE Challenge 2021* Datensatz, um 24% reduziert.

Abstract

In this thesis, we focus on the NP-hard graph-based problem Weighted Cluster Editing. In Weighted Cluster Editing we deal with weighted graphs. An edge weight denotes the cost of either inserting or deleting an edge. Weighted Cluster Editing asks, whether it is possible to transform a graph into a cluster graph by inserting and/or deleting edges with total cost upper-bounded by a number k. A cluster graph is a graph where every connected component is a clique. In our work, we concentrate on a data reduction rule for Weighted Cluster Editing where only heuristic approaches for applying this rule were known before. We provide a full proof of correctness and investigate properties of graphs related to minimum cuts, which lead to a recursive polynomial-time algorithm for applying the data reduction rule exhaustively. We further optimise the algorithm by providing properties of node sets that speed up the running time in practise. Lastly, we present our implementation of the algorithm and compare it to the heuristic approaches. Based on the results, we analyse and discuss the strengths and weaknesses of the algorithm. The data reduction rule shows to be effective, reducing the real world instances of the *PACE Challenge 2021* dataset by 24% on average.

Contents

Chapter 1

Introduction

Clustering is the task of finding groups of related objects based on a similarity or distance measure. For instance, in biology, the analysis of amino acid sequences plays an essential role in understanding the functionality of proteins [Wit+07]. Known properties of a protein might also hold true for new unknown proteins with similar amino acid sequences. Similarly, the analysis of gene data requires the grouping of related gene expressions [BDSY99; SS00]. In both cases, graph-based data clustering can be used to find groups of related proteins or genes. The idea is to represent the problem as a graph, with a node representing, e.g. a single protein and an edge between nodes indicating that the similarity between two nodes (or proteins) exceeds a certain predefined threshold. Given such a similarity graph, the task of Cluster Editing, also known as Correlation Clustering [BBC04], is to find the smallest possible set of edges to insert or delete such that the resulting graph is a cluster graph. A cluster graph is a graph where each connected component forms a clique. Formally, the NP-hard problem [KM86] Cluster Editing can be described as follows:

Cluster Editing
Input: An undirected graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Is it possible to transform G into a union of disjoint cliques by edge deletions and/or edge insertions so that the amount of edge modifications is upper-bounded by k?

In this thesis, we focus on the weighted version of cluster editing, called Weighted Cluster Editing, which is a generalisation of Cluster Editing. While in Cluster Editing every edge insertion or deletion has a cost of precisely 1, the cost in the context of Weighted Cluster Editing differs depending on the similarity between two nodes. The weight of an edge denotes the similarity of the nodes incident to that edge, while the weight of a missing edge denotes the dissimilarity of two nodes. The weight of an edge or missing edge can also be seen as the strength of physical attraction. An edge with a large weight means that the incident nodes are attracted to one another, whereas a large weight for a missing edge means that they repel each other. The goal of Weighted Cluster Editing is to find a set of edges to insert or delete such that the resulting graph is a cluster graph and the sum of weights of these edges is minimal.

Figure 1.1 shows an example which we discuss subsequently. There, a thick line represents strong (dis-)similarity and a thin line stands for a weak (dis-)similarity be-

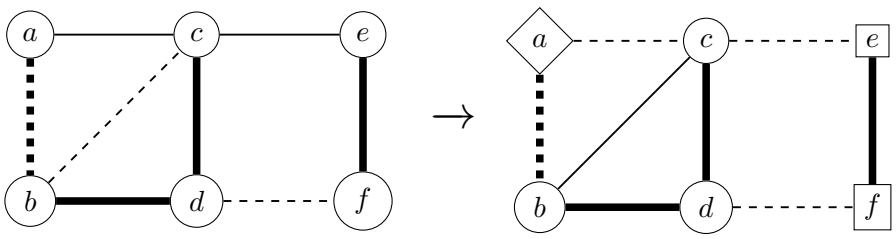

Figure 1.1: A weighted graph instance before (left) and after applying Weighted Cluster Editing (right). A dashed edge denotes that the incident nodes are dissimilar, and a solid edge shows that the incident nodes are similar. The thickness of a line indicates how strong the similarity or dissimilarity is and represents the value of the weight for that edge or missing edge. Note that, for simplicity reasons, we left out some edges which are negligible as their weights are small. By deleting the edges $\{a, c\}$ and $\{c, e\}$ and inserting the edge $\{b, c\}$, each connected component of the graph forms a clique.

tween two nodes. To give an intuitive feeling for Weighted Cluster Editing, we take a closer look at node a. Node a is similar to the node c and very unsimilar to b. To include a in a cluster, we either insert the edges $\{a, b\}$ and $\{b, c\}$ or we delete the edge $\{a, c\}$. As a and b are very unsimilar, the cost of inserting that edge is high. Hence, it is cheaper to make a its own cluster by deleting the edge $\{a, c\}$.

A Cluster Editing instance can be transformed to a Weighted Cluster Editing instance by assigning every edge and missing edge a weight of 1. Because of that, it is easy to see that also Weighted Cluster Editing is NP-hard.

Even when highly optimised, in practise, most of the exact algorithms for NP-hard problems are still too slow for larger instances. A powerful method of decreasing the running time is data reduction. The idea is to transform a problem instance into an equivalent but smaller problem instance in polynomial time called the kernel. In the best case, the resulting problem instance is trivial to solve.

But data reduction is not only beneficial for NP-hard problems. In this thesis, we make use of global minimum cuts, which are the minimum weight cuts that separate the graph into two connected components. The global minimum cut problem is solvable in polynomial time [NNI00], but the algorithm is still too slow in practise for large graphs. Henzinger et al. [Hen+20] apply different data reduction rules to first reduce the graph instance before using an algorithm that computes the exact result. By doing this, they can solve very large graphs with up to a billion edges within minutes.

1.1 Related Work

As mentioned above, a graph G with nodes V and edges E is a cluster graph if and only if every connected component forms a clique. This is the case if there are no three nodes $u, v, w \in V$ such that $\{u, v\} \in E \wedge \{v, w\} \in E \wedge \{u, w\} \notin E$, also called conflict triple or a P_3. The basic idea of most algorithms that solve Cluster Editing and Weighted Cluster Editing is to destroy each conflict triple $\{u, v, w\}$ by either deleting one of the edges $\{u, v\}$ or $\{v, w\}$ or inserting the missing edge $\{u, w\}$. A simple branch-and-bound algorithm, based on this idea of removing conflict triples, has a running time of $\mathcal{O}(3^k \cdot n^3)$

with $n := |V|$ and k being the size of the solution. Improvements of this idea, using clever branching techniques resulted in a $\mathcal{O}(2.27^k + n^3)$ algorithm by Gramm et al. [Gra+05] and the to this date fastest algorithm with a theoretical running time of $\mathcal{O}(1.62^k + m + n)$ by Böcker [Böc12] with $m := |E|$. Another approach, which is also based on conflict triples, is to formulate the problem as an integer linear program (ILP) instance [GW89], which can be solved using an ILP solver. According to Böcker, Briesemeister, and Klau [BBK08], this approach can solve large graphs in a reasonable amount of time. There also exist many algorithms that solve Cluster Editing heuristically [BDSY99; SS00; Wit+07].

Our work heavily relies on the work of Böcker, Briesemeister, and Klau [BBK08] from 2008. They present several data reduction rules for Weighted Cluster Editing. In this thesis, we focus on one specific data reduction rule, called *Almost Clique*, by Böcker, Briesemeister, and Klau [BBK08, Rule 4] where only a heuristic for applying the rule is presented.

Using *Almost Clique* at its core, Cao and Chen [CC12] construct a $2k$ node kernel for the integer-weighted version of the Cluster Editing problem. This means that after the kernelisation, the graph has at most $2k$ nodes. Additionally, they present a $4k$ node kernel for the real-weighted version of the problem.

For the unweighted version Gramm et al. [Gra+05] provided a k^2 node kernel. The to this day smallest node kernel is the $2k$ node kernel by Chen and Meng [CM12]. This $2k$ node kernel was tested empirically by Hartung and Hoos [HH15], who also presented a new data reduction rule, which outperforms the $2k$ node kernel in practise.

1.2 Our Results

In this thesis, we focus on one specific data reduction rule, called *Almost Clique*, presented by Böcker, Briesemeister, and Klau [BBK08, Rule 4]. We provide a full proof of correctness of the data reduction rule for Weighted Cluster Editing. Moreover, we provide a recursive algorithm that checks if the *Almost Clique* rule is applicable in $\mathcal{O}(n^4 \cdot (m + n \log(n)))$ time. We further present a set of conditions, which have the potential to improve the running time to $\mathcal{O}(n^2 \cdot (m + n \log(n)))$ for most graphs. We provide an implementation of the algorithm and test the algorithm on the *PACE Challenge 2021* [Kel+21] dataset. The algorithm reduces the number of nodes by around 24% for real-world graph instances and 2% for random graph instances, reducing the instances more than the heuristic approaches. Furthermore, 11% of the graph instances got solved completely. Lastly, we analyse the recursion behaviour of the algorithm and discuss the weaknesses of the algorithm that showed during testing. It shows that for most graphs, the recursion depth gets high, resulting in many expensive minimum cut calculations and a high running time in practise. Despite the high running time in practise, the algorithm can be of use as a preprocessing step, as it reduces the number of nodes significantly on average.

Chapter 2

Preliminaries

Graph theory. Let $G = (V, E)$ be an undirected graph, where V is the set of vertices and $E \subseteq \binom{V}{2}$ a set of edges with $n := |V|$ and $m := |E|$. For a pair of nodes $\{u, v\} \in \binom{V}{2}$ we write uv.

We denote by

$N_G(v)$ the (open) *neighbourhood* of v, formally, $N_G(v) := \{u \in V; uv \in E(G)\}$;

$N_G(V')$ the *(open) neighbourhood* of V', formally, $N_G(V') := \left(\bigcup_{u \in V'} N_G(u)\right) \setminus V'$ for $V' \subseteq V$;

$N_G[v]$ the *closed neighbourhood* of v, formally, $N_G[v] := N_G(v) \cup \{v\}$;

$N_G[V']$ the *closed neighbourhood* of V', formally, $N_G[V'] := N_G(V') \cup V'$;

$G[V']$ the *induced subgraph* of G on $V' \subseteq V$, formally, $G[V'] := (V', E(G) \cap \binom{V'}{2})$;

$G \triangle W$ the graph obtained from G by removing every edge $e \in W$ with $e \in E$ and inserting every edge $e \in W$ with $e \notin E$;

$\bar{S}$ for $S \subseteq V$ the complement is $\bar{S} = V \setminus S$;

$\max(A, B)$ let A, B be two node sets. The function $\max(A, B)$ returns the bigger set of the two. Formally, it returns A if $|A| \geq |B|$, and B if $|A| < |B|$.

2.1 Data Reduction

As mentioned in the introduction, the main idea of data reduction is to transform an instance for a problem into a smaller instance in polynomial time. We use the following definition by Cygan et al. [Cyg+15]: "A data reduction rule [...] for a parameterised problem Q is a function $\phi : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}$ that maps an instance (I, k) of Q to an equivalent instance (I', k') of Q such that ϕ is computable in polynomial time in $|I|$ and k. We say that two instances of Q are equivalent of $(I, k) \in Q$ if and only if $(I', k') \in Q$. [We further require] that there exists a computable function $g(\cdot)$ such that whenever (I', k') is the output for an instance (I, k), then it holds that $|I'| + k' \leq g(k)$." If a data reduction rule fulfils the conditions above, we call it correct.

2.2 Weighted Cluster Editing

In our work, we use the following graph model: Let $G = (V, E, s)$ denote an undirected weighted graph, where V is the set of vertices and s a weight function $s : \binom{V}{2} \to \mathbb{Z}$. The weight function s assigns every pair of nodes $uv \in \binom{V}{2}$ an integer weight. The weight function encodes the edges of the graph in the following way: If $s(uv) > 0$, then we call uv an edge, while we call uv a non-edge, if $s(uv) \leq 0$. A pair uv with edge weight exactly zero, we call a zero-edge and treat it as a missing edge. We define $E := \{uv \mid u, v \in V, u \neq v, s(uv) > 0\}$ as the set of edges.

We use the following definition of Weighted Cluster Editing and the notation by Böcker et al. [Bö+09]:

Weighted Cluster Editing
Input: An undirected graph $G = (V, E, s)$ with nodes V, an integer $k \in \mathbb{N}$ and a weight function $s : \binom{V}{2} \to \mathbb{Z}$.
Question: Is it possible to transform G into a union of disjoint cliques by edge deletions and/or edge insertions such that the sum of absolute weight values of the inserted and deleted edges is upper-bounded by k?

The absolute value $|s(uv)|$ defines the cost that arise, when inserting or deleting uv. A special case are the zero edges. We treat zero-edges as missing edges, which can be inserted without any additional cost. We call a set of node pairs $W \subseteq \binom{V}{2}$ a solution for Weighted Cluster Editing or WCE, if $G \triangle W$ is a cluster graph, which is the case if every connected component is a clique. The *cost* of a solution is $s(W) := \sum_{uv \in W} |s(uv)|$. In the context of Cluster Editing and Weighted Cluster Editing we can assume that G is connected, as we show in Lemma 2.1.

Lemma 2.1. *Let $G = (V, E, s)$ be a graph. There exists an optimal solution for the* Weighted Cluster Editing *problem, which does not insert edges between the connected components in G.*

Proof. Let $G[X]$ and $G[Y]$, with $X, Y \subset V$ be two connected components in G. Let us suppose there is an optimal solution for Weighted Cluster Editing with solution set W, which inserts edges between $G[X]$ and $G[Y]$. The cost of inserting the edges $s(W \cap (X \times Y))$ is at least 0, as we have to pay at least 0 to insert the edges between the nodes in X and Y. By definition, $G[X] \triangle W$ and $G[Y] \triangle W$ are cluster graphs. Therefore $W' = W \setminus (X \times Y)$ is a weighted cluster editing set with $s(W') \leq s(W)$. □

In this thesis, we make use of different properties of graph cuts. We use the following notation based on the work by Henzinger et al. [Hen+20].

Definition 2.2. Let $G = (V, E, s)$ be a graph and $S \subseteq V$. A cut c_S of $G[S]$ with $c_S = (A, S \setminus A) = (A, \bar{A}), A \subset S$ is a partitioning of the node set S into two non-empty partitions A and $S \setminus A$, each being called a *side* of the cut.

For a cut $c_S = (A, \bar{A})$ of $G[S]$, we define the following notation:

$E(c_S)$ are the edges between A and $\bar{A}$, formally $E(c_S) := (A \times \bar{A}) \cap E$;

$N(c_S)$ are the nodes incident to $E(c_S)$;

$\text{cost}(c_S)$ cost of removing the edges $E(c_S)$, formally, $\text{cost}(c_S) := \sum_{uv \in E(c_S)} s(uv)$;

Definition 2.3. Let $G = (V, E, s)$ and $S \subseteq V$. The minimum cut (*mincut*) is the cut of $G[S]$, with minimum cost $s(E(c_S))$ for all cuts c_S in $G[S]$. The cost of that cut is defined as $\text{mincutcost}_G(S)$.

The cut that separates a set $S \subseteq V$ from the rest of the graph $V \setminus S$ plays an essential role in the data reduction rule we present in the next chapter. Therefore we explicitly define the cost of that cut by $\text{cut}_G(S)$.

Definition 2.4. Let $G = (V, E, s)$ be a graph. The cut $c = (S, V \setminus S)$ that separates S from the rest of the graph $\bar{S}$ has cost:

$$\text{cut}_G(S) := \text{cost}(c)$$

Additionally, we define the cut cost between two disjoint node sets $A \subset V$ and $B \subset V$ by $\text{cut}_G(A, B)$.

Definition 2.5. Let $A \subset V$ and $B \subset V$ with $A \cap B = \emptyset$. We define $\text{cut}_G(A, B)$ as the cost to cut the edges between A and B.

$$\text{cut}_G(A, B) := \sum_{uw \in (A \times B) \cap E} s(uw)$$

Lastly, we introduce the deficiency of a node set $S \subseteq V$, which is the cost to insert all non-edges between nodes in S.

Definition 2.6. The deficiency of S of graph G are the costs of adding edges to transform $G[S]$ into a cluster graph.

$$\text{def}_G(S) := \sum_{u,w \in S, u \neq w, uw \notin E} |s(uw)|$$

Chapter 3

Almost Clique

There are many different polynomial-time data reduction rules for Weighted Cluster Editing [BBK08; CC12]. In this chapter, we focus on one specific data reduction rule, called *Almost Clique*, presented by Böcker, Briesemeister, and Klau [BBK08, Rule 4]. The rule looks at node sets $S \subseteq V$, where $G[S]$ almost forms a clique, and the nodes in S are loosely connected to $N_G(S)$. The main idea of this rule is that it is cheaper to convert S into its own cluster rather than splitting up S into multiple clusters. The rule calculates the minimum cut of $G[S]$ and checks whether it is more expensive than cutting off the edges from S to $N_G(S)$ and inserting the missing edges to make $G[S]$ a clique. An example can be seen in Figure 3.1.

We first introduce the merge operation, which is used to shrink the graph instance when the data reduction rule applies. Afterwards, we give a full proof of correctness of *Almost Clique*.

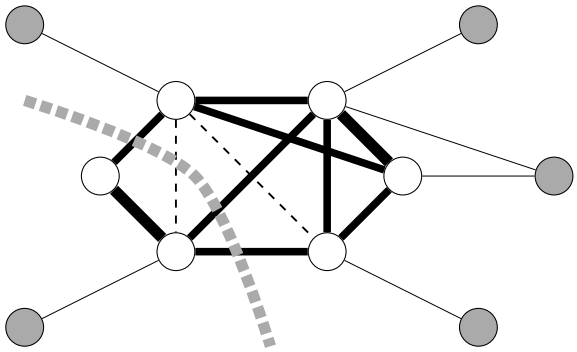

Figure 3.1: A graph $G = (V, E, s)$ with a set of nodes $S \subset V$. The white nodes represent the nodes in S, while the grey nodes represent the neighbourhood $N_G(S)$. A dashed line denotes a non-edge, and the thickness of the line is the absolute value of the weight of that (non-)edge. The grey dotted line shows the minimum cut of $G[S]$. The nodes S are tightly connected with only two missing edges with a small insertion cost. Additionally, the nodes in S are loosely connected to the neighbourhood of S. The example is based on Böcker, Briesemeister, and Klau [BBK08, Figure 2].

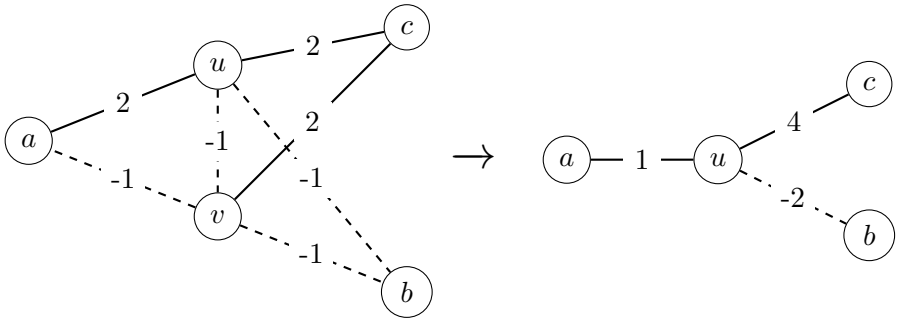

Figure 3.2: A graph G before merging the nodes u and v (left) and the graph G' after applying the merge operation (right). Note that the graph does not show the edges between a, b, and c, as the merge operation does not affect them.

3.1 Merge Operation

Given a graph $G = (V, E, s)$, we can treat two nodes $u, v \in V$ as a single node when we know they end up in the same cluster. To do so, we create a new graph G' based on G. We first remove the node v from G and every edge incident to v and treat u as the merged node of u and v. We then define new edge weights $s'(uw) = s(uw) + s(vw)$ for every $w \in (V \setminus \{u, v\})$. We call this operation the merging of u and v.

Definition 3.1. Let $G = (V, E, s)$ be a graph and $u, v \in V$. The graph G' obtained by merging u and v is $G' := ((V \setminus \{v\}), E', s')$ with

$$s'(xw) = \begin{cases} s(uw) + s(vw), & x = u \\ s(xw), & else \end{cases}$$

and $E' = \{xy \in \binom{V}{2} \mid s'(xy) > 0\}$

Figure 3.2 illustrates how the merge operation of two nodes u and v works and which fixed costs arise when performing it.

First, we need to account for the non-edge uv. As u and v end up in the same cluster, we need to insert the edge uv with cost $\mathrm{def}(uv) = 1$. The example illustrates the three possible cases when merging two nodes. The node c has an edge to u and v, whereas b has two non-edges to u and v. In the merged graph G', we sum up the edge weights without any fixed costs. The reason for this is that removing uc in G' represents the removal of uc and vc in G. Similarly, the insertion of ub in G' represents the insertion of edges ub and vb in G. Herefore, in the Graph G' resulting after merging u and v, we have $s'(uc) = 4$ and $s'(ub) = -2$.

Lastly, we need to deal with the node a. It has one edge ua and a non-edge va. When summing up the edge weights as specified, the edge ua in G' has weight 1. No matter if a ends up in the same cluster as u and v or does not, we have to pay at least 1 to either insert the edge va or pay part of the cost of deleting ua. In this case, the fixed cost are $\min(|s(ua)|, |s(va)|)$, to either include a in or exclude a from a cluster with u and v. We therefore increase the fixed cost by 1 and set $s'(ua) = 1$.

After merging u and v, the resulting graph G' shrank by one node, and we know part of the cost of the optimal solution for G. The merge cost are 2, for accounting for the deficiency of u and v and the fixed cost for a.

In general, for a node $w \in V \setminus (\{u, v\})$, if $\text{sign}(s(uw)) = \text{sign}(s(vw))$, then it is easy to see that inserting or removing the edge uw in G', costs the same as inserting or removing both edges uw and vw in G. Otherwise, if $\text{sign}(s(uw)) \neq \text{sign}(s(vw))$, then we know that we have fixed cost of $\min(|s(uw)|, |s(vw)|)$, since we either have to delete an edge or insert an edge to include or exclude w in a cluster with u and v.

Definition 3.2. Let $G = (V, E, s)$ be a graph and $u, v \in V$. The fixed cost that occurs when defining the new weights during the merge operation is

$$\text{fixedcost}_G(uv) := \sum_{w \in (V \setminus \{u,v\}), \text{sign}(s(uw)) \neq \text{sign}(s(vw))} \min(|s(uw)|, |s(vw)|).$$

If uv is a non-edge, then we have further costs for inserting the edge between u and v and need to increase the merge cost by $-s(uv) = \text{def}_G(uv)$.

Definition 3.3. Let $G = (V, E, s)$ be a graph and $u, v \in V$. The total merge cost are

$$\text{mergecost}(uv) := \text{def}_G(uv) + \text{fixedcost}_G(uv).$$

Coming back to the example shown in Figure 3.2, lets suppose the graph shown has an optimal solution with cost k, then in the merged graph G' the optimal solution for Weighted Cluster Editing has cost $k' = k - \text{mergecost}_G(uv) = k - 2$.

Similarly to Definition 3.1, we can merge a set of nodes $S \subseteq V$ if we know that all nodes in S end up in the same cluster. We achieve that by first inserting all non-edges between nodes in S, that is $\text{def}_G(S)$. Then, we pick a node $u \in S$ and iteratively merge u and $v \in S \setminus \{u\}$ until just the node u, also called the merged node or q, is left. The merged node represents the node set S in G'.

Observation 3.4. *Let $G = (V, E, s)$ be a graph and $S \subseteq V$. The cost $\text{fixedcost}_G(S)$ is the cost that arises when iteratively merging two nodes $u, v \in S$, after inserting the non-edges between nodes in S:*

$$\text{fixedcost}_G(S) := \sum_{w \in V \setminus S} \min\Big(\sum_{u \in S \setminus \{w\}, s(uw) > 0} s(uw), \sum_{u \in S \setminus \{w\}, s(uw) \leq 0} -s(uw) \Big)$$

Overall the cost for merging the set of nodes S in arbitrary order are

$$\text{mergecost}(S) := \text{def}_G(S) + \text{fixedcost}_G(S).$$

Proof. We first have to insert all non-edges between nodes in S, with a cost of $\text{def}_G(S)$. We then iteratively merge two nodes in S until just one node is left. For a node $w \in V \setminus S$ whenever we merge two nodes $u, v \in S$, where without loss of generality uw is an edge and vw is a non-edge, we have to pay fixed cost of $\min(s(uw), -s(vw))$. Note that the sum of these costs for w is independent of the order in which we merge the nodes in S. Hence, to calculate the cost, we can think of it the following way: First, we merge all

nodes in S that have edges to w and then merge all nodes that have non-edges to w. Since we only merge nodes that either have all edges or non-edges to w, no further costs arise. Afterwards, we are left with two nodes $x, y \in S$, with an edge xw and a non-edge yw. If we now merge those two nodes, we have to pay fixed cost of

$$\min(s(xw), -s(yw)) = \min(\sum_{u \in S \setminus \{w\}, s(uw) > 0} s(uw), \sum_{u \in S \setminus \{w\}, s(uw) \leq 0} -s(uw)).$$

The same goes for the other nodes in $V \setminus S$. Thus we sum up the cost over all $w \in V \setminus S$. □

3.2 Almost Clique Rule

In this section, we prove the correctness of the *Almost Clique* data reduction rule presented by Böcker, Briesemeister, and Klau [BBK08, Rule 4].

Definition 3.5. Let $S \subseteq V$ be a node set of $G = (V, E, s)$ with $|S| \geq 2$. A set S satisfies the *merge condition* if

$$\text{mincutcost}_G(S) \geq \text{def}_G(S) + \text{cut}_G(S). \tag{3.1}$$

Reduction Rule 3.1. *Let $S \subseteq V$ be a node set of $G = (V, E, s)$. If S satisfies the merge condition (3.1), then merge all nodes in S into a single node and increase the cost by* $\text{mergecost}_G(S)$.

Lemma 3.6. *Reduction Rule 3.1 is correct.*

Proof. Let $G = (V, E, s)$ be a graph with its weight function s and $G' = (V', E', s')$ be the graph after applying Reduction Rule 3.1 on G, with s' as its weight function. Let $S \subseteq V$ be a node set that satisfies the merge condition (3.1), that is, $\text{mincutcost}_G(S) \geq \text{def}_G(S) + \text{cut}_G(S)$. Let k be the solution size and $k' = k - \text{mergecost}_G(S)$. We show that (G, k) is a yes-instance $\iff$ (G', k') is a yes-instance.

(G, k) is a yes-instance $\Rightarrow$ (G', k') is a yes-instance:

Let $W \subseteq \binom{V}{2}$ be a weighted cluster editing set (WCE) for G of cost k and $C = G \triangle W$ be the cluster graph obtained from W with edges E_C. We show that, as the reduction condition applies for S, it is cheaper to transform $G[S]$ into a cluster than to split $G[S]$ into two separate components. Let q be the merged node after merging the nodes in S.

Case a: All nodes of S are in the same cluster in C. Let $x \in S$. We construct a new WCE W' for G':

$$\begin{aligned} W' := W \setminus (S \times V) \\ \cup \{qw \mid w \in \bar{S}, xw \in E_C, qw \notin E'\}\} \\ \cup \{qw \mid w \in \bar{S}, xw \notin E_C, qw \in E'\}\}. \end{aligned}$$

We first show that W' is a WCE for G'. We start with W, which is a WCE for G. We remove all incident edges to the nodes in S, as they no longer exist in G'. We then insert edges from q to the rest of the cluster of S in G and disconnect q from every other

node. Therefore q only is connected to the cluster in which S previously was in C, and every other cluster is unaffected by the merge rule. Because of that, $G' \triangle W'$ is a cluster graph.

From Observation 3.4 we know that after merging a set $S \subseteq V$ in G, we have a cost of $\text{mergecost}_G(S)$, which we do not have to pay in G'. Every other edge that is not incident to S is unaffected by the merge operation. Thus $k' = k - \text{mergecost}_G(S)$.

Case b: The nodes of S are in different clusters in C. As the nodes in S are in different clusters, $G[S]$ is split into at least two components. We will see that it is cheaper to form a new cluster consisting of the nodes in S instead of splitting up the nodes in S. We construct a new WCE $\hat{W}$ for G with

$$\begin{aligned}\hat{W} := W &\setminus (S \times V) \\ &\cup \left(\binom{S}{2} \setminus E\right) \\ &\cup ((S \times \bar{S}) \cap E).\end{aligned}$$

The set $\hat{W}$ is a WCE for G, as we first separate S from the rest of the graph and then add all missing edges between nodes in S, including those that were removed before, with cost larger than $\text{mincutcost}_G(S)$. The cost of $\hat{W}$ is

$$c := k + \text{def}_G(S) + \text{cut}_G(S) - \sum_{uv \in ((S \times V) \cap W)} |s(uv)|.$$

We know that $c \leq k$ because

$$\sum_{uv \in ((S \times V) \cap W)} |s(uw)| \geq \text{mincutcost}_G(S) \geq \text{def}_G(S) + \text{cut}_G(S).$$

Afterwards we can apply *Case a*, with $\hat{W}$ as the WCE, which gives us WCE for G' of size $k' = c - \text{mergecost}_G(S) \leq k - \text{mergecost}_G(S)$, since $c \leq k$.

(G, k) is a yes-instance $\Leftarrow$ (G', k') is a yes-instance:

Let $W' \subseteq \binom{V'}{2}$ be a WCE of G' of cost k' and $C' = G' \triangle W'$ the resulting cluster graph with edges $E_{C'}$. We create a weighted cluster editing set W for G:

$$\begin{aligned}W := W' &\setminus \{qw \mid w \in (V' \setminus q)\} \\ &\cup \{sw \mid w \in (V' \setminus q), s \in S, qw \in E_{C'}, sw \notin E\} \\ &\cup \{sw \mid w \in (V' \setminus q), s \in S, qw \notin E_{C'}, sw \in E\} \\ &\cup \{sw \mid s, w \in S, s \neq w, sw \notin E'\}.\end{aligned}$$

First, we remove all incident edges to q, as q does not exist in G. We then insert all edges from S to the former cluster of q and remove all edges to the other nodes in G. Lastly, we add all missing edges between nodes in S. Thus, the nodes in S are in the same cluster as q was, and as the rest of the graph is not affected, $G \triangle W$ is a cluster graph. We show that the cost of W are $k = k' + \text{mergecost}(S)$. First, we need to add back $\text{fixedcost}_G(S)$ which arise when defining the new edge weights for G' as described in Observation 3.4. Lastly we need to account for the deficiency $\text{def}_G(S)$. Again, every other edge is unaffected by the merge operation, thus $k = k' + \text{fixedcost}_G(S) + \text{def}_G(S) = k' + \text{mergecost}(S)$. □

Summary

In this chapter, we introduced the *Almost Clique* rule by Böcker, Briesemeister, and Klau [BBK08] and explained how the merge operation works. We further provided a full proof of correctness of *Almost Clique*. The question arises how a node set $S \subseteq V$ that satisfies the merge condition (3.1) can be found. We answer that question in the following chapter.

Chapter 4

Polynomial-time Algorithm for Almost Clique

To apply Reduction Rule 3.1 (*Almost Clique*) presented in the previous section, we need to find a set of nodes, which satisfies the merge condition (3.1). Searching through every possible set of nodes and testing them individually results in an exponential running time. Böcker, Briesemeister, and Klau [BBK08] provide a heuristic for finding such sets. We investigate how effective the heuristic is in Chapter 5. Cao and Chen [CC12] present a data reduction rule, which is also based on edge cuts. They use the closed neighbourhood $N_G[u]$ for a node $u \in V$ as node sets to test their data reduction condition. For this data reduction, unfortunately, this heuristic approach does not work when there is a set $S \subseteq V$ which satisfies the merge condition (3.1) and there is no node $u \in S$ incident to all other nodes in S. A counterexample can be seen in Figure 4.1.

In this chapter, we present a polynomial-time algorithm for applying Reduction Rule 3.1. More precisely, for a graph $G = (V, E, s)$, the algorithm finds the biggest set $S \subseteq V$ that satisfies the merge condition, that is $\text{mincutcost}_G(S) \geq \text{cut}_G(S) + \text{def}_G(S)$ if such a set exists.

The algorithm searches through possible candidate sets using a divide & conquer

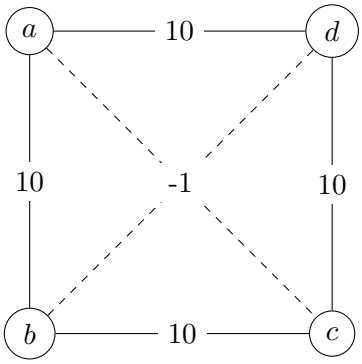

Figure 4.1: Counterexample of a graph that cannot be reduced with Reduction Rule 3.1 when using the closed neighbourhood of a single node as the node set S, although there is a set $S = \{a, b, c, d\}$ that satisfies the merge condition (3.1).

approach. The main idea is that calculating a mincut for $G[S]$ with $S \subseteq V$ provides a lower bound for $\text{cut}_G(S')$ for $S' \subset S$, which allows us to reduce the amount of node sets to check. To show the polynomial running time of the algorithm, we present the following small lemmata.

Lemma 4.1. *Let $G = (V, E, s)$ be a graph and $S \subseteq V$. For every $S' \subset S$, it holds that $\text{cut}_G(S') \geq \text{mincutcost}_G(S)$.*

Proof. Since $|S'| < |S|$, we have to cut edges to separate S' from $S \setminus S'$. To cut these edges, we have to pay at least $\text{mincutcost}_G(S)$, as otherwise, this cut would be the mincut of S. □

In addition to the lower bound for $\text{cut}_G(S')$ with $S' \subset S$, there exists an upper bound for $\text{mincutcost}_G(S')$ in $G[S']$ if S' contains nodes on both sides of a given mincut $c_S = (A, \bar{A})$, i.e. $S' \cap A \neq \emptyset$ and $S' \cap \bar{A} \neq \emptyset$.

Lemma 4.2. *Let $G = (V, E, s)$ be a graph and $S \subseteq V$. Let $c_S = (A, \bar{A})$ be the mincut of $G[S]$ and $S' \subset S$ with $S' \cap A \neq \emptyset$ and $S' \cap \bar{A} \neq \emptyset$. Then, $\text{mincutcost}_G(S') \leq \text{mincutcost}_G(S)$.*

Proof. As S' has nodes on both sides of the mincut, $G[S']$ shares some or all edges of $E[c_S]$, therefore S' has a cut $W = E[c_S] \cap \binom{S'}{2}$ with cost of at most $\text{mincutcost}_G(S)$. Thus $\text{mincutcost}_G(S') \leq \text{mincutcost}_G(S)$. □

With the two lemmata Lemma 4.1 and Lemma 4.2 we can derive the following strict conditions a subset $S' \subset S$ has to fulfil in order to satisfy the merge condition (3.1).

Lemma 4.3. *Let $G = (V, E, s)$ and $S \subseteq V$. Let $c_S = (A, \bar{A})$ be a mincut of $G[S]$. A set $S' \subset S$ with $S' \cap A \neq \emptyset$ and $S' \cap \bar{A} \neq \emptyset$ can only satisfy the merge condition (3.1) if $\text{cut}_G(S') = \text{mincutcost}_G(S') = \text{mincutcost}_G(S)$ and $\text{def}_G(S') = 0$.*

Proof. From Lemma 4.1 and Lemma 4.2 we know that for every $S' \subset S$ with nodes on both sides of the mincut, $\text{mincutcost}_G(S') \leq \text{mincutcost}_G(S) \leq \text{cut}_G(S')$. Therefore, S' can only satisfy the merge condition (3.1) if $\text{def}_G(S) = 0$ and $\text{mincutcost}_G(S') = \text{mincutcost}_G(S) = \text{cut}_G(S')$. □

Lemma 4.4. *Let $G = (V, E, s)$ be a graph and $S \subseteq V$. Let $c_S = (A, \bar{A})$ be the mincut of S and $S' \subset S$ with $S' \cap A \neq \emptyset$ and $S' \cap \bar{A} \neq \emptyset$. If S' satisfies the merge condition (3.1), then S' is equal to one of the sides of another mincut $c'_S \neq c_S$ of $G[S]$.*

Proof. From Lemma 4.3 we know that the only potential candidate sets S' are those with $\text{cut}_G(S') = \text{mincutcost}_G(S)$. The only cuts with such cost are the mincuts. Therefore, if such a set S', having nodes on both sides of the mincut c_S, exists, then S' must be equal to one side of one of the other mincuts $c'_S \neq c_S$ in $G[S]$. □

From Lemma 4.4, we can construct a recursive algorithm that finds the biggest node set $S \subseteq V$ satisfying the merge condition (3.1). We reduce the number of possible candidate sets in each recursion by calculating an arbitrary mincut and checking if there is a set that satisfies the merge condition (3.1) with nodes on both sides of the mincut.

If there is no such set, then the only possible candidates are on one of the sides of the mincut, which we can check separately in the same manner.

Algorithm 1 AlmostClique

Input: $G = (V, E, s)$
Output: Biggest node set $S \subseteq V$, with $\text{mincutcost}_G(S) \geq \text{cut}_G(S) + \text{def}_G(S)$
function AlmostClique(G)
return BiggestMincutSet(G, V)
end function

Algorithm 2 BiggestMincutSet

Input: $G = (V, E, s)$, $S \subseteq V$
Output: Biggest node set $S' \subseteq S$, with $\text{mincutcost}_G(S') \geq \text{cut}_G(S') + \text{def}_G(S')$

```
1:  function BiggestMincutSet(G, S)
2:      if |S| < 2 then return ∅
3:      if S satisfies the merge condition (3.1) then return S
4:
5:      c_S = arbitrary mincut in G[S]
6:
7:      S' = ∅
8:      for every other mincut c'_S in G[S] do
9:          if a side X of c'_S satisfies the merge condition (3.1) then
10:             S' = max(S', X)
11:         end if
12:     end for
13:
14:     (A, Ā) = c_S
15:     S' = max(S', BiggestMincutSet(G, A))
16:     S' = max(S', BiggestMincutSet(G, Ā))
17:
18:     return S'
19: end function
```

4.1 Proof of Correctness

In this section, we prove the correctness of Algorithm 1 based on Lemma 4.4 and show that Algorithm 1 always returns the biggest set $S \subseteq V$ for a graph $G = (V, E, s)$ that satisfies the merge condition (3.1).

Lemma 4.5. *Let $G = (V, E, s)$. Algorithm 1 returns a set $S \subseteq V$ satisfying the merge condition (3.1), if such a set exists in G.*

Proof. It is easy to see that if Algorithm 2 returns a set S with $S \neq \emptyset$, then S satisfies the merge condition (3.1).

Algorithm 1 calls Algorithm 2 with G and sets $S = V$. Thus we need to show that Algorithm 2 finds a set $S' \subseteq S$ satisfying the merge condition (3.1), if it exists. If $|S| < 2$, then S cannot satisfy the merge condition (3.1) in Line 2. If $|S| \geq 2$, then S is a potential candidate set. Hence, we check whether S satisfies the merge condition (3.1) in Line 3. If S satisfies the merge condition (3.1), then we are done and return S.

Otherwise, we know that the only potential candidate sets are proper subsets of S. We now make use of Lemma 4.4. We calculate an arbitrary mincut in the induced subgraph $G[S]$, called $c_S = (A, \bar{A})$. From Lemma 4.4 we know that, if a set $S' \subset S$ with nodes on both sides of c_S, i.e. $S' \cap A \neq \emptyset$ and $S' \cap \bar{A} \neq \emptyset$, exists, then S' is equal to one of the sides of the other mincuts of $G[S]$. Consequently, we can check each side of each mincut if it satisfies the merge condition (3.1), starting from Line 8. If there is no such set S', then we know that the only sets that can satisfy the merge condition (3.1) are subsets of one of the *sides* of the mincut $c_S = (A, \bar{A})$. Therefore we call Algorithm 2 recursively with A and $\bar{A}$. The correctness of the recursion follows by induction. □

Lemma 4.6. *Let $G = (V, E, s)$ be a graph. If Algorithm 1 returns a non-empty set $S \subseteq V$, then S is the biggest set, satisfying the merge condition (3.1) in G.*

Proof. Suppose that there is a bigger set $X \subseteq V$ that satisfies the merge condition (3.1). Let $c_V = (A, \bar{A})$ be a mincut of G. Then, X either has nodes on both sides of c_V, i.e. $X \cap A \neq \emptyset$ and $X \cap \bar{A} \neq \emptyset$ or is a subset of one of the sides, A or $\bar{A}$. If X has nodes on both sides of c_V, then Lemma 4.4 holds for X and Algorithm 2 would find and return X in Line 10.

Otherwise X is a subset of A or $\bar{A}$. We check each side recursively and potentially get two sets from each side, satisfying the merge condition (3.1). We use the max operation in Line 15 and Line 16. Thus, if X is returned by one of the recursive calls, then the algorithm returns X. □

4.2 Running Time Analysis

We now show that Algorithm 1 has a polynomial running time. This is because only a polynomial amount of global minimum cuts exists, which can be found in polynomial time [NNI00].

Lemma 4.7. *Let $G = (V, E, s)$. Algorithm 1 returns a non-empty set $S \subseteq V$, that satisfies the merge condition (3.1) in G, if such a set S exists, in $\mathcal{O}(n^4 \cdot (m + n \log(n)))$.*

Proof. Let $G = (V, E, s)$. We first analyse the running time of testing, whether a set $S \subseteq V$ satisfies the merge condition (3.1). To calculate the cut cost $\text{cut}_G(S)$, we need to iterate over every edge between S and $V \setminus S$ and sum up the positive edge weights, resulting in a running time of $\mathcal{O}(n^2)$. Similarly, to calculate $\text{def}_G(S)$, we need to sum up the weight of every non-edge in $G[S]$, which results in a running time of $\mathcal{O}(n^2)$. The running time of calculating the mincut of $G[S]$ is $\mathcal{O}(nm + n^2 \log(n))$, as shown in [SW97]. Therefore, the overall running time, to check if a set $S \subset V$ satisfies the merge condition (3.1), in Line 3 and Line 9, is $\mathcal{O}(nm + n^2 \log(n))$.

There exist up to $\binom{n}{2} = \frac{n(n-1)}{2}$ possible mincuts in G [Kar00], as we assume G is connected as shown in Lemma 2.1. Therefore checking every mincut in Line 8 has a running time of $\mathcal{O}(n^3m + n^4 \log(n))$.

But first, we need to calculate all possible mincuts, which is possible using a compact representation of all mincuts, called a cactus graph with $n^* = \mathcal{O}(n)$ nodes. It is possible to calculate the cactus representation and all mincuts in $\mathcal{O}(nm + n^2 \log(n) + n^*m \log(n))$ time [NNI00]. That is why the loop starting in Line 8 has a total running time of $\mathcal{O}(n^3m + n^4 \log(n) + n^*m \log(n)) = \mathcal{O}(n^3m + n^4 \log(n))$.

In Line 15 and Line 16, we check each side of the chosen mincut recursively. In the worst-case scenario, one side of the minimum cut is a single node. Then, the algorithm has a recursion depth of n. This results in a total running time of $\mathcal{O}(n^4 \cdot (m + n \log(n)))$, which is polynomial in n. □

The two lemmata presented above result in the following theorem.

Theorem 4.8. *Reduction Rule 3.1 can be applied to a graph instance in* $\mathcal{O}(n^4 \cdot (m + n \log(n)))$ *time .*

Proof. Follows from Lemma 4.5 and Lemma 4.7. □

We now know that it is possible to apply *Almost Clique* in polynomial time. However, for large graph instances, a theoretical running time of $\mathcal{O}(n^4 \cdot (m + n \log(n)))$ likely is too slow in practise. Therefore, in the following section, we present specific conditions to speed up the running time in practise.

4.3 Improvements for Practical Running Time

The high running time of the algorithm is primarily due to the calculation and loop over all minimum cuts starting in Line 8. However, if we know that no sets with nodes on both sides of the chosen minimum cut c_S exist, then we could skip the minimum cut loop in Line 8. In the following section, we describe different properties of node sets, which are not as expensive to check as calculating all minimum cuts, and are required for these sets to satisfy the merge condition (3.1).

Observation 4.9. *Let* $G = (V, E, s)$ *and* $S \subseteq V$*. Let* $c_S = (A, \bar{A})$ *be the mincut of* S*. Let* $S' \subset S$ *be a set with* $S' \cap A \neq \emptyset$ *and* $S' \cap \bar{A} \neq \emptyset$*. If* S' *satisfies the merge condition (3.1), then all nodes incident to the mincut, i.e.* $N(c_S)$*, are part of* S'*.*

Proof. If not all nodes incident to the edges of c_S are part of S', then there exists a mincut c'_S with $E(c'_S) = E(c_S) \cap \binom{S'}{2}$ and $|E(c'_S)| < |E(c_S)|$. Therefore, $\text{cost}(c'_S) < \text{cost}(c_S) = \text{mincutcost}_G(S)$, since $\forall e \in E(c_S) : s(e) > 0$, which contradicts Lemma 4.3. □

Because of Observation 4.9 and Lemma 4.3 we know that all nodes incident to the mincut need to be part of such a S' and need to have no deficiency, i.e. $\text{def}_G(N(c_S)) = 0$, which can be checked in $\mathcal{O}(n^2)$ time.

Observation 4.10. *Let $G = (V, E, s)$ and $S \subseteq V$. Let $c_S = (A, \bar{A})$ be the mincut of S. Let $S' \subset S$ be a set with $S' \cap A \neq \emptyset$ and $S' \cap \bar{A} \neq \emptyset$. If S' satisfies the merge condition (3.1), then $A \subset S'$ or $\bar{A} \subset S'$.*

Proof. Suppose that there is such a set S' that satisfies the merge condition (3.1), but with $S' \cap A \neq A$ and $S' \cap \bar{A} \neq \bar{A}$. As S' has nodes on both sides of the mincut c_S, Lemma 4.3 applies. Therefore $\text{cut}_G(S') = \text{mincutcost}_G(S)$. From Observation 4.9 we know that S' includes all nodes incident to the minimum cut c_S.

The cut denoted by $\text{cut}_G(S')$ consists of two separate cuts, one cutting of S' from nodes in A and the other one cutting of S' from nodes in $\bar{A}$.

Let us call the cut cost of these cuts c_1 and c_2. We know that $c_1 + c_2 = \text{cut}_G(S')$ and $c_i \neq 0$, therefore $c_i < \text{cut}_G(S') = \text{mincutcost}_G(S)$ for $i \in \{1, 2\}$, which contradicts the assumption that c_S is a mincut of G. □

To further restrict the possible sets S' with nodes on both sides of the chosen mincut $c_S = (A, \bar{A})$, we now only need to check those, where one side, without loss of generality A, is a subset of S', i.e. $A \subset S'$ and $\text{def}_G(A \cup N(c_S)) = 0$.

Observation 4.11. *Let $G = (V, E, s)$ and $S \subseteq V$. Let $c_S = (A, \bar{A})$ be the mincut of S. Let $S' \subset S$ be a set with $S' \cap A \neq \emptyset$ and $S' \cap \bar{A} \neq \emptyset$. If S' satisfies the merge condition (3.1), then $\text{cut}_G(S', V \setminus S) = 0$.*

Proof. The cut cost of S' is $\text{cut}_G(S') = \text{cut}_{G[S]}(S') + \text{cut}_G(S', V \setminus S)$. From Lemma 4.3 we know that such a set S' only exists, if $\text{cut}_G(S') = \text{mincutcost}_G(S)$. Lemma 4.3 also holds for the induced subgraph $G[S]$, with $\text{cut}_{G[S]}(S') = \text{mincutcost}_{G[S]}(S') = \text{mincutcost}_G(S')$, as otherwise this cut would be the minimum cut of $G[S]$. Therefore $\text{cut}_G(S') = \text{cut}_{G[S]}(S')$ and $\text{cut}_G(S', V \setminus S) = 0$. □

Using Observation 4.11 we can now check if $\text{cut}_G(A \cup N(c_S), V \setminus S) = 0$ and avoid calculating all minimum cuts, if $A \cup N(c_S)$ does not satisfy the condition.

Zero-Edges

If there is a set $S' \subset S$ with nodes on both sides of the chosen mincut satisfying Observation 4.9 to Observation 4.11, then we can check if it satisfies the merge condition (3.1). If it does, then we found a set, merge it, and call the algorithm on the merged graph. But if it does not satisfy the merge condition, then there are three possible cases:

1. If $\text{mincutcost}_G(S') = \text{mincutcost}_G(S)$, then $\text{cut}_G(S') > \text{mincutcost}_G(S)$;
2. If $\text{cut}(S') = \text{mincutcost}_G(S)$, then $\text{mincutcost}_G(S') < \text{mincutcost}_G(S)$;
3. Otherwise $\text{mincutcost}_G(S') < \text{mincutcost}_G(S)$ and $\text{cut}_G(S') > \text{mincutcost}_G(S)$.

In all cases, we can only reduce the cut of S' or increase the mincut of S', by adding nodes from $S \setminus S'$ that do not increase the deficiency and have no edges to nodes in $V \setminus S$. Formally these nodes are

$$C = \{u \in S \setminus S' \mid \text{def}_G(S' \cup \{u\}) = 0, \text{cut}_G(\{u\}, V \setminus S) = 0\}.$$

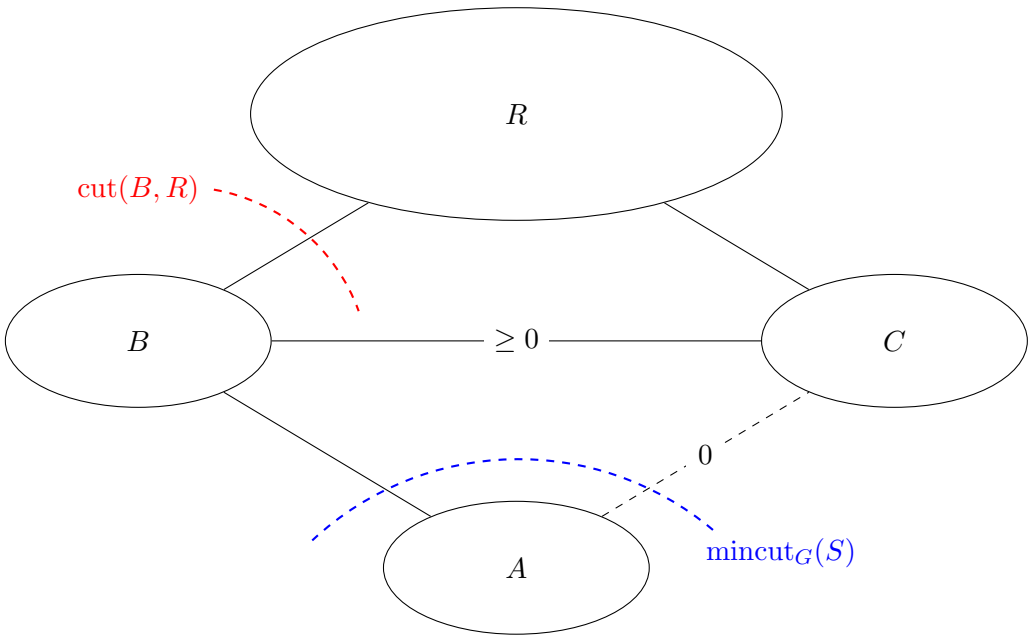

Figure 4.2: Partitioning of a the induced subgraph $G[S]$ into the four node sets A, B, C, and R for a graph G and $S \subseteq V$, as described below.

We call C the candidate set for nodes that could be added to S', without increasing the deficiency and the outside cut cost, to satisfy the merge condition. Note that every $u \in C$ has zero-edges to A, as u does not increase the deficiency and $u \notin N(c_S)$. Hence, if there are no nodes with zero-edges in the graph or, more specifically, no nodes with zero-edges from one side of the mincut to the other side, then the only set with nodes on both sides of the minimum cut is $N(c_S)$. Every other node that is not in $N(c_S)$ has a deficiency cost of at least one, to a node on the other side of the cut. If C is empty, then there are also no other possible sets other than $A \cup N(c_S)$ that could satisfy the merge condition.

Faster Mincut Calculation

With Observation 4.9 to Observation 4.11, we know that one side of the minimum cut is part of S', every node incident to the mincut is part of S', and we can only fix the merge condition (3.1) by including nodes from C in our set S'.

To illustrate this, Figure 4.2 shows the partitioning of the induced subgraph $G[S]$. Again, the minimum cut of $G[S]$ is called $c_S = (A, \bar{A})$. Without loss of generality, the node set $A \subset S$ is the side of the minimum cut of S that is a proper subset of S', whereas $B = N(c_s) \cap \bar{A}$ are the nodes incident to minimum cut on the other side of the minimum cut. Lastly, there is the rest of the graph $R = \bar{A} \setminus (C \cup B)$, which is only connected to C and/or B.

Algorithm 2 calculates all possible mincuts in $G[S]$, but we are only interested in the cuts separating R and $A \cup B$, since we know that A and B are part of S', i.e. $A \cup B \subset S'$. With that knowledge, we can improve the minimum cut calculation as follows: First of all, we need to cut B from R, with fixed cost $\text{cut}_G(B, R)$. In the resulting graph $A \cup B$

only has positive or zero-edges to C and no edges to R. Therefore, we can merge $A \cup B$ into a single node q while summing up the edge weights in the same way as described in Definition 3.1. We can reduce the number of nodes even more by only considering the nodes in R, which are neighbours of C, i.e. $R' = R \cap N(C)$. We now look for a minimum cut with cost $\text{mincutcost}_G(S) - \text{cut}_G(B, R)$, separating q and R'. If there is such a cut, then we found a set that potentially satisfies the merge condition (3.1). As we now only have to look at a (hopefully) small number of nodes instead of the whole induced subgraph $G[S]$, then this optimisation should speed up the running time in practise.

Also, note that if $\text{cut}_G(B, R) > \text{mincutcost}_G(S)$, then a cut that separates $A \cup B$ from R with cost $\text{mincutcost}_G(S)$ does not exist. Therefore, we can also abort the check in this case.

Running Time

As mentioned above, the high running time of Algorithm 1 is primarily due to the calculation of all minimum cuts and the check if one of these minimum cuts satisfies the merge condition (3.1). However, with the rules presented above, we can omit the calculation entirely in most cases, as we will see in Chapter 5.

Lemma 4.12. *Let $G = (V, E, s)$ be a graph and $S \subset V$. If neither the node set $A \cap N(c_S)$ nor $\bar{A} \cap N(c_S)$ for any minimum cut $c_S = (A, \bar{A})$ of G satisfy the conditions presented above, then we can omit calculating and looping over all minimum cuts in Algorithm 1. If the algorithm can omit this step in every recursion step, then the algorithm has a running time of $\mathcal{O}(n^2(m + n\log(n)))$.*

Proof. For a set $S' \subset S$ with nodes on both sides of the minimum cut c_S that satisfies the merge condition (3.1), either $A \cap N(c_S) \subset S'$ or $\bar{A} \cap N(c_S) \subset S'$, as shown in Observation 4.9 and Observation 4.11. Therefore if none of these sets satisfy the conditions presented above, there is no set S' with nodes on both sides of the minimum cut. Testing whether the deficiency of these two sets is equal to zero is possible in $\mathcal{O}(n^2)$ time. The same goes for calculating the outside cut cost to $V \setminus S$ and test if there are any zero-edges. Therefore, one single recursion step of Algorithm 2 has a running time of $\mathcal{O}(nm + n^2\log(n))$, including the checks presented above and without the need to calculate and loop over all minimum cuts. If the algorithm can omit the calculation and loop over all minimum cuts in Line 8 in every recursion step, then the overall running time of Algorithm 1 is $\mathcal{O}(n(nm + n^2\log(n)) = \mathcal{O}(n^2(m + n\log(n)))$. □

Summary

In this chapter, we introduced Algorithm 1 for the data reduction rule *Almost Clique*, which finds a set $S \subseteq V$ satisfying the merge condition (3.1), if such a set exists. We provided a proof of correctness of Algorithm 1 and showed that its running time is $\mathcal{O}(n^4 \cdot (m + n\log(n)))$. To improve the running time in practise of the algorithm, we showed several properties of S and conditions that can be used to speed up the running time in practise. In the following chapter, we test how effective the *Almost Clique* rule is in practise and if the rule can be carried out in a reasonable time, despite the high theoretical running time.

Chapter 5

Experiments

In this chapter, we present the implementation of Algorithm 1 and test how the algorithm performs in practise.

5.1 Implementation

We implemented a modified version of Algorithm 1 in the programming language Julia. We chose Julia as the programming language, as it enabled us to quickly test new ideas because of Julias modern programming paradigm without sacrificing speed. The performance of Julia is similar to C as shown in benchmarks[1]. The full implementation of the algorithm and the test setup can be found on Github[2].

In this modified version, we included the running time improvements presented in Section 4.3. We further return all possible sets that satisfy the merge condition (3.1), rather than looking for just the biggest set. *Almost Clique* also merges connected components that already form a clique into a single node, since they satisfy the merge condition (3.1). We added a check if the current connected component is a clique before applying the data reduction rule, as there is no real benefit in merging the nodes in this case.

To compute the minimum cut of a graph, we used the implementation by Henzinger et al. [Hen+20], which uses integer weighted graphs. Therefore our implementation also requires integer weighted input graphs.

5.2 Experiments

To test the implementation, we used the Cluster Editing instances of the *PACE Challenge 2021*[3]. This dataset includes primarily biologically graphs and additionally randomly generated graphs. For a more detailed description of the dataset, please refer to the PACE Challenge 2021 paper by Kellerhals et al. [Kel+21].

Because of Lemma 2.1, we treat each connected component of the test graphs as a single graph. We only tested the algorithms on graphs with 100 or more nodes and only

[1] https://julialang.org/benchmarks/
[2] https://github.com/venondev/AlmostCliquePoly
[3] https://github.com/PACE-challenge/Cluster-Editing-PACE-2021-instances

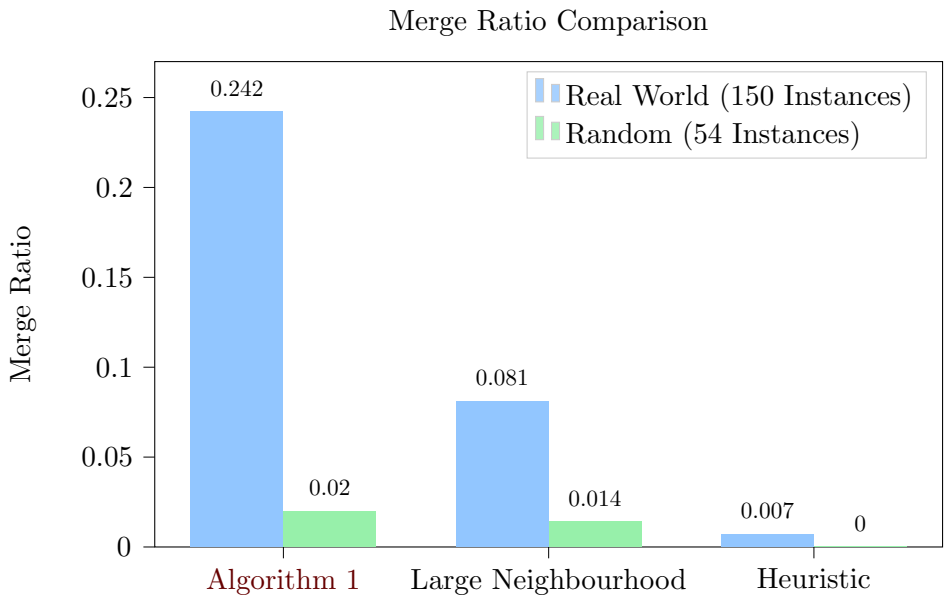

Figure 5.1: Comparison of the merge ratio (number of nodes merged / n) of the three algorithmic approaches "Large Neighbourhood", Algorithm 1 and "Heuristic" and the two instance categories real world and random.

graphs which are not cluster graphs already, as such instances are easy to solve. This resulted in 204 different graphs from the PACE challenge dataset, with 150 real-world instances and 54 randomly generated instances. The edge weights are floating-point values, and as our implementation uses integer weights, we multiplied the edge weights by a factor of 1000 to not lose too much accuracy.

We also implemented two other approaches for finding sets that satisfy the merge condition (3.1) to compare the results of our algorithm. The first one is the *Large Neighbourhood* (LN) approach, similar to Cao and Chen [CC12], where we use the closed neighbourhood $N_G[u]$ as the node set to test the merge condition (3.1) for every node $u \in V$. The second one is the implementation of the heuristic presented by Böcker, Briesemeister, and Klau [BBK08] of *Almost Clique*, which we embedded in our implementation.

5.3 Results

We were first interested in how much the graph size shrank after applying the data reduction rules. As one can see in Figure 5.1, Algorithm 1 merges 24.2% of the nodes of the real-world instances, roughly three times the amount of nodes, compared to the LN approach, which merges 8.1% of the nodes. The algorithms perform very poorly on the random instances, with nearly the same amount of merged nodes, 1.4% using the LN approach and 2% using Algorithm 1. The heuristic implementation by Böcker, Briesemeister, and Klau [BBK08] does not perform well on the dataset, with 0.7% for real-world instances and 0% for random instances.

When looking at the histogram of merge ratios in Figure 5.2, it shows that for most

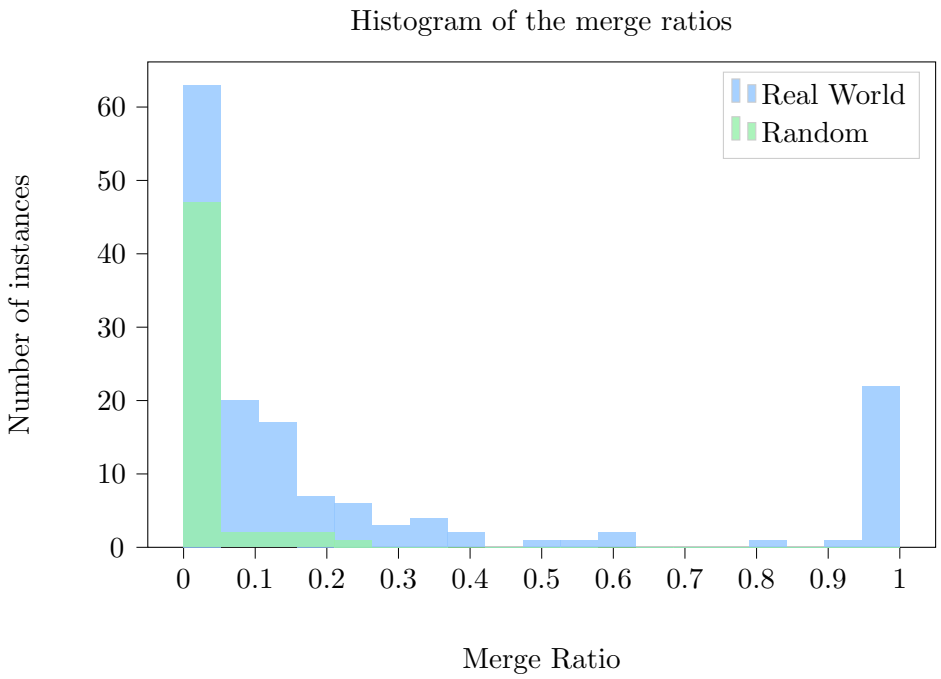

Figure 5.2: Histogram showing the number of instances with respect to the merge ratio (number of nodes merged / n).

graphs of the test set the rule reduces the graph size only slightly or not at all. But there are also 23 graphs which got solved entirely or almost entirely.

After testing how much the instance size shrank after applying the different algorithms, we wanted to know how many graph instances were solved completely. When using Algorithm 1, 11.3 % of the instances got solved completely, around 1.5 % when using the LN approach, and 0.5% when using the heuristic by Böcker, Briesemeister, and Klau [BBK08].

Running Time

We then tested how effective the optimisations described in Section 4.3 are in practise. It turns out that they are very effective indeed. They allowed us to skip Line 8 of Algorithm 2 in every recursion step of every input graph of the test set. Hence it is likely that for most graphs, the algorithm has a running time of $\mathcal{O}(n^2(m + n\log(n)))$, as described in Section 4.3.

The actual running time of our implementation is quite high. In Figure 5.3 we compare the running time of the algorithmic approaches LN and Algorithm 1.

Except for five instances, every instance was solved in less than 140 seconds, while the large neighbourhood (LN) approach only takes up to 3 seconds. For most of the instances, Algorithm 1 is more than ten times slower than the LN approach. For some instances, the algorithm is up to 1000 times slower. These are mostly the large instances of the newsgroup dataset described in the PACE challenge paper, with more than 3000 nodes. Using our algorithm, every random graph instance took longer while slightly

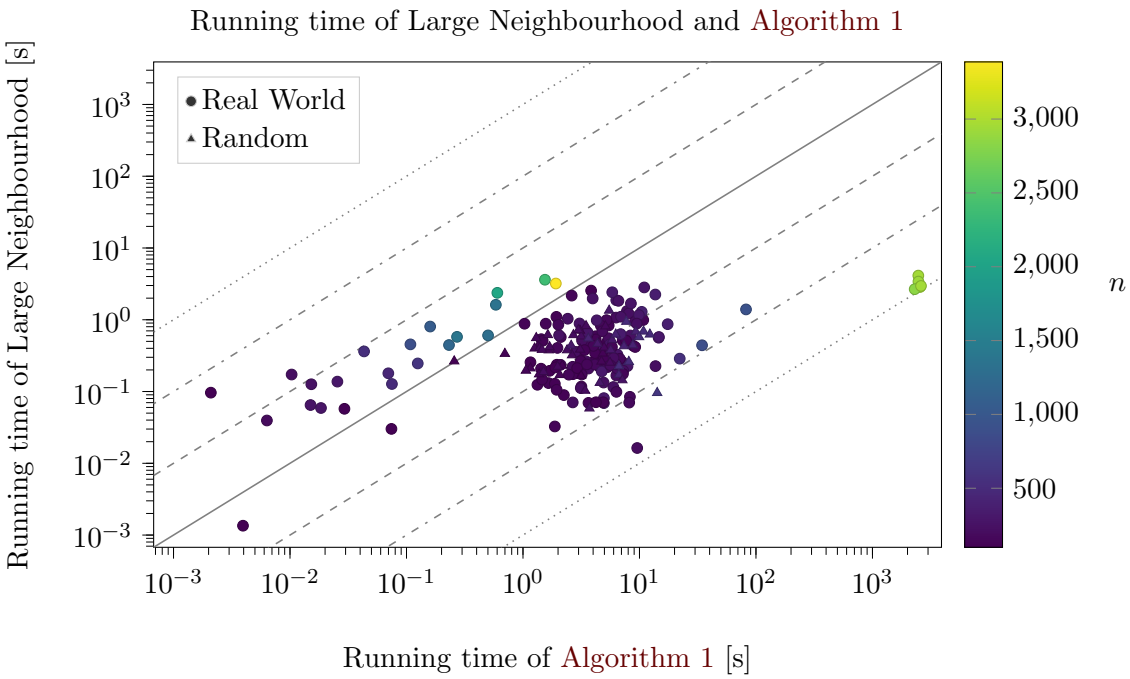

Figure 5.3: Comparison of the running time of Algorithm 1 and the Large Neighbourhood (LN) approach. Instances on the solid diagonal line have the same running time. In contrast, instances below the solid line take longer using Algorithm 1 and instances above the solid line take longer when using the LN approach. The dashed line represents a 10 time, the dashed-dotted 100 time, and the dotted line a 1000 time increase/decrease.

increasing the overall merge ratio for random instances. Hence, for these types of graphs, the Large Neighbourhood approach should be sufficient. There are a few instances where Algorithm 1 is faster. In these cases, the algorithm finds a large set of nodes to merge early on in the recursion and does not have to calculate n different mincuts, one for each node $u \in V$, as the LN approach does.

Interestingly, the recursion depth of the five outliers, depicted on the far right side of Figure 5.3, is the same as the number of nodes in the input graph, and no nodes got merged. This means that the minimum cut Algorithm 1 computes in each recursion step only separates a single node from the rest of the graph for these instances. Ideally, Algorithm 1 splits the graph into two halves in each recursion step. Therefore the recursion depth should be $\log(n)$ in the best case, provided that the algorithm does not find a set to merge earlier. In practise, unfortunately, this is not the case for most instances, which can be seen in Figure 5.4. The red line denotes the upper bound for the recursion depth, which is the number of nodes of the instance, while the green line marks the logarithm of the number of nodes, which is a lower bound under the assumption that the algorithm does not find a set to merge earlier, or stops earlier when the graph is already a cluster graph. For most of the instances, the recursion depth is nearly the amount of nodes in the graph. This means that in most recursion steps, the minimum

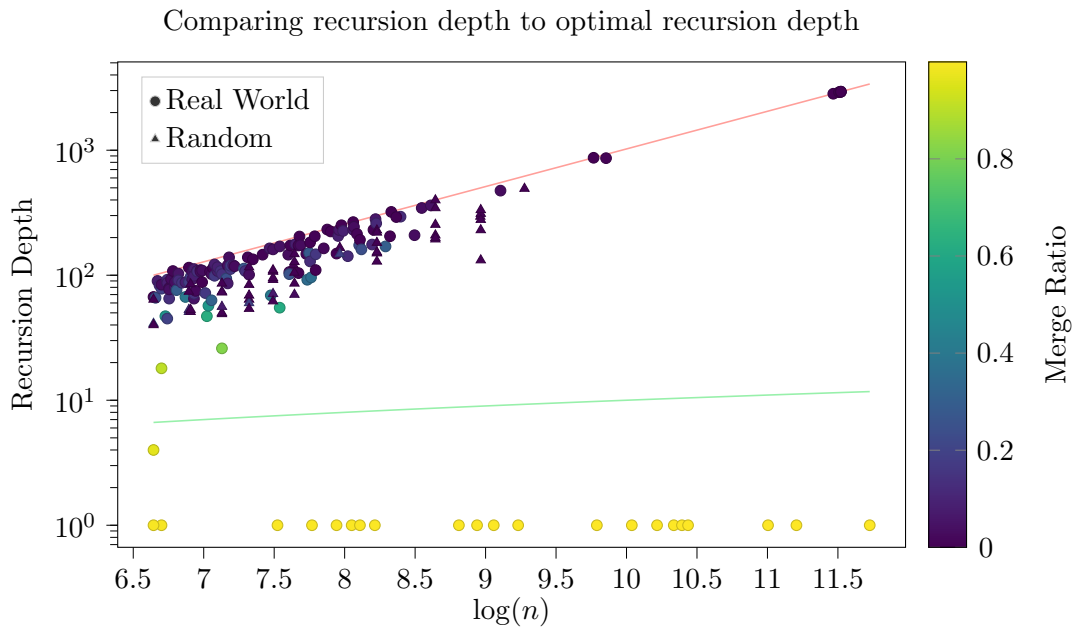

Figure 5.4: Comparing the recursion depth of the instances to their optimal recursion depth $\log(n)$. The red line denotes the upper bound of the recursion depth, which is the number of nodes n. Assuming that the algorithm does not find a set of nodes to merge earlier and the graph is not a cluster graph after merging a set of nodes, the green line denotes the lower bound of the recursion depth, which is $\log(n)$. The instances are coloured according to their respective merge ratio (number of nodes merged / n).

cut only cuts out a single node. Consequently, the size of the set $S \subseteq V$ that the algorithm looks at within each recursion step only decreases slowly. This is a problem because the most expensive operation in Algorithm 1 is the minimum cut calculation, whose running time depends on the size of the induced subgraph $G[S]$. Most of the graphs, including the five outliers mentioned above, compute many minimum cuts on large graphs, resulting in high running times in practise. There are some instances at the bottom of the graph where the algorithm merged the graph (nearly) entirely, or after merging some nodes, the graph was solved completely. Hence the recursion depth is less than $\log(n)$.

Summary

In this chapter, we analysed the *Almost Clique* data reduction rule and our implementation of Algorithm 1. We found that *Almost Clique* works well on the test dataset, reducing the real world graph instances by 24% on average. In comparison to the heuristic approaches, *Almost Clique* achieves better results overall. Notably, *Almost Clique* solves 11% of the graphs completely, including some larger graphs with up to 3000 nodes. Despite the improvements in Section 4.3, Algorithm 1 still comes at the cost of a high running time. We investigated why this is the case and found out that the main reason is the recursion behaviour. Instead of two equally sized sets of nodes,

Algorithm 1 splits the graph into one larger and a smaller set in each recursion step for most graphs. This results in a higher recursion depth and more expensive minimum cut calculations. Regardless of the high running time, the reduction ratio of Algorithm 1 shows to be good. Thus, the algorithm would work well as a preprocessing step before using an exact solver.

Chapter 6

Conclusion

In this thesis, we provided a full proof of correctness of the *Almost Clique* data reduction rule by Böcker, Briesemeister, and Klau [BBK08, Rule 4]. Moreover, we showed that applying the data reduction rule exhaustively is possible in polynomial time, while previously only heuristic approaches were known. As shown in Section 5.3, the rule works quite well, solving 11% of the graph instances entirely and reducing the graph size by 24% on average for real world instances. However, our algorithm comes at the cost of a high running time in practise. Thus, it might be too slow to use it inside a branch-and-bound algorithm but could be very useful as a preprocessing step before using a branch-and-bound or ILP approach to solve Weighted Cluster Editing.

We further came across the problem of weighted cluster edge deletion, which is closely related to Weighted Cluster Editing. The problem is the same, with the restriction that only edge deletions are allowed and no edge insertions. Algorithm 1 achieves this when preprocessing the input graph such that every non-edge weight is $-\infty$. Hence, the algorithm never inserts edges. Using this small adaptation, *Almost Clique* can also be applied to weighted cluster edge deletion.

Böcker, Briesemeister, and Klau [BBK08] claim that the data reduction rule does not improve the reduction of the test instances when using it in combination with other data reduction rules they show and when using their heuristic approach. This leads to the question of whether this is the case because the heuristic might not find all possible node sets *Almost Clique* could be applied to or if most of the cases are already covered by other data reduction rules. Furthermore, it would be interesting to see how *Almost Clique* can be used inside a search tree, which likely would require speeding up the minimum cut calculation or saving parts of the solution.

Even though we now know that it is possible to apply *Almost Clique* in polynomial running time, the question is if there is a faster algorithm than Algorithm 1. To achieve this, one idea would be to analyse sets satisfying the merge condition (3.1), now that it is possible to calculate such sets efficiently. It would be interesting to see if they all share common properties that could be used to speed up the calculation of such sets.

Literature

[BBC04] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. *Correlation clustering*. In: *Machine Learning* 56.1 (2004), pp. 89–113 (cit. on p. 9).

[BBK08] Sebastian Böcker, Sebastian Briesemeister, and Gunnar Klau. *Exact Algorithms for Cluster Editing: Evaluation and Experiments*. In: *Algorithmica* 60 (2008), pp. 316–334 (cit. on pp. 11, 17, 20, 22, 23, 32, 33, 37).

[BDSY99] Amir Ben-Dor, Ron Shamir, and Zohar Yakhini. *Clustering gene expression patterns*. In: *Journal of Computational Biology* 6.3-4 (1999), pp. 281–297 (cit. on pp. 9, 11).

[Böc12] Sebastian Böcker. *A golden ratio parameterized algorithm for cluster editing*. In: *Journal of Discrete Algorithms* 16 (2012), pp. 79–89 (cit. on p. 11).

[Bö+09] S. Böcker, S. Briesemeister, Q.B.A. Bui, and A. Truss. *Going weighted: Parameterized algorithms for cluster editing*. In: *Theoretical Computer Science* 410.52 (2009), pp. 5467–5480 (cit. on p. 14).

[CC12] Yixin Cao and Jianer Chen. *Cluster Editing: Kernelization Based on Edge Cuts*. In: *Algorithmica* 64.1 (2012), 152–169 (cit. on pp. 11, 17, 23, 32).

[CM12] Jianer Chen and Jie Meng. *A 2k kernel for the cluster editing problem*. In: *Journal of Computer and System Sciences* 78.1 (2012), pp. 211–220 (cit. on p. 11).

[Cyg+15] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015, pp. 18–19 (cit. on p. 13).

[Gra+05] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. *Graph-modeled data clustering: Exact algorithms for clique generation*. In: *Theory of Computing Systems* 38.4 (2005), pp. 373–392 (cit. on p. 11).

[GW89] Martin Grötschel and Yoshiko Wakabayashi. *A cutting plane algorithm for a clustering problem*. In: *Mathematical Programming* 45.1 (1989), pp. 59–96 (cit. on p. 11).

[Hen+20] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. *Finding All Global Minimum Cuts in Practice*. In: *European Symposium on Algorithms*. Vol. 173. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 59:1–59:20 (cit. on pp. 10, 14, 31).

[HH15] Sepp Hartung and Holger H Hoos. *Programming by optimisation meets parameterised algorithmics: A case study for cluster editing*. In: *International Conference on Learning and Intelligent Optimization*. Springer. 2015, pp. 43–58 (cit. on p. 11).

[Kar00] David R Karger. *Minimum cuts in near-linear time*. In: *Journal of the ACM* 47.1 (2000), pp. 46–76 (cit. on p. 27).

[Kel+21] Leon Kellerhals, Tomohiro Koana, André Nichterlein, and Philipp Zschoche. *The PACE 2021 Parameterized Algorithms and Computational Experiments Challenge: Cluster Editing*. In: *International Symposium on Parameterized and Exact Computation*. Vol. 214. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 26:1–26:18 (cit. on pp. 11, 31).

[KM86] Mirko Křivánek and Jaroslav Morávek. *NP-hard problems in hierarchical-tree clustering*. In: *Acta Informatica* 23.3 (1986), pp. 311–323 (cit. on p. 9).

[NNI00] Hiroshi Nagamochi, Yoshitaka Nakao, and Toshihide Ibaraki. *A fast algorithm for cactus representations of minimum cuts*. In: *Japan Journal of Industrial and Applied Mathematics* 17.2 (2000), pp. 245–264 (cit. on pp. 10, 26, 27).

[SS00] Roded Sharan and Ron Shamir. *Center CLICK: A Clustering Algorithm with Applications to Gene Expression Analysis*. In: *International Conference on Intelligent Systems for Molecular Biology*. AAAI, 2000, pp. 307–316 (cit. on pp. 9, 11).

[SW97] Mechthild Stoer and Frank Wagner. *A simple min-cut algorithm*. In: *Journal of the ACM* 44.4 (1997), pp. 585–591 (cit. on p. 26).

[Wit+07] Tobias Wittkop, Jan Baumbach, Francisco P Lobo, and Sven Rahmann. *Large scale clustering of protein sequences with FORCE - A layout based heuristic for weighted cluster editing*. In: *BMC Bioinformatics* 8.1 (2007), pp. 1–12 (cit. on pp. 9, 11).