UNIVERSITY OF BERGEN

DEPARTMENT OF INFORMATICS

ALGORITHMS

Kernelization for Balanced Graph Clustering

Author:
Andreas STEINVIK

Supervisor:
Petr GOLOVACH

Master Thesis

September 10, 2020

Abstract

The problems of FACTOR-α BALANCED CLUSTERING and DIFFERENCE-δ BALANCED CLUSTERING ask whether it is possible to modify a graph such that it becomes a cluster graph where no cluster has a size larger than a given multiplicative factor α or absolute difference δ relative to any other cluster in the graph, by doing at most k graph modifications. In this thesis we study the problems with respect to the graph modification operations *Vertex Deletion*, *Edge Addition* and *Edge Deletion*.

We will show NP-completeness and give polynomial kernels for each version. In the case when edge addition is the operation allowed, we give a kernel with $\mathcal{O}(k)$ vertices, while we give kernels with $\mathcal{O}(k^2)$ vertices for edge deletion and edge editing, which is the problem when given the possibility of both adding and deleting edges. For the case of vertex deletion we give a kernel with $\mathcal{O}(k^2)$ vertices for the weighted version and a kernel with $\mathcal{O}(k^4)$ vertices for the unweighted version. We will also provide a $\mathcal{O}(3^k \cdot (|V| + |E|))$ time FPT algorithm for vertex deletion.

Acknowledgements

I would like to thank my supervisor, Petr Golovach, for excellent guidance during the work on this thesis. I would also like to thank my fellow students, and particularly Ole Magnus, Nora and André for all the long lunches. Finally i would like to thank my family for all the support, and for housing me during the coronavirus lockdown.

Andreas Steinvik
September 10, 2020

Contents

Chapter 1

Introduction

In computer science, a *clustering problem* is the task of grouping the objects of a data set in such a way that objects are more similar to the other objects in the same cluster than to objects in other clusters. This similarity can be any metric, and what property we choose to cluster by depends on the data set used. A simple real life example of a clustering problem is PARTY PLANNING, which is the problem asking whether given a list of people attending a party and some information on each guest, e.g. their interests or which other guests they know, is it possible to find a way of grouping people so that all people sitting at the same table have something in common. An example of a scientific use is the task of finding clusters in a population based on the similarities and differences in their DNA. Clustering problems similar to the aforementioned examples occur in many fields of study, ranging from social sciences to biology and medicine [26].

Clustering problems are not bound to any specific data structure, and can occur in most types of data structures. In this thesis we will focus on clustering in *graphs*, an abstract data structure composed of *vertices* and *edges*. The vertices are objects in our data set, for example the guests in our PARTY PLANNING example. The edges are a *relation* between vertices, and in our PARTY PLANNING example, one such relation could be "guests A and B are friends". As a lot of study has been done into the theoretical properties of graphs, they are a widely used data structure in computer science. There are no single unified definition into what constitutes a *cluster* in a graph, with different definitions based on among others the vertex density of the graph or its connectivity. In this thesis we use the definition of a cluster as a *complete component*, which is a component which is also a clique. By this definition, a cluster graph is then a graph that is a *disjoint union of cliques*. For a more comprehensive study of different cluster measures, see the survey by Shaeffer [23].

If the problem is asking "Is this graph a cluster graph?", it is easy to see that this is then quite an easy problem to decide. One can by checking each vertex and their edges see if for each vertex it has edges to all other vertices in the same cluster, and no edges to vertices in other clusters. A harder question can be "how far away from a cluster graph is this graph?". Problems of this type are called *modification problems*, and ask whether we by doing some modification operation can alter the graph to become a cluster graph.

The types of modification operations we will consider are *vertex deletion*, *edge addition* and *edge deletion*. Often when working with modification problems, the problems can be stated as modifying a graph to get a *H-free graph* where H is a set of forbidden subgraphs. That is, can we modify a graph such that no member of H can be found as an induced subgraph. Examples of such problems is the problem of finding a VERTEX COVER which ask for a minimum set of vertices such that all edges in the graph are incident to a vertex in the vertex cover, can be restated as what is the minimum number of vertex deletions we need for the graph to be an edge-free graph. Clustering problems can likewise be restated as modifying to a P_3-free graph, where P_3 is a path of 3 vertices.

For modification problems we often look for a *minimum* number of modifications to reach the property. As the problems we will study in this thesis are NP-hard, it is assumed that there does not exist polynomial algorithms to find an optimal solution. This makes the the framework of *parameterized complexity* introduced by Downey and Fellows [11] very useful. In parameterized complexity, instead of only evaluating the running time of an algorithm based on the size of the input graph, we can add one or more additional *parameters*. These parameters can be e.g. the number of modifications we need in a solution, or it can be based on some property of the input. Working with k as the number of modifications we need, we can rephrase the PARTY PLANNING question to "is it possible to achieve a clustering of the guests by at most k modifications?".These modifications can in our problems be either that you remove at most k guests from the guest list, or that you accept at most k unfulfilled relations. That is, for example having an unknown person at your table, or that you have a friend at another table. The main principle of parameterized complexity is that even though the problems we study are hard to solve in the general case, we try to achieve algorithms which for small values of the parameter does admit fairly efficient algorithms. As it is assumed that no polynomial time algorithms for solving the problems exist, we will in this thesis work on finding *kernelization algorithms*. This is a preprocessing algorithm, which aims to reduce the input to its hard *kernel*. For a more formal definition of parameterized complexity, see Chapter 2.

There have been a lot of work done into both finding faster algorithms and better kernelizations for both the regular graph clustering problem and many different variations. The problem of CLUSTER VERTEX DELETION is the problem of finding a set of vertices in the graph such that by deleting them we get a cluster graph. As the property of being a cluster graph is a *hereditary* property, meaning that if a graph G has a property, then so does any induced subgraph of G, CLUSTER VERTEX DELETION was shown to be NP-complete by Lewis and Yannakakis [20] when they showed that all vertex deletion problems are NP-complete when the desired graph property is non-trivial and hereditary. The first study of the parameterized complexity of CLUSTER VERTEX DELETION was by Huffner et al. [17], and later work by Boral et al.[6] have produced an algorithm running in time $\mathcal{O}(1.9102^k(|V|+|E|))$. The smallest kernel for CLUSTER VERTEX DELETION is the kernel given by Le et al. [19] which shows that the problem admit a subquadratic kernel with $\mathcal{O}(k^{\frac{5}{3}})$ vertices. In the paper by Huffner et al. [17] they also study two variations of CLUSTER VERTEX DELETION, giving an $\mathcal{O}(2^k k^9 + |V| \cdot |E|)$ algorithm for WEIGHTED CLUSTER VERTEX DELETION, which is the problem of finding a cluster graph when the cost of deleting vertices is given by a weight function for each vertex. And a $\mathcal{O}(2^k k^6 \log k + |V| \cdot |E|)$ algorithm for d-CLUSTER VERTEX DELETION, which is the problem of finding a cluster graph with exactly d clusters.

We will also study the problems where we modify edges. When we are allowed to both add and delete edges, we get the problem CLUSTER EDITING, also called CORRELATION CLUSTERING. The problem was shown to be NP-complete by several independent people, one such being Shamir et al.[24]. The first results on the parameterized complexity of CLUSTER EDITING were given by Gramm et al [14] where they gave a $\mathcal{O}(2.27^k + |V|^3)$ algorithm and a kernel with $\mathcal{O}(k^2)$ vertices. The fastest algorithm has since then been improved to $\mathcal{O}(1.62^k + |V| + |E|)$ by Böcker [2], and the smallest kernel was provided by Cao and Chen [7] and gives a kernel with $\mathcal{O}(2k)$ vertices. CLUSTER EDITING has also received a lot of practical attention, especially within the field of bioinformatics. For a more in depth read into the practical implementations see the survey by Böcker and Baumbach [3]. For the related problem of CLUSTER COMPLETION, which is the problem where the only allowed operation is adding in edges, we can easily observe that this problem is polynomial time solvable. To solve CLUSTER COMPLETION we need to compute the components in the graph, and then add in all edges missing for the graph to become a cluster graph. The problem of CLUSTER DELETION, where only edge deletions are allowed has not gotten as much attention as CLUSTER EDITING, but a kernel with $\mathcal{O}(k^3)$ vertices was given by Gramm et al. [14] and an algorithm running in time $\mathcal{O}*(1.415^k)$ was given by Böcker et al. [4]. For an overview of what is known about the parameterized complexity of edge modification problems, see the survey by Crespelle et al.[8].

1.1 Our Results

In this thesis we will look at a version of graph clustering we call FACTOR-α BALANCED CLUSTERING, where we in addition to looking for a cluster graph have the added constraint that the sizes of each cluster has to have a size bounded by a multiplicative factor of the sizes of the other clusters. If for example we have $\alpha = 2$, no cluster can be more than double the size of any other cluster. For our PARTY PLANNING problem, this can be seen as a natural extension: for our party we do not want 40 guests at one table, while the remaining 5 guests are split up into 5 different single tables. By then setting α to a reasonable value based on our preferences we instead look for a table configuration of more equally sized tables. We will also provide results for a related version called DIFFERENCE-δ CLUSTERING, where instead of using an multiplicative factor, we look at the absolute difference between cluster sizes. Note that unlike for regular cluster graphs, being a Factor-α balanced cluster graph is not a hereditary property. Because of this we also have to provide proofs for NP-completeness of the vertex deletion versions we will study.

For the problems of FACTOR-α BALANCED CLUSTER VERTEX DELETION (FACTORCVD) and DIFFERENCE-δ BALANCED CLUSTER VERTEX DELETION (DIFFERENCECVD) we will be studying both the unweighted and the weighted versions of the problems. In the unweighted version the operation is vertex deletion, while in the weighted versions we use an operation called weight decreasing. Weight decreasing works by reducing the weight of a vertex by 1 if its weight is larger than 1, and deleting the vertex if its weight equals 1. Observe that the unweighted problems are special cases of the weighted problems where we have the weight of each vertex equaling 1. This property is used to transform an unweighted instance into an instance of the weighted problem. Let $\alpha \in \mathbb{R}$ be a fixed constant such that $\alpha \geq 1$, and let $\delta \in \mathbb{Z}^+$ be a fixed integer such that $\delta \geq 0$.

FACTOR-α BALANCED CLUSTER VERTEX DELETION

Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$

Question: Is there a vertex set $X \in V$ such that $|X| \leq k$ and $G - X$ results in a cluster Graph (i.e a disjoint union of cliques) where for all components C_i, C_j in G, it holds that $\alpha \cdot |V(C_i)| \geq |V(C_j)|$?

DIFFERENCE-δ BALANCED CLUSTER VERTEX DELETION

Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$

Question: Is there a vertex set $X \in V$ such that $|X| \leq k$ and $G - X$ results in a cluster graph (i.e a disjoint union of cliques) where for all components C_i, C_j in G, it holds that $\delta + |V(C_i)| \geq |V(C_j)|$?

WEIGHTED FACTOR-α BALANCED CLUSTER VERTEX DELETION

Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$ and a weight function $w : V(G) \to \mathbb{N}$

Question: Is it possible to obtain a cluster graph G' from G by at most k weight decrease operations such that for all components C_i, C_j in G, it holds that $\alpha \cdot \sum_{v \in V(C_i)} w(v) \geq \sum_{v \in V(C_j)} w(v)$?

WEIGHTED DIFFERENCE-δ BALANCED CLUSTER VERTEX DELETION

Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$ and a weight function $w : V(G) \to \mathbb{N}$

Question: Is it possible to obtain a cluster graph G' from G by at most k weight decrease operations such that for all components C_i, C_j in G, it holds that $\delta + \sum_{v \in V(C_i)} w(v) \geq \sum_{v \in V(C_j)} w(v)$?

In this thesis we will show that the VERTEX DELETION problems are NP-complete, and then give a $\mathcal{O}(3^k(|V| + |E|))$ FPT-algorithm solving the problems. We will also give a polynomial kernel for the weighted problems with $\mathcal{O}(k^2)$ vertices and with each vertex having a weight bounded by $\mathcal{O}(k^2)$. We also show that this kernel can be used to obtain a kernel with $\mathcal{O}(k^4)$ vertices for the unweighted version. By combining the weighted kernel and the FPT-algorithm we are able to provide an algorithm solving the problems in time $\mathcal{O}(3^k k^4 + |V|^2(|V| + |E|))$.

For edge modification, we will study the three different variations of BALANCED CLUSTER COMPLETION, which is the problem of finding a balanced cluster graph by adding in edges, BALANCED CLUSTER DELETION which asks whether it is possible to find a balanced cluster graph by deleting edges, and BALANCED CLUSTER EDITING which is the problem of finding a balanced cluster graph when we have the option to both delete and add in edges. As we will study both the FACTOR-α and DIFFERENCE-δ versions of each problem we get a total of six different edge modification problems. Let $\alpha \in \mathbb{R}$ be a fixed constant such that $\alpha \geq 1$, and let $\delta \in \mathbb{Z}^+$ be a fixed integer such that $\delta \geq 0$.

FACTOR-α BALANCED CLUSTER COMPLETION (FACTORCC)

Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$

Question: Is there a set $A \subseteq \binom{V(G)}{2} \setminus E(G)$ such that $|A| \leq k$ and the graph $G + A$ is a cluster graph (i.e a disjoint union of cliques) where for all components C_i, C_j in G, it holds that $\alpha \cdot |V(C_i)| \geq |V(C_j)|$?

DIFFERENCE-δ BALANCED CLUSTER COMPLETION (DIFFERENCECC)

Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$

Question: Is there a set $A \subseteq \binom{V(G)}{2} \setminus E(G)$ such that $|A| \leq k$ and the graph $G + A$ is a cluster graph where for all components C_i, C_j in G, it holds that $\delta + |V(C_i)| \geq |V(C_j)|$?

FACTOR-α BALANCED CLUSTER DELETION (FACTORCD)

Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$

Question: Is there a set $A \subseteq E(G)$ such that $|A| \leq k$ and the graph $G - A$ is a cluster graph (i.e a disjoint union of cliques) where for all components C_i, C_j in G, it holds that $\alpha \cdot |V(C_i)| \geq |V(C_j)|$?

DIFFERENCE-δ BALANCED CLUSTER DELETION (DIFFERENCECD)

Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$

Question: Is there a set $A \subseteq E(G)$ such that $|A| \leq k$ and the graph $G - A$ is a cluster graph where for all components C_i, C_j in G, it holds that $\delta + |V(C_i)| \geq |V(C_j)|$?

FACTOR-α BALANCED CLUSTER EDITING (FACTORCE)

Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$

Question: Is there a set $A \subseteq \binom{V(G)}{2}$ such that $|A| \leq k$ and the graph $G \triangle A$ is a cluster graph where for all components C_i, C_j in G, it holds that $\alpha \cdot |V(C_i)| \geq |V(C_j)|$?

DIFFERENCE-δ BALANCED CLUSTER EDITING (DIFFERENCECE)

Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$

Question: Is there a set $A \subseteq \binom{V(G)}{2}$ such that $|A| \leq k$ and the graph $G \triangle A$ is a cluster graph where for all components C_i, C_j in G, it holds that $\delta + |V(C_i)| \geq |V(C_j)|$?

Unlike for CLUSTER COMPLETION, which is polynomial time solvable, we will in this thesis show that FACTOR-α BALANCED CLUSTER COMPLETION and its absolute difference version are NP-complete for any fixed α or δ respectively. We will also show NP-completeness for the other edge modification problems.

We will provide a kernel with $\mathcal{O}(k)$ vertices for FACTORCC and DIFFERENCECD, and give kernels with $\mathcal{O}(k^2)$ vertices for the problems of EDGE DELETION and EDGE EDITING.

1.2 Thesis Outline

In Chapter 2 we will provide definitions for notation and concepts used in this thesis. In Chapter 3 we will study the VERTEX DELETION problems. There we will start by proving NP-completeness, and then give an algorithm and a kernel. In Chapter 4 we will give proofs of NP-hardness and kernels for the edge modification problems. Lastly, in Chapter 5 we will present some open problems and related problems for future work.

Chapter 2

Preliminaries

2.1 Set Theory

A *set* is a collection of distinct objects where the ordering of the objects does not matter. Named set will have names in capital letters e.g. A or B. We will be using the common notation for common set operations and properties as *membership* $(\in, \notin)$, *subsets* $(\subseteq, \subset)$, *union* and *intersection* $(\cup, \cap)$, *difference* $(\setminus)$ and *cardinality* or *size* $(|A|)$. We will also use the notation $\binom{A}{2}$ which gives a set of all distinct pairs of elements from A. The *symmetric difference* $\triangle$ of two sets A and B is defined as the union minus the intersection, $A \triangle B = (A \cup B) \setminus (A \cap B)$.

2.2 Algorithms and Complexity

In this thesis we will work with *parameterized problems*, and finding efficient ways to solve them. To do this we first have to define what a *decision problem* is, give some formal definitions of what a *parameterized algorithm* is, and how we evaluate its *complexity*. We will also show how to evaluate the efficiency of an algorithm by introducing *big-O notation*. Lastly we will define what a *kernelization algorithm* is.

2.2.1 Computational Problems

An *alphabet* Σ is a fixed, finite set of symbols, and a *string* is a sequence of symbols from Σ. We denote all possible *strings* over Σ by Σ^*. A *language* L over Σ is some subset of strings $L \subseteq \Sigma^*$.

A *decision problem*, is a *language* $L \subseteq \Sigma^*$ where for a *string* $w \in \Sigma^*$, if $w \in L$ we say that w is , and likewise if $w \notin L$ we say that it is a NO-instance. A *parameterized decision problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$ where for an instance $(x, k) \in L$, x is some *string* over Σ and k is the *parameter*.

2.2.2 Algorithms

For this thesis it is sufficient to go with an informal definition of an *algorithm* as a finite set of rigorous instructions for how to solve a *problem*, but more formally an *algorithm* can be expressed by the mathematical concept of *Turing machines*. For the more formal definitions see the book by Sipser [25]. A *parameterized decision algorithm* accept as input an *instance* $(x, k) \in \Sigma^* \times \mathbb{N}$ and upon halting returns whether the instance is a YES-instance if $(x, k) \in L$ or a NO-instance if it is not.

2.2.3 Complexity

An informal definition of the complexity of an algorithm is that it is how much time and space it requires to solve a problem of a given size. In this thesis we will mostly talk about *time complexity*, the running time of an algorithm in the number of steps it takes to solve an instance. As this can vary between instances of same size we usually talk about the *worst case* running time, which is the maximum number of steps needed to solve instances of a given size. We describe running time as a function $f(n)$ where n is the size of our instance, and for a parameterized problem we also include the parameter into the function to get $f(n, k)$. Often it is not necessary to have the exact function $f(n)$, and more useful to talk about an *upper bound* to the running time. For this we use *big-O notation*, which for some function $f(n)$ is defined as: If the function $f(n) \leq c \cdot g(n)$ for a sufficiently big n and some constant c, we say that $f(n) = \mathcal{O}(g(n))$.

Using the *time complexity* we can categorize problems into *classes* by how difficult they are to solve. The classes most relevant for this thesis are the classes P, NP and FPT. P is the class of problems which can be decided in polynomial time. That is, there exist an algorithm which decides the problems with running time bounded by some function $f(n) = \mathcal{O}(n^c)$ where c is a constant. NP is the class of problems where given an instance I and a *certificate* $S \in \Sigma^*$, a string showing how this is an YES-instance, it can be *verified* in polynomial time whether it is correct. As we see by the definitions of P and NP we have the relation $P \subseteq NP$, but whether $P = NP$ or not is the famous *P vs. NP* problem [25].

The class FPT is the class of parameterized problems which can be solved in time $\mathcal{O}(g(k) \cdot n^c)$ where c is a constant and $g(k)$ is a computable function only dependent on the parameter k. The parameter k can be of many different types, like the size of the solution or some bound on a property of the input x. The motivation behind FPT is that for problems in FPT, even though they are not polynomial time solvable, given a small parameter k they still can have an efficient algorithm. Similar to the relation between P and NP, there is also a *hierarchy of classes* for parameterized problems, the *W-hierarchy*, where $W[0] \subseteq W[1] \subseteq W[2] \subseteq ... \subseteq XP$. The class of FPT is equal to the class W[0] and all problems that are complete for some class W[i] where $i > 0$ are believed to not have FPT algorithms. An example of such a problem is the W[1]-complete problem CLIQUE of finding a clique of size k in a graph. For a more thorough read on FPT and parameterized algorithms see the book by Cygan et. al. [9].

2.2.4 Kernelization

A related concept to a FPT algorithm is a *kernelization algorithm.* This is a *preprocessing algorithm* which takes an instance (x, k) of some problem as input and in time polynomial in $|(x, k)|$ outputs an instance (x', k') where $|x'|, k' \leq g(k)$ for some computable function $g(k)$ where (x', k') is a YES-instance if and only if (x, k) is a YES-instance. The reduced instance is often referred to as the *kernel* of the instance. The relation between kernelization and FPT is easy to see, as the *kernelization algorithm* runs in polynomial time and outputs a *kernel* bounded by some $g(k)$, it can then be combined with a superpolynomial decision algorithm to give an answer in FPT time. In this thesis we will also use the notion of a *bikernel* [1] , also called a *generalized kernel* [5]. A bikernel is an algorithm which takes as input an instance (x, k) of a language L, and in time polynomial in $|(x, k)|$ returns an instance (x', k') of another language L', such that $|x'|, k' \leq f(k)$ and $(x, k) \in L$ if and only if $(x', k') \in L'$. Notice that this is a generalization of a kernel, since for a kernel we have that $L = L'$.

Notice that although all problems in FPT have kernels, under some computational complexity assumptions not all problems admit polynomial kernels. One such example is the problem of k-LONGEST PATH. This asks whether a graph has a path of length k, and it is believed that the optimal kernels for this problem has a size exponential in k.

The most common strategy to create *kernelization algorithms* is by the use of *reduction rules.* A *reduction rule* is a function $f : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}$ which maps an instance (x, k) to an equivalent instance (x', k') in polynomial time and is *safe*. That is, the reduced instance (x', k') is a YES-instance *if and only if* (x, k) is a YES-instance. Often a *kernelization algorithm* is a sequence of such *reduction rules*, applied to the problem instance in order and often repeated until no rule applies. The running time such algorithms is then bounded by the running time of each rule, and how many times they can be applied in worst case. For a deeper read into the subject *kernelization* we refer to the book by Fomin et. al. [12]

2.3 Graph Theory

A *graph* is a tuple $G = (V, E)$ where V is a set of *vertices*, and $E \subseteq \binom{V}{2}$ is a set of *edges*. An *edge* is an unordered pair of vertices $e = \{u, v\} \in E$ where $u, v \in V$. We denote the set of vertices in a graph G as $V(G)$ and likewise the set of edges as $E(G)$. In this thesis we will only consider *undirected graphs*, which is graphs where the edges are *symmetric*, i.e the edges $\{u, v\}$ and $\{v, u\}$ are the same. We say that two vertices u and v are *adjacent* if there exist an edge $\{u, v\} \in E$, and an edge $e \in E$ is *incident* to a vertex w if $w \in e$. A *neighbour* of vertex u is the same as a vertex *adjacent* to u. The *open neighbourhood* of u denoted as $N(u)$ is the subset of vertices of $V(G)$ which are neighbours of the vertex u , while the *closed neighbourhood* $N[u]$ of u is defined as $N[u] = N(u) \cup \{u\}$. If two vertices have equal closed neighbourhoods, that is $N[u] = N[v]$, we say that they are *true twins*.

The number of edges incident to a vertex u is called the *degree* of u and is denoted $deg(u)$. If a vertex u has degree 0 we say that it is a *isolated* vertex. We will mostly follow the notation of Diestel [10] and recommend this book for a more in-depth study of graphs.

2.3.1 Subgraphs and Connectivity

A *subgraph* $G' = (V', E')$ of a graph $G = (V, E)$ is a graph where $V' \subseteq V$ and $E' \subseteq \binom{V'}{2} \cap E$. And the special case where E' contains all edges $\{v, u\} \in E$ if $u, v \in V'$,is an *induced subgraph*. For a set of vertices $U \subseteq V(G)$, the subgraph it induces in G is denoted as $G[U]$.

If for a sequence of vertices $U = \{u_0, u_1, ..., u_k\} \subseteq V(G)$ there exist a set of edges $E' = \{\{u_0, u_1\}, \{u_1, u_2\}..., \{u_{k-1}, u_k\}$ $E(G)$ we say that there exist a *path* between the end vertices u_0 and u_k. An *induced path* of length k in G, denoted P_k is an induced subgraph of k vertices forming a path of k vertices and $k-1$ edges. A *cycle* is a sequence of vertices $U = \{u_0, u_1, ..., u_k\}$ such that there is a path from u_0 to u_k, and there also exist an edge $\{u_k, u_0\} \in E$.

If for two vertices $x, y \in V(G)$ there exist a path between them, we say that x and y are *connected*, and if this is true for any pair of vertices in $V(G)$ we say the graph G is *connected*. An inclusion maximal connected subgraph of G is a *component*, and note that an isolated vertex is also a component. If for a graph $G = (V, E)$ all vertices in V are pairwise adjacent we say that G is a *complete graph*, and if a component in G is a complete graph, we say that the component is a *complete component*. If for a graph G all components are complete components we say that the graph is a *cluster graph*. Another and equal definition of a cluster graph is a P_3*-free graph*, i.e. a graph without induced paths of length 3. In this thesis we will use the term *balanced cluster graph* to mean a cluster graph in which each component is bounded in size relative to all other components, but how it is bounded is either not relevant or obvious from the context.

A *clique* is a subset of vertices $C \subseteq V(G)$ such that the induced subgraph $G[C]$ is a complete graph.

2.3.2 Graph Editing

If $G = (V, E)$ is a graph and $U \subseteq V(G)$, then the deletion $G - U = G'$ gives the graph $G' = G[V(G) \setminus U]$. Deletion of a set $A \subseteq E(G)$ is defined as $G - A = G'$ where $G' = (V(G), E(G) \setminus A)$. The operations will mostly be referred to as *vertex deletions* and *edge deletions*. The operation of *edge addition* is defined as $G + A = G'$ where $A \subseteq \binom{V(G)}{2} \setminus E(G)$ gives the new graph $G' = (V(G), E(G) \cup A)$. The combination of both adding edges and removing a set of edges $G \triangle A = G'$ where $A \subseteq \binom{V(G)}{2}$ is the *symmetric difference* of $E(G)$ and A, gives the graph $G' = (V(G), E(G) \triangle A)$. The graph G' then contains all edges which is in either $E(G)$ or A, but not in both.

When adding edges, we sometimes need make the *complete join* of two complete components. This is the operation of adding all $|V(C_1)| \cdot |V(C_2)|$ edges between the vertices in $V(C_1)$ and $V(C_2)$. After this addition the new component is a complete component.

2.3.3 Critical Cliques

We will use the concept of a critical clique K and the corresponding critical clique graph $\mathcal{K}$ as first introduced by Lin [21] and later used by Guo [15] to create a $4k$ kernel for CLUSTER EDITING.

A *critical clique* K in G is an inclusion maximal clique where all vertices in K have the same closed neighbourhood. That is, if u and v is in K, then they are *true twins*. As all vertices in a given critical clique has the same neighbours, if the vertices of two critical cliques are adjacent, we say that the critical cliques are adjacent. Using these properties we can build a *critical clique graph* $\mathcal{K} = (V_{\mathcal{K}}, E_{\mathcal{K}})$, where $V_{\mathcal{K}}$ is the set of *critical cliques* of G, and the edges in $E_{\mathcal{K}}$ are between adjacent *critical cliques*. Two critical cliques are adjacent if for all $u, v \in V(G)$, if u is a member of the critical clique K_i and v is a member of a different critical clique K_j and $\{u, v\} \in E(G)$ then in the critical clique graph we have the edge $\{K_i, K_j\} \in E_{\mathcal{K}}$.

The critical clique for a graph G can be constructed in time $\mathcal{O}(|V| + |E|)$ [16] by creating a lexicographical ordering of vertices based on their closed neighbourhoods. The critical cliques is then the sets of vertices in the same class.

We will use the notation $K(v)$ to describe the critical clique containing v, and $V(K)$ to describe the vertices contained in K. Open and closed neighbourhoods in $\mathcal{K}$ are written as $N_{\mathcal{K}}(K)$ and $N_{\mathcal{K}}[K]$ respectively. Observe that an isolated critical clique in $\mathcal{K}$ corresponds to a complete component in G.

Chapter 3

Vertex Deletion

In this chapter we will study the problem of deleting vertices to obtain a cluster graph with cluster sizes balanced by either a multiplicative factor α, or the absolute difference δ. The two problems we will study are FACTOR-α BALANCED CLUSTER VERTEX DELETION (FACTORCVD) and DIFFERENCE-δ BALANCED CLUSTER VERTEX DELETION (DIFFERENCECVD). First we will present proofs for NP-completeness of each problem, then we will give an FPT-algorithm, and lastly we will give a polynomial kernel for the problems. In creating a polynomial kernel for these problems we will use the weighted version, the case when vertices can have an integer weight. Note that the unweighted case is a special case of the weighted version where all weights are equal to 1.

For clarity we will restate the problems here as earlier stated in the introduction. Let $\alpha \in \mathbb{R}$ be a fixed constant such that $\alpha \geq 1$, and let $\delta \in \mathbb{Z}^+$ be a fixed integer such that $\delta \geq 0$.

FACTOR-α BALANCED CLUSTER VERTEX DELETION
Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$
Question: Is there a vertex set $X \in V$ such that $|X| \leq k$ and $G - X$ results in a balanced cluster graph (i.e a disjoint union of cliques) where for all components C_i, C_j in G, it holds that $\alpha \cdot |V(C_i)| \geq |V(C_j)|$?

DIFFERENCE-δ BALANCED CLUSTER VERTEX DELETION
Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$
Question: Is there a vertex set $X \in V$ such that $|X| \leq k$ and $G - X$ results in a balanced cluster graph (i.e a disjoint union of cliques) where for all components C_i, C_j in G, it holds that $\delta + |V(C_i)| \geq |V(C_j)|$?

While for the unweighted versions FACTORCVD and DIFFERENCECVD the editing operation used is deleting vertices, for the weighted problems the equivalent operation is weight decreasing. This operation decreases the weight by one if the weight of a vertex is greater than 1, and deletes the vertex if the weight is equal to 1.

Definition 3.0.1. The weight decrease operation is defined as: For a vertex $u \in V(G)$, if $w(u) > 1$: $w(u) = w(u) - 1$ else $G = G - \{u\}$

WEIGHTED FACTOR-α BALANCED CLUSTER VERTEX DELETION
Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$ and a weight function $w : V(G) \to \mathbb{N}$
Question: Is it possible to obtain a cluster graph G' from G by at most k weight decrease operations such that for all components C_i, C_j in G, it holds that $\alpha \cdot \sum_{v \in V(C_i)} w(v) \geq \sum_{v \in V(C_j)} w(v)$?

WEIGHTED DIFFERENCE-δ BALANCED CLUSTER VERTEX DELETION
Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$ and a weight function $w : V(G) \to \mathbb{N}$
Question: Is it possible to obtain a cluster graph G' from G by at most k weight decrease operations such that for all components C_i, C_j in G, it holds that $\delta + \sum_{v \in V(C_i)} w(v) \geq \sum_{v \in V(C_j)} w(v)$?

3.1 Hardness

In this section we will show that FACTORCVD and DIFFERENCECVD are NP-complete for every fixed $\alpha \geq 1$ and $\delta \geq 0$ respectively. To do this we have to show that they are members of the class NP, and that they are NP-hard. To show NP-hardness we will use a reduction from the problem VERTEX COVER, which is known to be NP-complete [13].

VERTEX COVER
Input: A graph $G = (V, E)$ and an integer k
Question: Does there exist a set $X \subseteq V(G)$ of size at most k such that $G - X$ has no edges?

Lemma 3.1.1. FACTORCVD *and* DIFFERENCECVD *are members of NP.*

Proof. To show membership in NP we will show that given a instance $I = (G, k)$ of either FACTORCVD or DIFFERENCECVD and a *certificate* $S \subseteq V(G)$, which is a set of vertices which when deleted gives a balanced cluster graph, it is possible in polynomial time to verify whether I is a YES-instance. First verify that $|S| \leq k$. Then, create the graph $G' = G - S$ and check if G' is P_3-free. Lastly, check for each pair of components C_i, C_j if the sizes are within $|C_i| \leq \alpha \cdot |C_j|$ or $|C_i| \leq \delta + |C_j|$ for FACTORCVD and DIFFERENCECVD respectively.
As all of these operations are clearly polynomial, FACTORCVD and DIFFERENCECVD are in NP. □

Lemma 3.1.2. FACTORCVD *and* DIFFERENCECVD *are NP-hard for every fixed $\alpha \geq 1$ and $\delta \geq 0$ respectively.*

Proof. We prove NP-hardness by a reduction from VERTEX COVER, and since both the reductions from VERTEX COVER to FACTORCVD and DIFFERENCECVD are almost identical, we will create a dummy variable s which will then take on different values for each problem, and by this show that the reduction works for both problems. Given an instance (G', k) of VERTEX COVER and let $s = \lfloor\alpha\rfloor - 1$ for FACTORCVD or $s = \delta$ for DIFFERENCECVD, we build a graph $G = (V, E)$:

First we construct a copy of G' to G. Then, for each vertex $u \in V(G')$ we add a complete component Q_u of size s. Note that for the special cases of when $\alpha < 2$ and $\delta = 0$, $s = 0$ so we add nothing. Each vertex of these complete components are then connected to the corresponding vertex $u \in V(G')$ by adding edges between u and each vertex $v \in Q_u$, creating cliques of size $s + 1$. Lastly we add a set I of $k + 1$ isolated vertices to G. We now have to prove that there exists a solution to FACTORCVD or DIFFERENCECVD of instance (G, k) if and only if there is a solution of VERTEX COVER of instance (G', k).

Claim 3.1.3. ($\Rightarrow$) *If the set $C \subseteq V(G')$ is a vertex cover of G' of size at most k, then C is also a solution to* FACTORCVD *and* DIFFERENCECVD *of graph G.*

Each clique of size s is either adjacent to a vertex in C, or adjacent to a vertex in $V(G') \setminus C$. In the first case, it is now obviously a component with size s. In the latter case, it is now a component of size $s + 1$, since $E(G') = \emptyset$ follows from the definition of VERTEX COVER. To check for size balance, we see that the isolated vertices in I have not been touched, so the smallest components have size 1. The largest component has size at most $s + 1$, which then for the two versions of the problems is within the bound of $s + 1 = \lfloor\alpha\rfloor \leq \alpha \cdot 1$ or $s + 1 = \delta + 1$.

Claim 3.1.4. ($\Leftarrow$) *If the set $A \subseteq V(G)$ is a inclusion minimal solution set to* FACTORCVD *or* DIFFERENCECVD *of G we will show that it can be used to obtain a solution A' to* VERTEX COVER *in G' such that $|A| = |A'|$.*

As $|I| > k$, not all vertices of I can be a member of A. This implies that there is a set $A' = A \setminus I$ which is a solution if A is a solution. Further, since for each vertex $u \in V(G')$ the graph induced by $Q_u \cup \{u\}$ is a complete graph, or if $s = 0$ it is only a single vertex u, all P_3s in G has to contain at least two members not in Q_u. If vertex $v \in Q_u$ is in A, then there is an equivalent solution $A' = A \setminus \{v\} \cup \{u\}$ where $u \in V(G') \cap N(v)$. This is because all P_3s that the vertex v is a member of goes through its neighbour u, and also all P_3 that $N(v)$ is part of goes through u. The size of the complete component containing v can increase as a result of this substitution, but as $|Q_u| = s$ and the smallest complete component is of size 1, the size constraint is satisfied.

For the case of $s = 0$, all P_3s in G are contained in G' by default, and by deleting A what is left are isolated vertices in $V(G') \cup I$. This is true because having a remaining edge in $E(G')$ would create components of size 2, not satisfying the size requirement.

After this substitution $A \subseteq V(G')$, and since an edge $\{u, v\} \in E(G')$ would induce a P_3 with members of both Q_u and Q_v, no such edges should exist. This implies that A is a vertex cover of graph G' with $|A| \leq k$.

□

Theorem 3.1.5. FACTOR-α BALANCED CLUSTER VERTEX DELETION *and* DIFFERENCE-δ BALANCED CLUSTER VERTEX DELETION *are NP-Complete for every fixed α or δ.*

Proof. As we now have proved that FACTORCVD and DIFFERENCECVD are both in NP by Lemma 3.1.1 and are NP-hard by Lemma 3.1.2 They are both NP-Complete. □

To show hardness of the weighted variations we use that the unweighted version is a special case of the weighted.

Theorem 3.1.6. WEIGHTED FACTOR-α BALANCED CLUSTER VERTEX DELETION *and* WEIGHTED DIFFERENCE-δ BALANCED CLUSTER VERTEX DELETION *are NP-Complete for every fixed α or δ.*

Proof. As we now have proved that FACTORCVD and DIFFERENCECVD are both NP-Complete, notice that they are equivalent with the weighted instance where for all $u \in V(G)$, $w(u) = 1$. This shows that that the weighted versions are also NP-complete. □

3.2 FPT-algorithm for FACTORCVD

We will now present a *branching algorithm* for WEIGHTED FACTORCVD. The algorithm uses the observation that a cluster graph is a P_3-free graph, i.e. without induced paths of length 3, by finding a P_3 and branching on deleting each of the three vertices. It repeats this until the graph is a cluster graph, and then computes the weight of all components and sorts them by increasing weight. Finding a set of components to delete to get to a balanced cluster graph is then done by finding a sequence in the list which upholds the constraints. The steps of the algorithm will be presented as a set of rules to apply until they are no longer possible to apply.

This algorithm can also easily be modified to work for DIFFERENCECVD, by in Step 4 instead of looking for an upper bound of some α times the lower bound, the upper bound is some absolute value greater than the lower bound. For this algorithm to work for the unweighted versions of DIFFERENCECVD or FACTORCVD, we use the property that an unweighted instance is a special case of a weighted instance where all weights are set to 1.

Step 1. Given an instance (G, k), find a P_3 in G, and for each of the vertices $\{u, v, w\} \in V(G)$ in the P_3, branch into the 3 instances $(G - \{u\}, k - w(u))$,$(G - \{v\}, k - w(v))$ and $(G - \{w\}, k - w(w))$. If $k < 0$ return NO.

Lemma 3.2.1. *Step 1 is safe and can be done in time $\mathcal{O}(3^k \cdot (|V| + |E|))$.*

Proof. In one direction, it is obvious that if either $(G - \{u\}, k - w(u))$,$(G - \{v\}, k - w(v))$ or $(G - \{w\}, k - w(w))$ are a YES-instance, then (G, k) is also a YES-instance.

For the other direction, to create a P_3-free graph we have to remove at least 1 vertex from each P_3. If $\{u, v, w\}$ induces a P_3, one of them has to be removed. As the algorithm tries all three possibilities, if (G, k) is a YES-instance, at least one of $(G - \{u\}, k - w(u))$,$(G - \{v\}, k - w(v))$ or $(G - \{w\}, k - w(w))$ has to be a YES-instance.

Finding an induced P_3 can be done in time $\mathcal{O}(|V| + |E|)$ by DFS. For each component in G, we run DFS, and for each visited vertex we ensure that it is in a clique with all marked vertices. If we find a vertex u such that its set of visited neighbours is smaller than the set of visited vertices in the component, we have found a P_3. Finding the vertex v which is not a neighbour of u can be done by comparing the visited vertices with the vertices adjacent to u. As we decrease k in all branches the height of the branching tree is at most k, and the total number of nodes in the tree is 3^k. The total running time of the branching step is then $\mathcal{O}(3^k \cdot (|V| + |E|))$. □

As we now know that each component is a complete graph, we can compute the total weight of each component and create a list M of the weights of all components. The total weight is the sum of the weights of all vertices in a component.

We now introduce the operation of weight decreasing in lists which is defined in Definition 3.2.1. By using this we can change the problem into finding a set of elements in M such that by at most k weight decreases each element in the list is either 0, or satisfies the property that for any two elements $M[i]$ and $M[j]$ it holds that $\alpha \cdot M[i] \geq M[j]$ for FACTORCVD or $\delta + M[i] \geq M[j]$ for DIFFERENCECVD. Observe that as each element in M corresponds to a component in G, the size of M is bounded by $|V|$.

Definition 3.2.1. The weight decrease operation for elements in a list is defined as: For a list M with elements $M[i]$ where $0 \leq i < |M|$, if $M[i] > 0$ then decreasing the weight gives $M[i] = M[i] - 1$.

Step 2. Compute the weight of each component, and add this to a list M. Sort M so that weights are in increasing order.

Lemma 3.2.2. *Step 2 can be done in time $\mathcal{O}(|V|)$.*

Proof. Computing the weights of all components can be done in time $\mathcal{O}(|V|)$. As the number of components are at most $|V|$, and the value of each element is also bounded by $|V|$, sorting by bucket sort can be done in time $\mathcal{O}(|V|)$. As all operations are linear, this gives a total running time bounded by $\mathcal{O}(|V|)$. □

As the list is now sorted, a solution will be to decrease some amount of the smallest elements down to zero, and reduce the value of some amount of the largest elements, so that the remaining elements form a consecutive sequence in the list. To make the search for this subsequence of M easier, we create the two auxiliary lists S and L, such that $S[i]$ is the sum of all elements from $M[0]$ to $M[i-1]$ and $L[i]$ is the sum of the absolute difference of $M[i]$ and each element from $M[i+1]$ to the end of the list. The element $S[i]$ can be understood as "how much has to be deleted for $M[i]$ to be the smallest element in M", and $L[i]$ is the cost of reducing all elements larger than $M[i]$ down to the value of of $M[i]$. As M is sorted in increasing order it holds that $S[i] \leq S[i+1]$ and $L[i] \geq L[i+1]$.

Step 3. Create list S and L such that $S[i] = \sum_{j=0}^{i-1} M[j]$ is the sum of all elements smaller that $M[i]$ and $L[i] = \sum_{j=i+1}^{|M|-1} M[j] - M[i]$ is the sum the difference of $M[i]$ and all elements after $M[i]$ in the list.

Lemma 3.2.3. *Step 3 can be done in time $\mathcal{O}(|V|)$.*

Proof. As $|M| \leq |V|$, creating each list S and L can be done in time $\mathcal{O}(|V|)$ as $S[i] = S[i-1] + M[i-1]$ and $L[i] = L[i+1] + (M[i+1] - M[i]) \cdot (|M| - 1 - i)$ and $S[0] = 0, L[|M| - 1] = 0$ □

Now we use a version of the sliding window technique to find a sequence which upholds the constraints. For each step $0 \leq i < |M|$ we assume that element $M[i]$ is the smallest in the sequence and everything in $M[j]$ where $j < i$ has to be deleted. To simplify reading, we use the value MAX to mean $MAX = \lfloor \alpha \cdot M[i] \rfloor$ for FACTORCVD or $MAX = \delta + M[i]$ for DIFFERENCECVD. As we do not know the length of the sequence at each step, we have to search from the last value of x' to find the new $M[x]$ where $x > x'$ such that $M[x] \leq MAX < M[x+1]$. When this value $M[x]$ is found a lookup of S and L is enough to check if this is a valid solution.

Step 4. For each value $0 \leq i < |M|$:
If $S[i] > k$, then return NO.
Find x such that $M[x] \leq MAX < M[x+1]$.
Check if $S[i] + (L[x+1] + (M[x+1] - MAX) \cdot (|M| - 1 - i)) \leq k$ and return YES if it holds.
If this completes to $i = |M| - 1$ without finding a valid solution, then return NO.

Lemma 3.2.4. *Step 4 is safe and can be done in time $\mathcal{O}(|V|)$.*

Proof. This uses the fact that a cluster graph satisfying the size constraint has to have one of the original components as its smallest element. Assume C_{min} is the smallest component. By only deleting some of its vertices, it is now a smaller component, and the budget is smaller. If the instance did not satisfy the size constraint before, it does not do that now either. Therefore, either C_{min} is the smallest component, or it has to be deleted entirely.

As the branching step of the algorithm tries all possible configurations of removing all P_3s, it will clearly always find a valid solution if there is one, or conclude that there is no solution if it has checked all configurations. As all components greater than the size constraint has to be decreased, we see that the optimal decrease is to make them all of size $\lfloor \alpha \cdot C_{min} \rfloor$ or $\delta + C_{min}$.

For each value of i we have to find the largest component which satisfies the size constraint. As this value is greater than the previous value, the pointer indicating the largest value has to only search in one direction. A lookup of S and L to find how much has to be deleted for this particular solution is done in constant time. As $i \leq |M| \leq |V|$ the total running time is $\mathcal{O}(|V|)$. $\square$

3.2.1 Running Time

Theorem 3.2.5. FACTORCVD, DIFFERENCECVD *and* WEIGHTED FACTORCVD *are solvable in time* $\mathcal{O}(3^k \cdot (|V| + |E|))$.

Proof. As the safeness of each step is shown by Lemmas 3.2.1 to 3.2.4, the only thing remaining is to show running time.

The total running time for the branching step is $\mathcal{O}(3^k \cdot (|V| + |E|))$. For each of 3^k the leaves of the branching tree we have to execute the steps 2 to 4. All of these are bounded by $\mathcal{O}(|V|)$. The total running time is therefore bounded by $\mathcal{O}(3^k \cdot (|V| + |E|))$. $\square$

3.3 Polynomial Kernel for FACTORCVD

In this chapter we will first give a kernel for the problem of WEIGHTED FACTORCVD. The kernel is split into two parts, with the first part giving a polynomial bound on the number of vertices in G, and the second part bounds the weight of each vertex by a function $g(k)$. The kernel for WEIGHTED FACTORCVD can then be used as a *bikernel* for FACTORCVD by giving each vertex in an instance of FACTORCVD a weight of 1. We will also discuss how the results from this kernel can be used to transform a reduced instance of WEIGHTED FACTORCVD back into a reduced instance of FACTORCVD.

3.3.1 Definitions

We will now give some definitions of notation and operations used in the kernel.

Definition 3.3.1. For the sum of weights of a set A of vertices we will overload the notation by giving $w()$ different meaning for application on vertices and sets of vertices. For a set A of vertices, $w(A)$ is defined as the sum of the weights of all vertices in A. That is, $w(A) = \sum_{v \in A} w(v)$. This will mostly be used in the sense of the total weight of a closed neighbourhood $w(N[v])$, as this is an upper limit on the weight of a clique where v is a member.

Definition 3.3.2. For a YES-instance (G, k), a *solution* is a triple $A = (A_1, A_2, dec)$, where $A_1 \subseteq V(G)$ is a set of weighted vertices which has to be deleted from G. $A_2 \subseteq V(G)$ is a set of weighted vertices where the weight has to be decreased and $dec(u)$ is a function giving the amount of weight decreasing needed for each vertex in A_2. The total weight of a solution $w(A) = w(A_1) + \sum_{u \in A_2} dec(u)$ is the total weight of the deleted vertices plus the sum of all weights decreased in A_2 by dec. For a solution to be valid it has to have $w(A) \leq k$ and $G - A$ is a *solved instance*. That is, $G - A$ is P_3-free and also a balanced cluster graph. When using the notation $G - A$ here we use it as a combination of the two operations $G - A_1$ and $w(u) = w(u) - dec(u)$ for all vertices $u \in A_2$.

Definition 3.3.3. In the trivial YES-instance (G, k), let $V(G) = \{u\}, E(G) = \emptyset, k = 0$.

Definition 3.3.4. In the trivial NO-instance (G, k), let $V(G) = \{u, v, w\}, E(G) = \{\{u, v\}\{v, w\}\}, k = 0$.

3.3.2 Reduction Rules

The reduction rules of this kernel takes an instance of WEIGHTED FACTORCVD as input, and outputs a reduced instance of WEIGHTED FACTORCVD. For an instance of FACTORCVD, each vertex has to be given weight of 1. The reduction rules will be applied exhaustively, by always applying the lowest numbered rule which can be applied. An exception to this is in the bounding of weights, where there are multiple rules concerned with bounding weights, but if they are already bounded we can skip to the last reduction rule.

Reduction Rule 1. If $k < 0$, then return a trivial NO-instance.

If the parameter k is smaller than $\frac{\alpha+1}{\alpha-1}$, then the instance is solvable in polynomial time. We therefore use the FPT-algorithm to solve the instance and return either YES-instance or NO-instance.

Reduction Rule 2. If $\alpha > 1$ and $k \leq \frac{\alpha+1}{\alpha-1}$, then solve the instance using the algorithm provided in section 3.2, and return either a trivial YES-instance or NO-instance.

By Theorem 3.2.5 this is safe and runs in time $\mathcal{O}(3^{\frac{\alpha+1}{\alpha-1}} \cdot (|V| + |E|))$.

We will now employ the marking procedure given by Le et al. [19]. It uses the observation that the problem of CLUSTER VERTEX DELETION has a trivial 3-approximation to find an approximated solution S of size at most $3k$. This is done by finding 3 vertices $u, v, w \in V(G)$ which induces a P_3 in G and adding them to the set S. This operation is repeated until $G - S$ is P_3-free, and we have the set S which we know is at most 3 times an optimal solution.

Reduction Rule 3. If $|S| > 3k$, then return the trivial NO-instance.

Lemma 3.3.1. *Reduction Rule 3 is safe and can be applied in time* $\mathcal{O}(|V| \cdot |E|)$.

Proof. As S is comprised of more than k P_3s, and at least one vertex of each induced P_3 has to be deleted, there can be no solution S' of weight at most k.

Finding all P_3s in G can be done in time $\mathcal{O}(|V| \cdot |E|)$ as shown by Huffner et al. [17]. This is done by for each vertex $u \in V(G)$, finding the P_3s where u has degree 2. To do this, for each pair of vertices $v, w \in N(u)$, check if there exist an edge $\{v, w\}$ in $E(G)$. As checking if an edge exist is $\mathcal{O}(1)$, and for each vertex u we check for $\mathcal{O}(deg(u)^2)$ edges, for the entire graph G this is bounded by $\mathcal{O}(|V| \cdot |E|)$. □

The marking procedure from [19] runs by for each vertex $s \in S$, marking edges in $G - S$ by adding them to the set $mark(s)$. It does this by finding an edge $\{u, v\} \in E(G - S)$ such that the vertices s, u, v induce a P_3 in G. The edge is then added to $mark(s)$. This is then repeated until either a vertex has $k + 1$ marked edges, or there is no edges left to mark for any vertex in S.

Reduction Rule 4. If there exist a vertex $s \in S$ such that $|mark(s)| > k$, then delete s and reduce k by $w(s)$. The new instance is then $(G - \{s\}, k - w(s))$.

Lemma 3.3.2. *Reduction Rule 4 is safe and can be applied in time* $\mathcal{O}(|V|(|V| + |E|))$.

Proof. In one direction it is clear that if A where $w(A) \leq k - w(s)$ is a solution to $(G - \{s\}, k - w(s))$, then $A = (A_1 \cup \{s\}, A_2, dec)$ is a solution to (G, k) of weight $w(A) \leq k$.

For the other direction, let A be a solution to (G, k). Assume that $s \notin A_1$. Then we have that one vertex from each edge in $mark(s)$ has to be a member of A_1. As $w(A) \leq k$, we see that this is not possible. This implies that $s \in A_1$ and therefore $A_1 \setminus \{s\}$ is a solution to $(G - \{s\}, k - w(s))$.

This is done by for each vertex in $s \in S$, find any vertex u with distance 2 from s such that $u \notin S$. This can be done by e.g. BFS in time $\mathcal{O}(|V| + |E|)$. As $|S| \leq |V|$ the total time is bounded by $\mathcal{O}(|V|(|V| + |E|))$. □

Reduction Rule 5. If a vertex $s \in S$ is adjacent to vertices in more than $k+1$ different cliques in $G - S$, then delete s and decrease k by $w(s)$. The new instance is then $(G - \{s\}, k - w(s))$.

Lemma 3.3.3. *Reduction Rule 5 is safe and can be applied in time $\mathcal{O}(|V|(|V| + |E|))$.*

Proof. In one direction in it clear that if A where $w(A) \leq k - w(s)$ is a solution to $(G - \{s\}, k - w(s))$, then $A = (A_1 \cup \{s\}, A_2, dec)$ is a solution of (G, k) of size $w(A) \leq k$.

For the other direction assume the opposite, that the set A is a solution to (G, k) and $s \notin A_1$. For vertices $u, v \in V(G - S)$ and $u, v \in N(s)$, such that $u \notin N(v)$, this implies that they are part of different cliques in $G - S$. We then know that s, u, v induce a P_3 in G such that at least one of them has to be deleted. If s is adjacent to at least $k + 2$ cliques and $s \notin A_1$, the vertices adjacent to s in $k + 1$ of those cliques has to be deleted, but as $w(A) \leq k$, this is not possible. We then have that $(G - \{s\}, k - w(s))$ is a YES-instance if (G, k) is a YES-instance.

This is done by for each vertex $s \in S$, run DFS in $(G - S) \cup \{s\}$. The number of branches from s in the DFS-tree is the number of cliques neighbouring s. For each vertex this is bounded by $\mathcal{O}(|V| + |E|)$, and since $|S| \leq |V|$ the total time is bounded by $\mathcal{O}(|V|(|V| + |E|))$. ☐

If two vertices $u, v \in V(G)$ have the same closed neighbourhood, i.e. they are true twins, we know that if u has to be deleted for there to be a P_3-free graph, so does v. This is because as they has the same neighbourhood, there is no vertex x such that $\{x, u, v\}$ induce a P_3, and if there are vertices y, w such that $\{y, u, w\}$ induce a P_3, then so does $\{y, v, w\}$. As the cost of deleting u and v is $w(u) + w(v)$, deleting v from G and increasing $w(u)$ by $w(v)$ keeps the cost of removing all P_3 in G the same.

Reduction Rule 6. If there exist two vertices $u, v \in V(G)$ with $N[u] = N[v]$, then remove v and set $w(u) = w(u) + w(v)$.

Lemma 3.3.4. *Reduction Rule 6 is safe and can be carried out in time $\mathcal{O}(|V| + |E|)$.*

Proof. For two vertices $u, v \in V(G)$ where $N[u] = N[v]$, there can not exist a vertex w such that u, v, w induce a P_3 in G. If there exist vertices x, y such that x, y, u induce a P_3, then so does x, y, v. We will now show that if the triple $A = (A_1, A_2, dec)$ is a solution to the instance (G, k) before application of Rule 6, then there exist solution $A' = (A'_1, A'_2, dec')$ to (G', k) after the application of Rule 6 such that $w(A) = w(A')$ and vice versa.

In one direction, if there is a vertex $u \in V(G')$ with $w(u) > 1$ obtained by applying Rule 6 on vertices $u, v \in V(G)$ such that x, y, u induce a P_3, then at least one of them has to be a member of A'_1. If this member is either x or y, then we have that $A = A'$, while if $u \in A'_1$, then in (G, k), we have the vertices u and v, such that $w(u) = w(u) + w(v)$ and $N[u] = N[v]$. This implies that $A = (A'_1 \cup \{v\}, A'_2, dec')$ with $w(A) = w(A')$. If $u \in A'_2$, we have that $dec'(u) < w(u) = w(u) + w(v)$, which gives us three different cases of A. If $dec'(u) < w(u)$, we have that $A = (A'_1, A'_2, dec')$, if $dec'(u) = w(u)$, we have that $A = (A'_1 \cup \{u\}, A'_2 \setminus \{u\}, dec')$, while if $dec'(u) > w(u)$ we have $A = (A'_1 \cup \{u\}, (A'_2 \setminus \{u\}) \cup \{v\}, dec)$ with $dec(v) = dec'(u) - w(u)$. As all three cases have $w(A) = w(A')$, this implies that if A' is a solution to (G', k'), then A is a solution to (G, k).

For the reverse direction, if vertices $u, v \in V(G)$ has the same closed neighbourhood, Rule 6 delete vertex v, and update the weight of vertex u such that $w(u) = w(u) + w(v)$. To show that $w(A) = w(A')$, we have to study each combination of u, v being members of A. If both vertices $u, v \in A_1$, then this implies that the new solution is $A' = (A_1 \setminus \{v\}, A_2, dec)$, and as $w(u) = w(u) + w(v)$ we have that $w(A) = w(A')$. If $u \in A_1$ and $v \in A_2$, we have that $w(u) + dec(v) < w(u) + w(v)$. This implies that the new solution is $A' = (A_1 \setminus \{u\}, (A_2 \setminus \{v\}) \cup \{u\}, dec')$ where $dec'(u) = w(u) + dec(v)$. If $u \in A_1$ and $v \notin A_1 \cup A_2$, we have that $w(u) < w(u) + w(v)$. This implies that we can not delete u, but rather decrease its weight by $w(u)$. By this we have $A' = (A_1 \setminus \{u\}, A_2 \cup \{u\}, dec')$ where $dec'(u) = w(u)$. If both vertices $u, v \in A_2$, then this implies that the new solution is $A' = (A_1, A_2 \setminus \{v\}, dec')$ where $dec'(u) = dec(u) + dec(v)$. If $u \in A_2$ and $v \notin A_1 \cup A_2$, this implies that $A' = A$. As we in each case above have the property of $w(A) = w(A')$ this implies that if A is a solution to (G, k), then A' is a solution to (G', k).

This can be constructed in time $\mathcal{O}(|V| + |E|)$ by creating a lexicographical ordering of vertices based on their closed neighbourhoods. The true twins are then the sets of vertices in the same class [16].

□

After application of Rule 6, a complete component in G is now reduced to an isolated vertex and is not part of any P_3, but as we may need to reduce the weight of the vertex to get a balanced cluster graph we need to store the vertex for later. Because of this, they are removed from G and stored in a separate list I_v.

Reduction Rule 7. If a vertex $u \in V(G)$ is isolated, then delete it from G and add it to I_v. We then have $G = G - \{u\}, I_v = I_v \cup \{u\}$

As we have removed all isolated vertices from G, we can now give a bound on the number of vertices in G

Claim 3.3.5. *After application of Rules 1 to 7, G has at most $9k^2 + 6k$ vertices.*

Proof. After Rule 7, G does not have any isolated vertices. Each vertex is either part of S, part of a marked edge or neither. We will now show that the size of each of these is bounded.

By Rule 3 we know that $|S| \leq 3k$. As $|mark(s)| \leq k$ for each vertex in S, the total number of vertices in marked edges is at most $2 \cdot 3k \cdot k = 6k^2$. For a vertex $u \in V(G)$ to be part of neither of these sets, all its neighbours has to be in a marked edge, or it does not induce a P_3 with any vertex in S and any of its unmarked neighbours. If two vertices do not induce a P_3 with any neighbour, they has to be true twins, which would be impossible after application of Rule 6. As each component in $G - S$ is complete, this shows that there can be at most 1 vertex in each component of $G - S$ which is not part of a marked edge. By Rule 5 each vertex in S can be connected to at most $k + 1$ components in $G - S$, and the total number of components in $G - S$ is at most $3k^2 + 3k$. Adding up all sets of vertices, we get $(3k^2 + 3k) + 3k + (6k^2) = 9k^2 + 6k$. □

If there is an edge in $\{x, y\} \in E(G)$ such that both its vertices has weight greater than k, neither of them can be deleted. This implies that all vertices which is a neighbour of one of them but not the other, has to be deleted. After application of this rule the vertices x and y has the same neighbourhood, and will therefore be merged by Rule 6.

Reduction Rule 8. If there exist and edge $\{x, y\} \in E(G)$ such that $w(x) > k$ and $w(y) > k$, then remove every vertex in $N(x) \setminus N(y)$ from G and decrease k by $w(N(x) \setminus N(y))$.

Lemma 3.3.6. *Reduction Rule 8 is safe and can be applied in time $\mathcal{O}(|V| + |E|)$.*

Proof. In one direction, if A is a solution to $((G - N(x) \setminus N(y)) \cup I_v, k - w(N(x) \setminus N(y)))$, then $A = (A_1 \cup (N(x) \setminus N(y))), A_2, dec)$ is a solution to $(G \cup I_v, k)$.

For the other direction, for a vertex $u \in (N(x) \setminus N(y))$, the vertices u, x, y induce a P_3 in G. For A to be a solution to $(G \cup I_v, k)$, at least one of u, x, y has to be a member of A_1, but as $w(A) \leq k$ and $w(x) > k$, $w(y) > k$, we can conclude that $(N(x) \setminus N(y)) \subseteq A_1$. Then $(A_1 \setminus (N(x) \setminus N(y)), A_2, dec)$ is a solution to $(G - (N(x) \setminus N(y)), k - w(N(x) \setminus N(y)))$.

The rule can be applied in time $\mathcal{O}(|V| + |E|)$ by using DFS. If for an edge $\{x, y\}$ both vertices has weight greater than k, finding vertices neighbouring only one of x, y and deleting them is also a linear operation. □

After removing all edges where both vertices has weight greater than k (*big vertex*), we know that the only way for a vertex u to have more than one *big* neighbour is for the big neighbours to not share an edge. This is because after application of Rule 8, neighbouring big vertices will become true twins and therefore be reduced by Rule 6. Because of this, they induce a P_3 together with u, and the only way of removing this P_3 is to delete u.

Reduction Rule 9. If there exist distinct vertices $x, y \in N(u)$ such that $w(x) > k$ and $w(y) > k$, then remove u and decrease k by $w(u)$.

Lemma 3.3.7. *Reduction Rule 9 is safe and can be applied in time $\mathcal{O}(|V| + |E|)$.*

Proof. In one direction, if A is a solution to $((G - \{u\}) \cup I_v, k - w(u))$ and $w(A) \leq k - w(u)$, then $(A_1 \cup \{u\}, A_2, dec)$ is a solution to $(G \cup I_v, k)$.

For the other direction, as we after application of Rule 8 know that there is no edge $\{x, y\} \in E(G)$, the vertices x, y, u induce a P_3 in G. If A is a solution of $(G \cup I_v, k)$, then at least one of x, y, u has to be members of A_1. Seeing that $w(A) \leq k$ then this shows that $x, y \notin A_1$. As A is a solution to $(G \cup I_v, k)$, $(A_1 \setminus \{u\}, A_2, dec)$ has to be a solution to $(G - \{u\}, k - w(u))$.

It can be carried out in time $\mathcal{O}(|V| + |E|)$ by for each vertex in G, iterate though the adjacency list and count the number of neighbours with weight greater than k. If for a vertex v it has more than one such neighbour, deleting v is a linear operation. □

Next we use another observation given by [19], namely that for a vertex $s \in S$ and a clique C in $G - S$ such that $N(s) \cap C \neq \emptyset$ and $C \setminus N(s) \neq \emptyset$, we have to deltete at least one of s, $(N(s) \cap C)$ or $(C \setminus N(s))$. If then both $w(N(s) \cap C) > k$ and $w(C \setminus N(s)) > k$, this implies that we have to delete s.

Reduction Rule 10. If for a vertex $s \in S$ and a clique C in $G - S$, both $w(N(s) \cap C) > k$ and $w(C \setminus N(s)) > k$, then delete s and decrease k by $w(s)$.

Lemma 3.3.8. *Reduction Rule 10 is safe.*

Proof. In one direction it is clear that if $A' = (A'_1, A'_2, dec')$ where $w(A') \leq k - w(s)$ is a solution to $((G - \{s\}) \cup I_v, k - w(s))$, then $A = (A'_1 \cup \{s\}, A'_2, dec')$ is a solution of $(G \cup I_v, k)$ of size $w(A) \leq k$.

For the other direction, let $A = (A_1, A_2, dec)$ be a solution to $(G \cup I_v, k)$, and assume that $s \notin A_1$. As $w(N(s) \cap C) > k$ and $w(C \setminus N(s)) > k$, then for all vertices $x \in (C \setminus N(s))$ and $y \in (C \cap N(s))$, x, y, s induce a P_3 in G. As $w(A) \leq k$ we see that neither $N(s) \cap C \subseteq A_1$ or $N(s) \setminus C \subseteq A_1$. By this we see that $s \in A_1$, and $A = (A_1 \setminus \{s\}, A_2, dec)$ is a soluton to $((G - \{s\}) \cup I_v, k - w(s))$.

□

If there exist a vertex $s \in S$ such that $w(s) > k$, we know that it can not be deleted. If this vertex is then neighbour with a clique C in $G - S$ such that $w(N(s) \cap C) > k$ we know that all other neighbours of s in $G - S$ has to be deleted.

Reduction Rule 11. If for a vertex $s \in S$ with weight $w(s) > k$ and a clique C in $G - S$ such that $w(N(s) \cap C) > k$, then delete $(N(s) \setminus C) \cap (V(G) \setminus S))$ and decrease k by $w((N(s) \setminus C) \cap (V(G) \setminus S))$.

Lemma 3.3.9. *Reduction Rule 11 is safe.*

Proof. In one direction, it is obvious that if $A = (A_1, A_2, dec)$ is a solution to $(G - (N(s) \setminus C) \cap (V(G) \setminus S) \cup I_v, k - w((N(s) \setminus C) \cap (V(G) \setminus S))$, then $A = (A_1 \cup (N(s) \setminus C) \cap (V(G) \setminus S), A_2, dec)$ is a solution to $(G \cup I_v, k)$.

For the other direction, assume that $A = (A_1, A_2, dec)$ is a solution to $(G - (N(s) \setminus C) \cap (V(G) \setminus S), k - w((N(s) \setminus C) \cap (V(G) \setminus S))$ and $(N(s) \setminus C) \cap (V(G) \setminus S) \cap A_1 = \emptyset$. As we know that two vertices $u, v \in V(G - S)$ which is adjacent to s but not in the same clique in $G - S$ induces a P_3 together with s, for G to become P_3-free s can be adjacent to only one clique in $G - S$. If $s \in A_1$ or $N(s) \cap C \subseteq A_1$, then $w(A) > k$. This implies that $(N(s) \setminus C) \cap (V(G) \setminus S) \subseteq A_1$ for $w(A) \leq k$.

□

Claim 3.3.10. *Reduction Rules 10 and 11 can be applied in time $\mathcal{O}(|V|(|V| + |E|))$.*

Proof. For each vertex $s \in S$, checking the weight of the neighbouring cliques in $G - S$ and also the the weights of the partitions of a clique either adjacent to s or not adjacent to s can be done in time $\mathcal{O}(|V| + |E|)$ by e.g. DFS. Eventually deleting vertices can also be done in linear time. As $|S| \leq |V|$ the total running time for the rules are bounded by $\mathcal{O}(|V|(|V| + |E|))$. □

After application of Rule 11, we can give bounds on the total weight of a clique C in G.

Claim 3.3.11. *For a clique C' in G, $w(C') \leq 6k^2 + w(u)$ where u is the vertex in C' with largest weights.*

Proof. The clique C' contains the vertices of at most one clique C in $G - S$ and all vertices in S. We also have that at most one vertex in C' has weight greater than k. As $|S| \leq 3k$, we have that $w(S \cap C') \leq 3(k - 1)k + w(s_{max})$ where s_{max} is the vertex in $S \cap C'$ of maximum weight.

By Rule 10 we know that for each vertex $s \in S$, either $w(N(s) \cap C) \leq k$ or $w(C \setminus N(s)) \leq k$. If there is a vertex $s' \in S$ such that $w(N(s) \cap C) \leq k$ and $w(C \setminus N(s)) \leq k$, we know that $w(C) \leq 2k$.

If no such vertex exist, we have the partition $S_1, S_2 \subseteq S$, where $S_1 = \{s \in S \mid w(C \cap N(s)) \leq k\}$ and $S_2 = \{s \in S \mid w(C \setminus N(s)) \leq k\}$.

Let $C_1 \subseteq C$ be the set of vertices adjacent to at least one vertex in S_1, and $C_2 \subseteq C$ be the set of vertices not adjacent to at least one vertex in S_2. Then we have $w(C_1) + w(C_2) \leq k|S_1| + k|S_2| \leq k|S| \leq 3k^2$. If $|C \setminus (C_1 \cup C_2)| > 0$, this means that there is some vertex $v \in C$ such that it is not adjacent to any vertex in S_1, and adjacent to all vertices in S_2. By Rule 6, we know that $|C \setminus (C_1 \cup C_2)| \leq 1$ and that $w(C) \leq 3k^2 + w(v)$.

We now have that $w(C') = w(S) + w(C) \leq (3(k-1)k + w(s_{max})) + (3k^2 + w(v))$. By Rules 8 and 9, we have that at most one of s_{max} and v can have weight greater than k. This gives the bound $w(C') \leq 6k^2 + w(u)$.

□

By Claim 3.3.11 we then know that if there exist vertices with weight greater than k such that they have a neighbourhood with total weight of more than $6k^2 + k$, it is a NO-instance.

Reduction Rule 12. If there exist a vertex $u \in V(G)$ such that $w(u) > k$ and $w(N(u)) > 6k^2 + k$, then return trivial NO-instance.

Lemma 3.3.12. *Reduction Rule 12 is safe and can be applied in time $\mathcal{O}(|V| + |E|)$.*

Proof. As we showed in Claim 3.3.11 that no vertex with weight bigger than k can have a neighbourhood with total weight of more than $6k^2$ in the same clique, we know that if a vertex has a neighbourhood of weight more than $6k^2 + k$, we can not decrease the weight of the neighbourhood enough. This implies that we have a NO-instance.

This can be checked in time $\mathcal{O}(|V| + |E|)$ by for each vertex with weight greater than k, compute the sum of the weights of its neighbouring vertices. □

Weight reduction

After bounding the number of vertices in G, we have to give a bound on the weight of each vertex as a polynomial of k. If no vertex in $G \cup I_v$ has weight greater than $\alpha(6k^2 + k) + k$, we already have a bound on the weights of each vertex, and can therefore skip to reduction Rule 17. If there exist vertices with weight greater than this, we have to bound these weights by substitution.

The first step in bounding the weight is done by partitioning all vertices into three sets X, Y, Z, where members of X has weight greater than k, members of Y are neighbours of a member of X, and Z is the rest of $V(G)$. See Figure 3.1. Observe that after Rule 8, the vertices in the set X form an independent set in G, and by Rule 12 a vertex $x \in X$ has $w(N(x)) \leq 6k^2 + k$. Each vertex in Y has exactly one neighbour in X, but can have many neighbours in $Y \cup Z$, while the vertices in Z only has neighbours in $Y \cup Z$. This implies that they can only be part of a clique with vertices of weight at most k. If there exist vertices $x \in X, y \in Y$ and $z \in Z$ such that $x, z \in N(y)$, this implies that x, y, z induces a P_3 in G. By this we see that no complete component in a solution can contain vertices from both Y and Z, since that implies that some vertex in X has to be deleted.

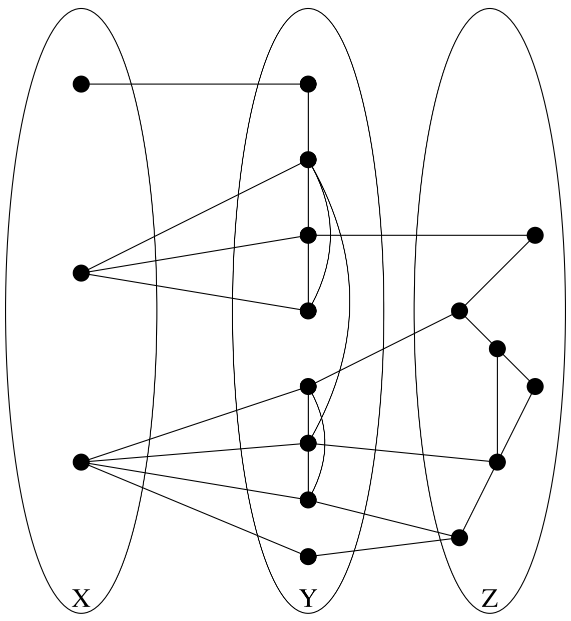

Figure 3.1: An example of the partition of a graph G into sets X, Y, Z

As we have by Claim 3.3.11 that a clique containing only vertices of weight at most k have a total weight of at most $6k^2 + k$, and we already have concluded that there exist a vertex with weight greater than $\alpha(6k^2 + k) + k$, we see that if Z is not empty, we need to delete all vertices in Z.

Reduction Rule 13. If $|Z| > 0$, then remove every vertex in Z and decrease k accordingly. The reduced instance is $((G - Z) \cup I_v, k - w(Z))$.

Lemma 3.3.13. *Reduction Rule 13 is safe and can be applied in time $\mathcal{O}(|V| + |E|)$.*

Proof. In one direction it is obvious that if $((G - Z) \cup I_v, k - w(Z))$ is a YES-instance, then $(G \cup I_v, k)$ is the same. To show that if $(G \cup I_v, k)$ has a solution then so does $(G - Z, k - w(Z))$, we assume the opposite, that there exist a solution where Z is not deleted. Observe that there is a vertex $u \in V(G)$ such that $w(u) > \alpha \cdot (6k^2 + k) + k$. Then there exist a vertex $v \in Z$ which is part of a clique with total weight greater than $6k^2 + k$. By the definition of Z no vertex in $N[v]$ has weight greater than k, so by Claim 3.3.11 we know this to not be possible. This shows that there is no solution without deleting Z.

Finding the partition of G into X, Y, Z can be done in time $\mathcal{O}(|V| + |E|)$. Then deleting the vertices in Z is also a linear operation. ☐

By the same reasoning as in Rule 13, we can delete vertices in I_v with small weights as we know there exist vertices with large weights.

Reduction Rule 14. If there exist a vertex $u \in I_v$ such that $w(u) \leq 6k^2 + k$, then delete u and decrease k by $w(u)$. The new instance is then $(G \cup I_v \setminus \{u\}, k - w(u))$.

Lemma 3.3.14. *Reduction Rule 14 is safe and can be applied in time $\mathcal{O}(|V|)$.*

Proof. This use the same observation as Rule 13. As there exist a vertex with weight greater than $\alpha \cdot (6k^2 + k) + k$, no component in a YES-instance can have weight less than $6k^2 + k$.

As the size of I_v is bounded by $|V|$, checking each for size and deleting elements which is too big is linear. □

Definition 3.3.5. We call the vertex with $w(u) > k$ which minimizes $w(N[u])$ for all vertices in $V(G) \cup I_v$ for u_{min}, and we also denote $w_{min} = w(N[u_{min}])$ as this is used repeatedly throughout the proofs.

Find the vertex u where $w(u) > k$ with smallest neighbourhood weight $w_{min} = w(N[u])$ as defined in Definition 3.3.5, and call it u_{min}. As the only operation is weight decreasing, we know that $\alpha \cdot w_{min}$ is the largest a component can be in an YES-instance.

Reduction Rule 15. If there exist a vertex $v \in V(G) \cup I_v$ such that $w(N[v]) > \alpha \cdot w_{min} + k$ then return NO-instance.

Lemma 3.3.15. *Reduction Rule 15 is safe and can be applied in time $\mathcal{O}(|V| + |E|)$.*

Proof. As w_{min} is an upper bound on the size of the smallest component in a solution, if there is a component which require more than k weight reductions to get down to a weight of αw_{min}, this implies that the instance is a NO-instance.

Finding the total weight of the neighbourhood of all vertices can be done by for each edge, adding the weights each endpoint to the neighbourhood weight of the other endpoint. By this, each edge is visited once. Then checking for vertices with too big neighbourhoods can be done in time $\mathcal{O}(|V|)$. □

Now we apply the weight substitution function. For the case of $\alpha > 1$, it uses the vertex u_{min} as defined in Def 3.3.5 as an anchor, while for the case for $\alpha = 1$ it uses the vertex x_{small}, which is the vertex in the set X of the partitioning of G with lowest weight.

For each value of $k > 1$ there is a value of $\alpha > 1$ such that there exist a size of w_{min} where the weight substitution does not work. This is the case when $w_{min} \leq \frac{\alpha+1}{\alpha-1} \cdot k$, but as we in this case already have weights bounded linearly by k, we can skip to Rule 17.

Definition 3.3.6. The weight substitution function is the function used to substitute the weights of vertices with a weight bounded by a function of k. It is separated into two functions, one for $\alpha = 1$ and one for $\alpha > 1$.

The function for $\alpha > 1$ uses u_{min} and w_{min} as defined in Definition 3.3.5. u_{min} is used as an anchor and given the new weight $\lambda = (6k^2 + \lceil \frac{\alpha k + k + 1}{\alpha - 1} \rceil + k)$, and all other vertices are shifted in relation to this vertex.

$$sub(v) = \begin{cases} w(v) & \text{if } w(v) \leq k \\ \lambda + (w(v) - w(u_{min})) & \text{if } w(v) > k \\ & \text{and } w(N[v]) \leq w_{min} + k \\ \lambda + (k+1) + (w(N(u_{min})) - w(N(v))) & \text{if } w(v) > k \\ & \text{and } w_{min} + k < w(N[v]) < \alpha \cdot w_{min} - \alpha k \\ \alpha\lambda + (w(v) - \alpha w(u_{min})) & \text{if } w(v) > k \\ & \text{and } \alpha \cdot w_{min} - \alpha k < w(N[v]) \leq \alpha \cdot w_{min} + k \end{cases}$$

For the case of $\alpha = 1$, the weight substitution is simpler. Here we use the vertex x_{small}, which is the vertex in $X \cup I_v$ with lowest weight.

$$sub(v) = \begin{cases} w(v) & \text{if } w(v) \leq k \\ (k+1) + (w(v) - w(x_{small})) & \text{if } w(v) > k \end{cases}$$

Reduction Rule 16. Apply the weight substitution function $sub(v)$ on all $v \in V(G) \cup I_v$

Lemma 3.3.16. *Reduction Rule 16 is safe and can be carried out in time $\mathcal{O}(|V|)$.*

Proof. The proof of this will be split in parts, by first showing safeness for the case when $\alpha = 1$, and then the cases for $\alpha > 1$. Note that u_{min} will be used as previously defined. For each neighbourhood $N[u]$ in $I_v \cup G$, it is either an isolated vertex in I_v, or a neighbourhood from G containing a single vertex $x \in X$, and a set $N[x] \subseteq Y$, such that $w(N[x]) \leq 6k^2 + k$ by Rule 12. Because of this, setting the anchor value to be at least $(6k^2 + \lceil \frac{\alpha k + k + 1}{\alpha - 1} \rceil + k)$ when $\alpha > 1$ guarantees that after substitution, a vertex with $w(u) > k$ get $sub(u) > k$. Since no two vertices with weight greater than k can be part of the neighbourhood of any vertex, for each neighbourhood in G only one vertex has its weight substituted.

Claim 3.3.17. *If $\alpha = 1$, weight substitution is safe with $\lambda = k + 1$ as the anchor value.*

In one direction, assume that instance $(G \cup I_v, k)$ is a YES-instance where $A = (A_1, A_2, dec)$ is a solution. That implies that $G - A$ is a balanced cluster graph where all components have the same weight. As $w(A) \leq k$ we know that no vertex in $X \cup I_v$ is a member of A_1. For a vertex $u \in (X \cup I_v)$, we know that it is part of a closed neighbourhood such that $w(N[u]) - k \leq w(C_u)$ where C_u is the clique in $G - A$ such that $u \in V(C_u)$. We now have to show that after substitution we have that $A' = A$ is a solution to $(G' \cup I'_v, k)$.

For vertex $u \in (X \cup I_v)$ we have that after substitution its new weight is $sub(u) = (k+1) + (w(u) - w(x_{small}))$, and as x_{small} is defined as the vertex in $(X \cup I_v)$ with lowest weight, this implies that $sub(u) > k$.

For each closed neighbourhood $N[u]$ we have that weight substitution is only applied once, since each vertex in Y has exactly one neighbour in X. If we first look at the vertices in A_1, we see that application of weight substitution does not affect them since their weight is at most k. If we have a vertex $v \in A_2 \cap (X \cup I_v)$, we see that $dec(v) \leq k < w(v)$. This implies that after substitution we still have that $dec(v) \leq k < sub(v)$. For the neighbourhoods $N[v]$ and $N[x_{small}]$, we have that $w(N[v]) - w(N[x_{small}]) = sub(N[v]) - sub(N[x_{small}])$, and if A is a solution to $(G \cup I_v, k)$, then $A' = A$ is a solution to $(G' \cup I'_v, k)$.

For the other direction, assume that $A' = (A'_1, A'_2, dec')$ is a solution to the instance $(G' \cup I'_v, k)$ after application of weight substitution. We will then show that this implies that $A = A'$ is a solution to the instance $(G \cup I_v, k)$ before weight substitution. As each closed neighbourhood in $G' \cup I'_v$ has had its weight changed by an equal amount, we know that if some vertices in the neighbourhood are members of $A'_1 \cup A'_2$, this implies that we also need to either remove them or decrease their weight by the same amount in the instance $(G \cup I_v, k)$. As no vertex in $X \cup I_v$ can be a member of A'_1, we know that this is also true before substitution. This implies that if A' is a solution to $(G' \cup I'_v, k)$, it is also a solution to $(G \cup I_v, k)$. □

Claim 3.3.18. *If $\alpha > 1$, weight substitution is safe with $\lambda = (6k^2 + \lceil \frac{\alpha k + k + 1}{\alpha - 1} \rceil + k)$ as the anchor value.*

Proof. First we will show that for any value of $\alpha > 1$ the difference $\alpha sub(N[u_{min}]) - sub(N[u_{min}]) > \alpha k + k$ as this is used later in the proof. As the smallest value $sub(N[u_{min}])$ can have is λ, this will be used in the proof, as a larger value will give a larger difference by properties of multiplication. By Rule 2 we have that $k > \frac{\alpha+1}{\alpha-1}$, which for instances where α is close to 1 ensures that we have a sufficiently large k. By not skipping weight substitution we also have that $w_{min} > \frac{\alpha+1}{\alpha-1} \cdot k$, as when this is not the case it is impossible to make the difference between $\alpha\lambda$ and λ big enough. We will now show that with $\lambda = (6k^2 + \lceil \frac{\alpha k + k + 1}{\alpha - 1} \rceil + k)$, after substitution the difference $\alpha\lambda - \lambda$ is greater than $\alpha k + k$ for any $\alpha > 1$ as we need this to have enough separation after substitution.

$$\begin{aligned}
\alpha\lambda - \lambda = & \\
& = (\alpha - 1)(6k^2 + \lceil \frac{\alpha k + k + 1}{\alpha - 1} \rceil + k) \\
& = (\alpha - 1)6k^2 + (\alpha - 1)\lceil \frac{\alpha k + k + 1}{\alpha - 1} \rceil + (\alpha - 1)k \\
& \text{As we know that } \lceil x \rceil \geq x \text{ we have:} \qquad (3.1) \\
& \geq (\alpha - 1)6k^2 + (\alpha - 1)\frac{\alpha k + k + 1}{\alpha - 1} + (\alpha - 1)k \\
& \text{As } (\alpha - 1) > 0 \text{ and } k \geq 0 \text{ we have that:} \\
& \geq (\alpha - 1)\frac{\alpha k + k + 1}{\alpha - 1} = \alpha k + k + 1
\end{aligned}$$

In one direction, assume that $A = (A_1, A_2, dec)$ is a solution to the instance $(G \cup I_v, k)$ such that $w(A) \leq k$. For vertices $v, u_{min} \in (X \cup I_v)$, we know that since $w(v) > k$ and $u_{min} > k$ they can not be members of A_1. Then they are either not part of A, or they are part of A_2. If $v \in A_2$ we have that $dec(v) \leq k < w(v)$, and as $sub(v) > k$ we have that after substitution $A' = A$.

If $(G \cup I_v, k)$ is a YES-instance, then this implies that for the neighbourhoods $N[v], N[u_{min}]$ in $(G \cup I_v)$, they correspond to some cliques C_v, C_{min} in $(G \cup I_v) - A$. By this we also have that $w(N[v]) - k \leq w(C_v)$ and $w(N[u_{min}]) - k \leq w(C_{min})$, as we can at most reduce the weight of a neighbourhood by k.

To show that if A is a solution to $(G \cup I_v, k)$, then it is also a solution to to $(G' \cup I'_v, k)$ after substitution, we have to study each case of the weight substitution function. The three different cases for $N[v]$ are:

- **(i)** $w(N[v]) - w_{min} \leq k$.
  The weight of $N[v]$ is within a distance of k to the weight of w_{min}
- **(ii)** $w_{min} + k < w(N[v]) < \alpha w_{min} - \alpha k$.
  The weight of $N[v]$ is more than k bigger than w_{min}, and also more than k smaller than αw_{min}.
- **(iii)** $\alpha w_{min} - k < w(N[v]) < \alpha w_{min} + k$.
  The weight of $N[v]$ is within a difference of k smaller or bigger than αw_{min}.

We will now show that weight substitution is safe for each of the three cases separately.

Case (i):

As we know by Equation 3.1 that $\alpha w_{min} - w_{min} > \alpha k + k$, $\;w_{min} - k \leq w(C_{min}) \leq w_{min}$ and $w(A) \leq k$, we know that if $w(N[v]) - w_{min} \leq k$ then $w(C_v) \leq w(C_{min}) + 2k$. This implies that it is impossible for $w(C_v)$ to become larger than $\alpha \cdot w(C_{min})$. We now have to show that this is also true after substitution, and that if $v \in A_2$, then this implies that $v \in A'_2$.

If $v \in A_2$, we know that $dec(v) \leq k < w(v)$. After substitution we have that $sub(v) = \lambda + (w(v) - w(u_{min}))$. As we know that $w(N(v))$ and $w(N(u_{min}))$ are at most $6k^2 + k$ by Rule 12, and that $w(N[u_{min}]) \leq w(N[v])$ we have that $w(u_{min}) + w(N(u_{min})) \leq w(v) + w(N(v))$. This implies that the difference $w(u_{min}) - w(v) \leq 6k^2 + k$. By this we have that the smallest weight $sub(v)$ can get is $sub(v) = \lceil \frac{\alpha k + k + 1}{\alpha - 1} \rceil$. As this is greater than k it holds that $dec(v) \leq k < sub(v)$.

To show that it is impossible for $sub(C_v)$ to become larger than $\alpha \cdot sub(C_{min})$, we have that $sub(C_{min}) \geq sub(u_{min}) + w(N(u_{min})) - k$, and we also have that $\alpha \cdot sub(N[u_{min}]) - sub(N[u_{min}]) > \alpha k + k$ for any α. As we know that weight substitution changes the weight of $N[v]$ and w_{min} by an equal amount, if it was true that $w(C_v)$ could not become bigger than $\alpha w(C_{min})$, then it is also true that $sub(C_v)$ can not become bigger than $\alpha \cdot sub(C_{min})$ by decreasing the weight of $sub(u_{min}) + w(N(u_{min}))$ by at most k.

Case (ii):

Before weight substitution the closed neighbourhood $N[v]$ had a weight such that by decreasing the weight by at most k it could not become smaller than w_{min}, and that by decreasing the weight of w_{min} by at most k, $w(N[v])$ could not become larger than $\alpha w(C_{min})$. To show that this is also true after substitution, we see that $sub(v) = \lambda + (k+1) + (w(N(u_{min})) - w(N(v)))$. As we also have that $sub(u_{min}) = \lambda$, we see that $sub(N[v]) - sub(N[u_{min}]) = k + 1$. This means that by doing at most k weight decreases, $w(C_v)$ can not become smaller than $w(C_{min})$, and as $\alpha \cdot sub(N[u_{min}]) - sub(N[u_{min}]) > \alpha k + k$, by having $sub(C_{min}) = sub(N[u_{min}]) - k$ we still have that $\alpha \cdot sub(C_{min}) \geq$

$sub(C_v)$.

Case (iii):

As we know that $w(N[v])$ can not become smaller than w_{min} by the fact that members of this case have a larger weight than in **Case (ii)**, what we need to show is that if $w(C_v) \leq \alpha w(C_{min})$, then it implies that $sub(C_v) \leq \alpha sub(C_{min})$. As $sub(C_{min}) \geq sub(N[u_{min}]) - k$, we have that $\alpha sub(C_{min}) \geq \alpha sub(N[u_{min}]) - \alpha k$. This gives us a lower bound of the upper size bound at $\alpha(\lambda + w(N(u_{min}))) - \alpha k$.

For neighbourhoods in this range, their weight are substituted with $sub(v) = \alpha\lambda + (w(v) - \alpha w(u_{min}))$, which gives a new weight such that $\alpha sub(N[u_{min}]) - sub(N[v]) = \alpha w(N[u_{min}]) - w(N[v])$. If then A decreases the weight of w_{min} by an amount $c \leq k$ before substitution, the weight of $sub(N[u_{min}])$ is also decreased by c after substitution. This lowers the upper bound by $\alpha \cdot c$ both before and after substitution. As this change is equal, and the difference between the upper bound and $sub(N[v])$ is the same, if A was a solution to $(G \cup I_v, k)$ before substitution, it is also a solution to $(G' \cup I'_v, k)$ after substitution.

For the other direction, assume that $A' = (A'_1, A'_2, dec')$ is a solution to $(G' \cup I'_v, k)$ such that $w(A') \leq k$. Then we know that no vertex where weight substitution was applied can be a member of A'_1, and if a vertex in A'_2 was changed by weight substitution, we know that its weight is still larger than k. This implies that $dec'(u) = dec(u)$ for all vertices $u \in A_2$.

To show that if $sub(C'_v) \leq \alpha sub(C'_{min})$ then this implies that $w(C'_u) \leq \alpha w(C'_{min})$ before weight substitution, we need to do a study of the three different cases of the weight substitution function.

- **(i)** $sub(N[v]) - sub(N[u_{min}]) \leq k$.
  The weight of $N[v]$ is within a distance of k to the weight of $N[u_{min}]$
- **(ii)** $sub(N[u_{min}]) + k < w(N[v]) < \alpha sub(N[u_{min}]) - \alpha k$.
  The weight of $N[v]$ is more than k bigger than $sub(N[u_{min}])$, and also more than k smaller than $\alpha sub(N[u_{min}])$.
- **(iii)** $\alpha sub(N[u_{min}]) - k < w(N[v]) < \alpha sub(N[u_{min}]) + k$.
  The weight of $N[v]$ is within a difference of k smaller or bigger than $\alpha sub(N[u_{min}])$.

Case (i): If the weight of neighbourhood $N[v]$ is within a distance of k to the weight of $N[u_{min}]$ after substitution, we have that the same had to be true before substitution. This is because in the previous case analysis we showed that this was not possible for the two other cases. As we know that A' is a solution to $(G' \cup I'_v, k)$, and we know that $\alpha w_{min} - w_{min} > \alpha k + k$ before substitution, and $\alpha\lambda - \lambda > \alpha k + k$ after substitution. This implies that as by decreasing the weight of either $sub(N[v])$ or $sub(N[u_{min}])$ by at most k, neither $sub(C_v)$ after substitution or $w(C_v)$ before substitution can not become too big.

Case (ii): By the definition of the weight substitution function, $sub(N[v])$ has a weight of $sub(N[u_{min}]) + k + 1$. The distance between $\alpha w(N[u_{min}]) - w(N[u_{min}])$ both before and after substitution are greater than $\alpha k + k$, and $w(N[v])$ and $sub(N[v])$ are more than αk smaller than $\alpha w(N[u_{min}])$ and at least $k+1$ larger than $w(N[u_{min}])$. This implies that if A' if a solution to $(G' \cup I'_v, k)$ after substitution, then it is a solution to $(G \cup I_v, k)$ before substitution.

Case (iii): As we know that A' is a solution to $(G' \cup I'_v, k)$, and that $sub(N[v]) - \alpha sub(N[u_{min}]) = w(N[v]) - \alpha w(N[u_{min}])$ by the definition of the weight substitution function, this implies that if $(G' \cup I'_v) - A'$ creates cliques C'_v and C'_{min} such that $w(C'_v) \leq \alpha w(C'_{min}$, the same is also true for the instance

$(G \cup I_v, k)$ before substitution. □

Reduction Rule 17. Sort I_v with respect to increasing weights. Delete all elements excluding the $k+1$ with largest weights and the x smallest such that $\sum_{i=1}^{i=x} w(u_i) > k$.

Lemma 3.3.19. *Reduction Rule 17 is safe and can be applied in time $\mathcal{O}(|V|)$.*

Proof. For this to be safe, no solution have to edit the weights of the deleted elements. As the reduction stores up to $k+1$ of the smallest members of I_v in such a way that the total weight of the small elements is greater than k. This ensures than not all small elements can be deleted, and we know that at least one of them has to be either the smallest component in $G \cup I_v$ or bigger than the smallest component.

If there are some component too big for the size constraint, there can be at most k of them for $(G \cup I_v, k)$ to be a YES-instance. So deleting all but the $k+1$ largest will not change the instance in any way.

As the number of elements in I_v is bounded by $|V|$, computing their sizes and sorting them by bucket sort are both operations bounded by $\mathcal{O}(|V|)$. □

3.3.3 Analysis

Theorem 3.3.20. WEIGHTED FACTORCVD *has a polynomial kernel with $\mathcal{O}(k^2)$ vertices, each with weight of at most $\mathcal{O}(k^2)$ that can be constructed in $\mathcal{O}(|V|^2(|V| + |E|))$ time. This kernel is also a bikernel for* FACTORCVD.

Proof. After exhaustive application of the reduction rules, the number of vertices in $V(G)$ is at most $9k^2 + 6k$ as shown in Claim 3.3.5, and I_v has at most $2k+2$ by Lemma 17. This gives a total number of vertices as $9k^2 + 8k + 2$.

The bound on the weights of each vertex is either bounded by $\alpha(6k^2+k)+k$, $\alpha(6k^2+\lceil\frac{2k+1}{\alpha-1}\rceil+k)+k$ or $\alpha \cdot \frac{\alpha+1}{\alpha-1} \cdot k$. Where the first two are $\mathcal{O}(k^2)$ and the last is $\mathcal{O}(k)$. The safeness of this reduction is shown in the proofs for each reduction rule.

For use as a bikernel for an instance $(G \cup I_v, k)$ of FACTORCVD, give each vertex in $V(G)$ a weight of 1. As this is clearly a polynomial time operation, it is a bikernel.

To get the bound on the running time, observe that each reduction rule has a running time bounded by at most $\mathcal{O}(|V|(|V| + |E|))$. As the reduction rules are applied exhaustively they are bounded by the number of vertices deleted. As they can at most be applied $|V|$ times, we have a total bound on the running time of $\mathcal{O}(|V|^2(|V| + |E|))$. □

Corollary 3.3.20.1. FACTORCVD *has a kernel with $\mathcal{O}(k^4)$ vertices.*

Proof. To transform an instance (G', k) of WEIGHTED FACTORCVD into an instance (G, k) of FACTORCVD, for each vertex $u' \in V(G')$ with weight $w(u) = q$, construct a clique of vertices $\{u_1, u_2 ... u_q\}$ in G. Then add edges $\{u_x, v_y\}$ for each u_x and v_y where $\{u', v'\} \in E(G')$.

As G' has a quadratic number of vertices with quadratic weights, the total number of vertices in G is bounded by $\mathcal{O}(k^4)$. □

Theorem 3.3.21. FACTOR-α BALANCED CLUSTER VERTEX DELETION *has a FPT-algorithm with running time* $\mathcal{O}(3^k k^4 + |V|^2(|V| + |E|))$

Proof. By transforming the instance of FACTORCVD into an equivalent weighted instance we can bound the number of vertices to $\mathcal{O}(k^2)$ and by this the number of edges to $\mathcal{O}(k^4)$ in time $\mathcal{O}(|V|^2(|V| + |E|))$ by Theorem 3.3.20. Then using the algorithm from Theorem 3.2.5 running in time $\mathcal{O}(3^k \cdot (|V| + |E|))$ on the reduced instance we get a total running time of $\mathcal{O}(3^k k^4 + |V|^2(|V| + |E|))$. □

3.4 Polynomial Kernel for DIFFERENCECVD

As the problem of DIFFERENCECVD is quite similar to FACTORCVD except for how the sizes of each component is compared, most of the reduction rules for a kernelization algorithm can be reused. For the task of bounding the number of vertices, we can use the exact same procedure, while for bounding the weights the procedure is simplified. This is because the property of absolute difference allows for a simpler weight substitution. As the difference between two weights stays constant by shifting all weights by a constant, each component has the same size relative to each other before and after weight substitution. Since the reduction rules are almost the exact same as for FACTORCVD, we refer to the statements of the corresponding rule in FACTORCVD for a proof of safeness.

3.4.1 Reduction Rules

As for FACTORCVD, we start by checking if there exist any vertex with weight larger than $(6k^2 + k) + k + \delta$. If no such vertex exist, we already have a bound on the weights of all vertices, and therefore we can skip to rule 5. If this is not the case we partition the vertices into three sets X, Y, Z, where X is the vertices with weight greater than k, Y is the neighbours of vertices in X, and Z is the set of vertices with weight at most k and no neighbour with weight greater than k. As we have established the existence of vertices with weights greater than $(6k^2 + k) + k + \delta$, by Claim 3.3.11 we have that no vertex in Z can be part of a clique with total weight larger than $(6k^2 + k)$. This implies that for there to exist a solution, all vertices in Z has to be deleted.

Reduction Rule 1. If $|Z| > 0$, remove every vertex in Z and decrease k accordingly. The reduced instance is $((G - Z) \cup I_v, k - w(Z))$.

By the same reasoning as in rule 1, we know that if there exist vertices with weight greater than $(6k^2 + k) + k + \delta$ in G, then if there exist vertices with weight smaller than $(6k^2 + k)$ in I_v, they has to be deleted.

Reduction Rule 2. If there exist a vertex $u \in I_v$ such that $w(u) \leq 6k^2 + k$, then delete u and decrease k by $w(u)$. The new instance is then $(G \cup I_v \setminus \{u\}, k - w(u))$.

Find the vertex u where $w(u) > k$ with smallest neighbourhood weight $w(N[u])$, and call it u_{min}. As the only operation is weight decreasing, we know that $\delta + w(N[u_{min}])$ is the largest weight a component can have in an YES-instance. If there exists neighbourhoods with weight more than k bigger than this upper bound, we can safely conclude that it is a NO-instance.

Reduction Rule 3. If there exist a vertex $v \in V(G) \cup I_v$ such that $w(N[v]) > \delta + w(N[u_{min}]) + k$ then return NO-instance.

The weight substitution is done by first finding the vertex $x_{small} \in X \cup I_v$, which is the vertex in $X \cup I_v$ with minimal weight. The weight of this vertex is then set to $k + 1$, and then the weight of all other vertices are set such that $w(v) - w(x_{small}) = sub(v) - sub(x_{small})$.

Reduction Rule 4. For each vertex $v \in V(G) \cup I_v$ such that $w(v) > k$. Replace the weight with $x_{small} = (k + 1) + (w(v) - w(x_{small}))$.

Lemma 3.4.1. *Reduction Rule 4 is safe.*

Proof. In one direction, assume that instance $(G \cup I_v, k)$ is a YES-instance where $A = (A_1, A_2, dec)$ is a solution. That implies that $G - A$ is a balanced cluster graph where all components have weights differing by at most δ. As $w(A) \leq k$ we know that no vertex in $X \cup I_v$ is a member of A_1. For a vertex $u \in (X \cup I_v)$, we know that it is part of a closed neighbourhood such that $w(N[u]) - k \leq w(C_u)$ where C_u is the clique in $G - A$ such that $u \in V(C_u)$. We now have to show that after substitution we have that $A' = A$ is a solution to $(G' \cup I'_v, k)$.

If vertex u with $w(u) > k$, we have that after substitution its new weight is $sub(u) = (k + 1) + (w(u) - w(x_{small}))$, and as $(w(u) - w(x_{small}) \geq 0$ by definition of x_{small}, this implies that $sub(u) > k$. If $u \in A_2$, we see that $dec(u) \leq k < sub(u)$, which implies that this is safe after substitution.

For each closed neighbourhood $N[u]$ we have that weight substitution is only applied once, since each vertex in Y has exactly one neighbour in X. As we know that any vertex in A_1 has to have weight at most k, they are not affected by weight substitution. For the neighbourhoods $N[u]$ and $N[x_{small}]$, we have that $w(N[u]) - w(N[x_{small}]) = sub(N[u]) - sub(N[x_{small}])$, and if A is a solution to $(G \cup I_v, k)$, then $A' = A$ is a solution to $(G' \cup I'_v, k)$.

For the other direction, assume that $A' = (A'_1, A'_2, dec')$ is a solution to the instance $(G' \cup I'_v, k)$ after application of weight substitution. We will then show that this implies that $A = A'$ is a solution to the instance $(G \cup I_v, k)$ before weight substitution. As each closed neighbourhood in $G' \cup I'_v$ has had its weight changed by an equal amount, we know that if some vertices in the neighbourhood are members of $A'_1 \cup A'_2$, this implies that we also need to either remove them or decrease their weight by the same amount in the instance $(G \cup I_v, k)$. As no vertex in $X \cup I_v$ can be a member of A'_1, we know that this is also true before substitution. This implies that if A' is a solution to $(G' \cup I'_v, k)$, it is also a solution to $(G \cup I_v, k)$.

□

Reduction Rule 5. Sort I_v with respect to increasing weights. Delete all elements excluding the $k + 1$ with largest weights and the x smallest such that $\sum_{i=1}^{i=x} w(u_i) > k$.

3.4.2 Analysis

As for the kernelization for FACTORCVD, the number of vertices in the kernel for DIFFERENCECVD is bounded by $V(G) \leq 9k^2 + 6k$, $V(I_v) \leq 2k + 2$. Which gives a total of $9k^2 + 8k + 2$ vertices, with each vertex having a weight of at most $(k + 1) + \delta + k$. This gives a kernel with $\mathcal{O}(k^2)$ vertices and weight bounded by $\mathcal{O}(k)$.

Theorem 3.4.2. WEIGHTED DIFFERENCECVD *has a polynomial kernel with* $\mathcal{O}(k^2)$ *vertices, each with weight of at most* $\mathcal{O}(k^2)$ *that can be constructed in* $\mathcal{O}(|V|^2(|V|+|E|))$ *time. This kernel is also a bikernel for* DIFFERENCECVD.

Proof. As the kernel uses the same rules to bound the number of vertices as the kernel for WEIGHTED FACTORCVD, we showed in Theorem 3.3.20 that the total number of vertices is at most $9k^2 + 8k + 2$.

The weights of each vertex is either bounded by $(6k^2 + k) + k + \delta$, which is the case when we do not delete the vertices in Z, or they are bounded by $6k^2 + 2k + 1 + \delta$ by weight substitution.

The safeness and running time of each rule was showed in Theorem 3.3.20, except for Rule 4 which we showed in Lemma 3.4.1. This gives us a running time bounded by at most $\mathcal{O}(|V|^2(|V| + |E|))$. □

Corollary 3.4.2.1. DIFFERENCECVD *has a kernel with* $\mathcal{O}(k^4)$ *vertices.*

Proof. To transform an instance (G', k) of WEIGHTED DIFFERENCECVD into an instance (G, k) of DIFFERENCECVD, for each vertex $u' \in V(G')$ with weight $w(u) = q$, construct a clique of vertices $\{u_1, u_2...u_q\}$ in G. Then add edges $\{u_x, v_y\}$ for each u_x and v_y where $\{u', v'\} \in E(G')$.

As G' has a quadratic number of vertices with quadratic weights, the total number of vertices in G is bounded by $\mathcal{O}(k^4)$. □

Theorem 3.4.3. DIFFERENCE-δ BALANCED CLUSTER VERTEX DELETION *has a FPT-algorithm with running time* $\mathcal{O}(3^k k^4 + |V|^2(|V| + |E|))$

Proof. By transforming the instance of DIFFERENCECVD into an equivalent weighted instance we can bound the number of vertices to $\mathcal{O}(k^2)$ and by this the number of edges to $\mathcal{O}(k^4)$ in time $\mathcal{O}(|V|^2(|V| + |E|))$ by Theorem 3.4.2. Then using the algorithm from Theorem 3.2.5 running in time $\mathcal{O}(3^k \cdot (|V|+|E|))$ on the reduced instance we get a total running time of $\mathcal{O}(3^k k^4 + |V|^2(|V| + |E|))$. □

Chapter 4

Edge Modification

In this chapter we will look at different ways of obtaining a cluster graph by edge modification. The three different edge modification versions we will study are *Edge Addition*, *Edge Deletion*, and *Edge Editing*. We will study both the FACTOR-α BALANCED version and the DIFFERENCE-δ BALANCED version, giving a total of 6 problems restated below.

First we will restate the different problems as stated in the introduction, before we give proof for the NP-Completeness of each of them. Then we will present a kernel for the each of the FACTOR-α BALANCED version with a section showing how to use the kernel for the DIFFERENCE-δ BALANCED version.

4.1 Problem Statement

For all problems, let $\alpha \in \mathbb{R}$ be a fixed constant such that $\alpha \geq 1$, and let $\delta \in \mathbb{Z}^+$ be a fixed integer such that $\delta \geq 0$.

For the two EDGE ADDITION problems FACTOR-α BALANCED CLUSTER COMPLETION (FACTORCC) and DIFFERENCE-δ BALANCED CLUSTER COMPLETION (DIFFERENCECC) the graph modification we do is adding edges between vertices which was not already adjacent to create *complete components*.

FACTOR-α BALANCED CLUSTER COMPLETION
Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$
Question: Is there a set $A \subseteq \binom{V(G)}{2} \setminus E(G)$ such that $|A| \leq k$ and the graph $G + A$ is a cluster graph (i.e a disjoint union of cliques) where for all components C_i, C_j in G, it holds that $\alpha \cdot |V(C_i)| \geq |V(C_j)|$?

DIFFERENCE-δ BALANCED CLUSTER COMPLETION
Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$
Question: Is there a set $A \subseteq \binom{V(G)}{2} \setminus E(G)$ such that $|A| \leq k$ and the graph $G + A$ is a cluster graph where for all components C_i, C_j in G, it holds that $\delta + |V(C_i)| \geq |V(C_j)|$?

In the EDGE DELETION problems FACTOR-α BALANCED CLUSTER DELETION (FACTORCD) and DIFFERENCE-δ BALANCED CLUSTER DELETION (DIFFERENCECD), we ask whether there exist a set of edges in the graph which when deleted gives a balanced cluster graph.

FACTOR-α BALANCED CLUSTER DELETION
Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$
Question: Is there a set $A \subseteq E(G)$ such that $|A| \leq k$ and the graph $G - A$ is a cluster graph (i.e a disjoint union of cliques) where for all components C_i, C_j in G, it holds that $\alpha \cdot |V(C_i)| \geq |V(C_j)|$?

DIFFERENCE-δ BALANCED CLUSTER DELETION
Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$
Question: Is there a set $A \subseteq E(G)$ such that $|A| \leq k$ and the graph $G - A$ is a cluster graph where for all components C_i, C_j in G, it holds that $\delta + |V(C_i)| \geq |V(C_j)|$?

The EDGE EDITING problems FACTOR-α BALANCED CLUSTER EDITING (FACTORCE) and DIFFERENCE-δ BALANCED CLUSTER COMPLETION (DIFFERENCECE) ask whether it is possible to create a balanced cluster graph when we have the option of both adding and deleting edges.

FACTOR-α BALANCED CLUSTER EDITING
Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$
Question: Is there a set $A \subseteq \binom{V(G)}{2}$ such that $|A| \leq k$ and the graph $G \triangle A$ is a cluster graph where for all components C_i, C_j in G, it holds that $\alpha \cdot |V(C_i)| \geq |V(C_j)|$?

DIFFERENCE-δ BALANCED CLUSTER EDITING
Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$
Question: Is there a set $A \subseteq \binom{V(G)}{2}$ such that $|A| \leq k$ and the graph $G \triangle A$ is a cluster graph where for all components C_i, C_j in G, it holds that $\delta + |V(C_i)| \geq |V(C_j)|$?

4.2 Hardness

We will now show the hardness of the three different edge modification variations EDGE ADDITION, EDGE DELETION and EDGE EDITING. The hardness proofs of editing and deletion will be done together, while edge addition will be done separately. The proof of membership in NP will be done as one for all variations.

Lemma 4.2.1. FACTORCC, FACTORCD, FACTORCE, DIFFERENCECC, DIFFERENCECD *and* DIFFERENCECE *is in NP for all fixed $\alpha \geq 1$ or $\delta \geq 0$, respectively.*

Proof. To show membership in NP we have to show that given an instance (G, k) and a *certificate* $A \subseteq \binom{V(G)}{2}$ of edges to modify to get to a balanced cluster graph, it can be checked in polynomial time whether $G - A$ is a balanced cluster graph. Note that for EDGE ADDITION the set A has to be a subset of $\binom{V(G)}{2} \setminus E(G)$, and for EDGE DELETION A has to be a subset of $\binom{V(G)}{2} \cap E(G)$.

Given a *certificate* A to an instance (G, k), first check if $|A| \leq k$. Then construct the solved instance $G' = (V, E \triangle A)$ and check if G' is P_3-free. Finally, compute the sizes of each component to check if they all are within $\alpha \cdot |C_i| \geq |C_j|$ or $\delta + |C_i| \geq |C_j|$. All these operations are clearly polynomial, so the verifier is then polynomial. This implies that the problems are members of the class NP. □

4.2.1 Editing and Deletion

The proofs of the hardness of EDGE EDITING and EDGE DELETION will be split into two parts, where we use one reduction for the cases when $\alpha = 1$ and $\delta \leq 2$ and a separate reduction for $\alpha > 1$ and $\delta > 2$. For the case of inequality ($\alpha > 1, \delta > 2$) we will use a reduction from the NP-Complete problem EXACT 3-COVER [13], a variation of EXACT COVER. The reduction follows the lines of the reduction by Shamir et al.[24].

EXACT 3-COVER

Input: A collection C of triplets of elements in universe $U = \{u_1, u_2, ..., u_{3n}\}$, where each element in U is a member of at most 3 triplets in C

Question: Is there a set $I \subseteq C$ of size n which covers U.

Lemma 4.2.2. FACTORCE*,* DIFFERENCECE*,* FACTORCD *and* DIFFERENCECD *are NP-hard for every fixed* $\alpha > 1$ *or* $\delta > 2$.

Proof. In this proof we will give a polynomial reduction from EXACT 3-COVER. For the proof to work for both the case of $\alpha > 1$ and $\delta > 2$ we will create two dummy variables q and c which will be different for each problem. To show that the reduction works for both editing and deletion we will do the reduction for editing, but show that no optimal solution requires any edge additions, and therefore is equivalent to a deletion problem. Let $n = \frac{|U|}{3}$, for $\alpha > 1$ let $q = n + \lceil \frac{3}{\alpha - 1} \rceil$ and $c = \alpha q - 3$, while for $\delta > 2$ let $q = n + 3$ and $c = \delta + q - 3$. Let $k = n \cdot 6c + 12$ be the budget.

We construct the graph G by this procedure: First, we construct a set of vertices V_1 such that $|V_1| = |U|$ and each vertex in V_1 corresponds to an element in U. We then for each triple in the set C construct a complete component of c vertices in G corresponding to that triple, and for each vertex in the complete component we connect it to each of the vertices in V_1 corresponding to members of the triple, and also add edges so that those vertices in V_1 induce a triangle. Finally we construct n complete components of size q.

Claim 4.2.3. $(\Rightarrow)$ *If the set* $I \subseteq C$ *is a cover of* U *where* $|I| = n$*, there is a corresponding partitioning of* G *such that by making each partition a complete component,* G *becomes a balanced cluster graph by at most* k *edge editions.*

If I covers U, then for each triple in I, find the corresponding component K in G. Then for each vertex in $V_1 \cap K$, remove all edges between vertices in $V_1 \cap K$ and vertices in $V(G) \setminus (V_1 \cap K)$, which for each vertex is at most $2c + 4$. After this deletion each component is now a complete graph, and we have done at most $3n(2c + 4) = n(6c + 12) = k$ deletions. Observe that we now have components of 3 different sizes, the n components of size q, the components corresponding to I of size $c + 3$, and the components corresponding to $C \setminus I$ of size c. As $c+3 = \alpha \cdot q$ or $c+3 = \delta + q$ this is within the bound for both problems and $q \leq c < c+3$, this is a balanced cluster graph.

Claim 4.2.4. ($\Leftarrow$) *If the set $A \subseteq \binom{V(G)}{2}$ is a minimum editing set such that $|A| \leq k$ and $G = (V, E \triangle A)$ is a balanced cluster graph, we will show that it can be used to obtain a solution $I \subseteq C$ to* EXACT 3-COVER *such that $|I| = n$.*

For this proof we need to show that an optimal solution consists of components of size $c+3$ or c, where each component of size $c+3$ has exactly 3 vertices from V_1, and that is solution can be used to obtain a solution $I \subseteq C$ to the problem of EXACT 3-COVER for universe U.

The cost of creating a complete component of size $c+3$ in G is at most $6c+12$ assuming each of the three vertices from V_1 is part of 3 triplets, as for a vertex to be removed from a triplet we have to do c deletions $+2$ deletion to the other vertices from V_1 also in that triplet. To show that an optimal solution creates components of this size, we have to show that the cost of making components of size $c+1$ and $c+2$, or of size $c+2$ and $c+4$ is greater than k.

To split the three vertices from a triplet such that they create complete components of size $c+1$ and $c+2$, the deletion cost for the two vertices in $c+2$ is same as before ($2(c+2)$ for each vertex), while for the vertex in the component of size $c+1$, it has to delete all edges to vertices in V_1 and to 2 of the at most 3 cliques representing triplets for a total cost of $6+2c$. The total cost for this configuration is then $(6+2c) + 2(2c+4) = 6c+14$ which is 2 more than what we claimed to be an optimal solution.

To make complete components of size $c+4$ and $c+2$, the cost for the two vertices in $c+2$ is same as before ($2c+4$ for each). For the vertex in the component of size $c+4$, we have to add this vertex to a triple it is not part of already (assuming the other 3 vertices corresponds to the triple). To do this we have to delete its edges to all triplets it is a member of and vertices in V_1 not in the same component, at a cost of at least $3c$. It also has to add c edges to the c vertices it is in the same component as. The total editing cost is then $(4c) + 2(2c+4) = 8c+8$, $2c-4$ more than optimal.

In addition to this, $c+4$ is larger than $\alpha \cdot q$ or $\delta + q$, so we have to join each component of size q with either another component of same size at a total cost of $\frac{n}{2} \cdot q^2$, or with a component of size c at a cost of $n \cdot c \cdot q$. Both these operations comes with a cost in the order of n^3 which is greater than k.

To obtain the solution to EXACT 3-COVER, for each component of size $c+3$, the triplet in C corresponding to that component is a member of the solution I. We have now showed that if there exist a YES solution of size at most k, it corresponds to a solution of EXACT 3-COVER of size n. As we use only deletions in this reduction, and showed that having the option to add edges does not change the solution, the reduction is applicable to both EDGE EDITING and EDGE DELETION.

□

For the equality versions $\alpha = 1$ and $\delta \leq 2$ case we give a separate proof. This proof will be from the problem RESTRICTED PARTITIONING INTO TRIANGLES(RPIT), a variation of PARTITIONING INTO TRIANGLES where the input graph is a 4-regular graph with restricted neighbourhoods. Rooij et al.[22] showed that PARTITIONING INTO TRIANGLES is NP-complete for 4-regular graphs, and that this persists even for the case where G is a graph such that there is no clique of size 4. This reduction is based on a reduction by Komusiewicz[18].

RESTRICTED PARTITIONING INTO TRIANGLES

Input: An undirected 4-regular graph $G = (V, E)$ with neighbourhoods resticted such that there is no cliques of size 4 in G.

Question: Can V be partitioned into $|V|/3$ sets such that each set induce a triangle.

Lemma 4.2.5. FACTORCE, DIFFERENCECE, FACTORCD *and* DIFFERENCECD *are NP-hard when* $\alpha = 1$ *or* $\delta \leq 2$*.*

Proof. In this proof we will give a polynomial reduction from an instance $G' = (V', E')$ of RPIT. Let $n = |V'|$ and as budget we set $k = n$.

We construct the graph G by first constructing a copy of G', and then adding n isolated triangles. For the case of $\delta = 2$ or $\delta = 1$, we instead add $3n$ isolated components of size 1 or 2 respectively. We will now show that there exists a valid partitioning into triangles in G' if and only if there exists a cluster editing set of size at most k in G.

Claim 4.2.6. ($\Rightarrow$) *If A' is a valid partition of the vertices of G' into triangles, it can be used to obtain a set $A \subseteq \binom{V(G)}{2}$ such that $G \triangle A$ gives a balanced cluster graph and $|A| \leq k$.*

For each triplet $\{x, y, z\} \in A'$, remove all edges from each vertex where the vertex adjacent is not in the same triplet. Since each vertex has 4 neighbours, where 2 has to be members of the same triplet, we see that the cost of edits are k. Now each component of G is a triangle, which implies that they are all of same size, or for the case of $\delta = 1$ or $\delta = 2$, components are within δ of each other.

Claim 4.2.7. ($\Leftarrow$) *If a set $A \subseteq \binom{V(G)}{2}$ is an editing set of size at most k such that $G \triangle A$ is a balanced cluster graph, the components of G corresponds to a partition into triangles of G'.*

To rule out that there exist a solution with components of size larger than 3, we see that the minimal cost of adding a vertex to each of the isolated complete components has a total cost greater than n, which is larger than the budget.

To create complete components smaller than 3 we observe that as each vertex in V' has degree 4. This means that we at least have to delete 2 from each for each vertex to be part of a component complete. As this deletion costs $n = k$, there is no budget left to delete further edges. This then shows that the complete components in a solution has to have size 3. If after this deletion there is a valid cluster editing instance then the triangles from G' is also a valid RPIT solution.

Since this reduction uses only edge deletions and we show that having the option to add in edges does not change the solution, we see that RPIT can be reduced to both EDITING and DELETION.

□

Theorem 4.2.8. FACTORCE, DIFFERENCECE, FACTORCD *and* DIFFERENCECD *are NP-Complete for every fixed* $\alpha \geq 1$ *and* $\delta > 0$*.*

Proof. By Lemma4.2.1 we show membership in NP, and by Lemma 4.2.2 and 4.2.5 we show hardness for all $\alpha \geq 1$ or $\delta \geq 0$. The problems are therefore NP-Complete. □

4.2.2 Completion

As we have allready shown that FACTORCC and DIFFERENCECC are in NP it only remains to prove that they are NP-hard. This will be done with a reduction from the strongly NP-Complete problem 3-PARTITION [13].

3-PARTITION

Input: A collection of positive integers $S = \{q_1, q_2 ... q_{3N}\}$

Question: Can S be partitioned into N triplets such that the elements of each triplet sums up to B where $B = \frac{\sum_{q \in S} q}{N}$

Lemma 4.2.9. FACTORCC *and* DIFFERENCECC *are NP-Hard for every fixed* $\alpha \geq 1$ *or* $\delta \geq 0$.

Proof. In this proof we will give a polynomial reduction from an instance $S = \{q_1, q_2 ... q_{3N}\}$ of 3-PARTITION where all integers $q \in S$ is bounded by $B/4 < q < B/2$. Let $Q = \sum_{q \in S} q$ be the sum of all elements in S, $B = \frac{Q}{N}$ the target sum of each of the N triplets, and the budget $k = (3Q^2 + 2QB + \frac{B^2}{3}) \cdot N$.

We construct the graph G, by for each element $q \in S$ we construct a complete component of size $q + Q$. We also construct $(BN)^2$ complete components of size $\alpha(B+3Q)$ or $\delta + (B+3Q)$ for FACTORCC and DIFFERENCECC respectively. The $3Q$ part is used to force each component to be a complete join of 3 original components and the B forces each component to represent elements with sum B. We will now show that there is a solution to FACTORCC and DIFFERENCECC if and only if there is a solution to 3-PARTITION, and that it can be used to get to this solution.

Claim 4.2.10. $(\Rightarrow)$ *If the set A' of triplets is a valid 3-partition of S, it can be used obtain a set $A \subseteq \binom{V(G)}{2} \setminus E(G)$ such that $|A| \leq k$ and $G + A$ is a balanced cluster graph.*

For each triplet in A', join the corresponding components in G. All components will then have size $B + 3Q$ which is within a factor of α or δ of the large components. Also, the cost of this join is at most $k = (3Q^2 + 2QB + \frac{B^2}{3}) \cdot N$, which is the case when each component is of size $\frac{B}{3} + Q$.

Claim 4.2.11. $(\Leftarrow)$ *If there is a set $A \subseteq \binom{V(G)}{2} \setminus E(G)$ of at most k of edges such that $G + A$ is a balanced cluster graph, it can be used to obtain a solution to* 3-PARTITION *of S.*

For the instance to be a YES-instance, each final component C_i has to have size $B + 3Q \leq |C_i| \leq (\alpha(B + 3Q)$ or $\delta + (B + 3Q))$. For each of the small components to reach this size they has to be joined with at least 2 other components to reach a size of $x + 3Q$. For a joining of 4 components to happen, the lowest cost is for a join of components of size $\{(B/4 + Q, B/4 + Q, B/4 + Q, B/4 + Q\}$. The cost of this is $(6Q^2 + 3QB + \frac{6B^2}{16})$. For the remaining $3N - 4$ components there are now 2 components which has to be joined into components of 4 since $3N - 4 \mod 3 = 2$. The total cost of these 3 joins into components of 4 has a cost of $(18Q^2 + 9QB + \frac{9B^2}{8})$, and the remaining budget is then $k' = (3N - 18)Q^2 + (2N - 9)BQ + (\frac{N}{3} - \frac{9}{8})B^2$. The reduced instance S' now has $3(N - 4)$ elements to be joined into $(N-4)$ components. The budget needed for this is $(3Q^2 + 2QB + \frac{B^2}{3}) \cdot (N-4)$. We will now show that this is more than the remaining budget in k'

$$3NQ^2 + 2NBQ + \frac{NB^2}{3} - 18Q^2 - 9BQ - \frac{9B^2}{8} \tag{4.1}$$

$$-3NQ^2 + 2NBQ + \frac{NB^2}{3} - 12Q^2 - 8BQ - \frac{4B^2}{3} \tag{4.2}$$

$$= -6Q^2 - BQ + \frac{5B^2}{24} \tag{4.3}$$

$$= B^2(-6N^2 - N + \frac{5}{24}) < 0, \quad \forall N > 0 \tag{4.4}$$

For FACTORCC, if a component of size $B/4 + Q$ joins with a big component of size $\alpha \cdot (B+3Q)$ with a cost of $(3\alpha Q^2 + \frac{7\alpha QB}{4} + \frac{\alpha B^2}{4})$, this then raises the lower bound on the size of components from $B + 3Q$ to $\frac{(\frac{1}{4}+\alpha)B+(1+3\alpha)Q}{\alpha}$. As we have allready shown that merging of 4 components is not possible, each of the $3N$ small components has to join with a large component. As the cost of one such join is $(3\alpha Q^2 + \frac{7\alpha QB}{4} + \frac{\alpha B^2}{4})$, the total will be $(9\alpha Q^2 + \frac{21\alpha QB}{4} + \frac{3\alpha B^2}{4}) \cdot N$ which is greater than k for all $\alpha \geq 1$.

The same is true for DIFFERENCECC, the cost of merging a component of size $B/4 + Q$ with a big component of size $\delta + (B + 3Q)$ has a cost of $(3Q^2 + \frac{7QB}{4} + \frac{B^2}{4} + \frac{\delta(B+D)}{4})$, and the cost of doing $3N$ such joins has a cost of $(9Q^2 + \frac{21QB}{4} + \frac{3B^2}{4} + \frac{3\delta(B+D)}{4}) \cdot N$ which is greater than k for all $\delta \geq 0$.

As we now see that each component has to be a result of joining exactly 3 components, it remains to show that each component will have size exactly $B+3Q$. Assume that a component has size $(B+x)+3Q$, then there will be another component with size at most $(B-x)+3Q$. This is not within the bound of $B+3Q$, and can't be part of a YES-instance.

Recreating a solution of 3-PARTITION can be done by checking which components who joined each other, and which elements of S they corresponds to. These 3 elements then creates a triple in a solution of S.

□

Theorem 4.2.12. FACTORCC *and* DIFFERENCECC *are NP-Complete for every fixed* $\alpha \geq 1$ *and* δ.

Proof. By Lemma4.2.1 we show membership in NP, and by Lemma 4.2.9 we show hardness for all $\alpha \geq 1$ or $\delta \geq 0$. The problems are therefore NP-Complete. □

4.3 Polynomial Kernels

We will now give polynomial kernels for all 6 variations of edge modification problems. As the kernels for absolute difference balancing are very similar to their factor variation, the only differences are in rules comparing sizes of components. We will therefore state the rules and give safeness proofs for FACTORCC, FACTORCD and FACTORCE, while for DIFFERENCECC, DIFFERENCECD and DIFFER-

ENCECE we will state all reduction rules, but only give proofs for safeness for rules which are not equal to their factor version.

For all kernels we define the trivial YES-instance and NO-instance as the same.

Definition 4.3.1. In the trivial YES-instance(G, k), let $V(G) = \{u\}, E(G) = \emptyset, k = 0$.

Definition 4.3.2. In the trivial NO-instance(G, k), let $V(G) = \{u, v, w\}, E(G) = \{\{u, v\}\{v, w\}\}, k = 0$.

4.3.1 FACTORCC

We will now provide a polynomial kernel for the FACTORCC problem. The rules in this kernel each takes an instance (G, k) of the problem as input and then returns a reduced instance (G', k') in polynomial time. For this kernel each rule are only applied once, and in the order they are stated.

Reduction Rule 1. If $k < 0$, then return the trivial NO-instance.

As the only operation is edge addition, we observe that if there is a component which is not a complete component, then we need to add all edges between vertices in the same component which is not already adjacent. If there are more than k edges missing, this is obviously a NO-instance.

Reduction Rule 2. For each pair of connected vertices u, v such that $\{u, v\} \notin E(G)$, then add {u,v} to $E(G)$, and reduce k by 1. The new instance is $(G + \{u, v\}, k - 1)$.

Lemma 4.3.1. *Reduction Rule 2 is safe and can be carried out in time $\mathcal{O}(|V|^2)$.*

Proof. In one direction, if A is a solution to $(G + \{u, v\}, k - 1)$ of size $|A| \leq k - 1$, then it is clear that $(A \cup \{u, v\}$ is a solution to (G, k).

For the other direction, assume than for an YES-instance(G, k) there is a solution A' such that $\{u, v\} \notin A$. In that case we see that the vertices u and v has to be in different components, but as our only operation is edge addition, and we know that they are in the same component in G this gives a contradiction. By this we see that there are no solution without $\{u, v\} \in A$.

Computing the components can be done by DFS or BFS, and then adding in the edges missing for all components to become complete is at worst case $\mathcal{O}(|V|^2)$.

□

Rule 2 uses the fact that we know that each component has to be a clique to just fill in all missing edges. As each component now is a complete component, we want to check the size of each component, and sort them in increasing size to check if they already fulfill the bounds on sizes. The sizes of each component is then stored in a list for use in later rules.

Reduction Rule 3. Sort all complete components by size in increasing order from C_1 to C_{max} and store the size of each component in a list. If $|V(C_{max})| \leq \alpha \cdot |V(C_1)|$ return the trivial YES-instance.

Lemma 4.3.2. *Reduction Rule 3 is safe and can be carried out in time $\mathcal{O}(|V|)$.*

Proof. If all components are complete components with sizes within a factor of α from each other then the instance is solved.

Checking for this is done by first computing the size of each component in linear time, and then sorting the list with bucket sort has running time bounded by $\mathcal{O}(|V|)$. □

Since we now know that not all components are within the bounds, and that we cant make the largest components smaller, we have to join the smallest components to make them larger. Because of this we can remove everything but the smallest components and the largest component C_{max} to compare against.

Reduction Rule 4. If the number of vertices is $|V(G)| \leq 2k$ skip to Rule 6. Else, remove all components C_i except C_{max} and $\{C_1, ..., C_x\}$, where x is th smallest integer such that such that the total size of all elements in $|V(C_1)| + ... + |V(C_x)| \geq 2k$.

Lemma 4.3.3. *Reduction Rule 4 is safe and can be carried out in time $\mathcal{O}(|V| + |E|)$.*

Proof. As adding in k edges affects $2k$ vertices, storing more than $2k$ vertices ensures that we can not join all components. By this we know that at least one of the components $C_1, ..., C_x$ are not joined with another component. This implies that the smallest component in a solved instance is either an untouched component, or the result of joining two or more components. By this we see that this rule does not affect the smallest component in the solved instance. To show that no component C_i in $C_1, ..., C_x$ should join a component which is greater than C_x, we see that the cost of this is greater than or equal to joining it with any of the components $C_1, ..., C_x$. This implies that we now have $x - 2$ components which may need joining. If C_i joins with a component in $C_1, ..., C_x$ so that they become large enough, we now have a greater budget left and $x - 3$ components to join.

□

By keeping at least one more component than we have budget to join we can use C_x to check if those removed had size at least $\frac{|V(C_{max})|}{\alpha}$.

Reduction Rule 5. If $|V(C_x)| < \frac{|V(C_{max})|}{\alpha}$, then return the trivial NO-instance. If not, delete it from the instance.

Lemma 4.3.4. *Reduction Rule 5 is safe.*

Proof. By Reduction Rule 4 we know that $|V(C_1)| + ... + |V(C_x) > 2k$. This implies that at least one component in $C_1, ..., C_x$ can not be joined with another component. We can therefore conclude that the smallest component in a solved instance has to be smaller than or equal in size to C_x. □

Since we know that there are some small components which has to be joined, this gives a bound on the size of the largest components, since the largest components we can create with merging is of size $k + 1$.

Reduction Rule 6. If $|V(C_{max})| > \alpha \cdot (k + 1)$ return the trivial NO-instance.

Lemma 4.3.5. *Reduction Rule 6 is safe.*

Proof. As the cost of joining C_i and C_j is $|V(C_i)| \cdot |V(C_j)|$, creating a component $|V(C_{i+j})| \geq k+2$ cost at least $k+1$ with the minimum being $|V(C_i)| = 1$ and $|V(C_j)| = k+1$. $\square$

We also want to return a NO-instance if no joins at all is possible. This is the case when even the two smallest components can not be joined.

Reduction Rule 7. If $|V(C_1)| \cdot |V(C_2)| > k$ return the trivial NO-instance

Lemma 4.3.6. *Reduction Rule 7 is safe.*

Proof. If even joining the two smallest components with each other has a cost higher than k then no joins is possible in G and it's a NO-instance. $\square$

Lemma 4.3.7. *Reduction Rules 5, 6 and 7 can be carried out in time $\mathcal{O}(1)$.*

Proof. As we have stored the size of each component in a sorted list. These rules is applied by lookup of specific indexes in the list. $\square$

4.3.2 Analysis

Theorem 4.3.8. FACTORCC *has a polynomial kernel with $\mathcal{O}(k)$ vertices that can be constructed in $\mathcal{O}(|V|^2)$ time.*

Proof. Each reduction rule is applied in order, and only applied once. With each rule having a running time bounded by at most $\mathcal{O}(|V|^2)$ the kernel has a total running time also bounded by $\mathcal{O}(|V|^2)$.

To give a bound on the number of vertices in the reduced instance we see that there are at most $3k$ vertices in the small components, and that the size of C_{max} is no larger than $\alpha \cdot (k+1)$.The total number of vertices is then $|V(G)| \leq 3k + \alpha(k+1)$. $\square$

Theorem 4.3.9. FACTORCC *has a FPT-algorithm with running time $k^{\mathcal{O}(k)} + \mathcal{O}(|V|^2)$.*

Proof. By first employing the kernelization from Theorem 4.3.8 running in time $\mathcal{O}(|V|^2)$, we have an instance with $\mathcal{O}(k)$ vertices. By then brute force joining k components and checking if the graph is a balanced cluster graph, we get an algorithm running in time $k^{\mathcal{O}(k)} + \mathcal{O}(|V|^2)$. $\square$

4.3.3 DIFFERENCECC

For the related problem of adding edges to achieve a cluster graph such that the size of all components C_i, C_j are within a difference of δ of each other. That is, $\forall C_i, C_j \in G : |V(C_i)| \leq |V(C_j)| + \delta$, a polynomial kernel can be achieved by by mostly following the rules for the FACTORCC kernel. Because of this we will state the rules, but for those who are the same as for FACTORCC we refer to those for the proof of safeness and running time.

Reduction Rule 1. If $k < 0$ return the trivial NO-instance.

Reduction Rule 2. For each pair of connected vertices u, v such that $\{u, v\} \notin E(G)$, then add {u,v} to $E(G)$, and reduce k by 1. The new instance is $(G + \{u, v\}, k - 1)$.

Reduction Rule 3. Sort all complete component by size in increasing order from C_1 to C_{max} and store the size of each component in a list. If $|V(C_{max})| \leq \delta + |V(C_1)|$ return the trivial YES-instance.

Lemma 4.3.10. *Reduction Rule 3 is safe and can be carried out in time* $\mathcal{O}(|V|)$.

Proof. If all components are complete graphs with sizes within a difference of δ from each other the instance is solved.

Checking for this is done by first computing the sizes of the components in linear time, and then sorting the list with bucket sort has running time bounded by $\mathcal{O}(|V|)$. □

Reduction Rule 4. If the number of vertices is $|V(G)| \leq 2k$ skip to Rule 6. Else, remove all components C_i except C_{max} and $\{C_1, ..., C_x\}$, where x is th smallest integer such that such that the total size of all elements in $|V(C_1)| + ... + |V(C_x)| > 2k$.

Reduction Rule 5. If $|V(C_{max-1})| + \delta < |V(C_{max})|$, then return the trivial NO-instance. If not, delete it from the instance.

Lemma 4.3.11. *Reduction Rule 5 is safe.*

Proof. By Reduction Rule 4 we know that $|V(C_1)| + ... + |V(C_x) > 2k$. This implies that at least one component in $C_1, ..., C_x$ can not be joined with another component. We can therefore conclude that the smallest component in a solved instance has to be smaller than or equal in size to C_x. □

Reduction Rule 6. If $|V(C_{max})| > k + 1 + \delta$ return the trivial NO-instance.

Lemma 4.3.12. *Reduction Rule 6 is safe.*

Proof. As cost of joining C_i and C_j is $|V(C_i)| \cdot |V(C_j)|$, creating a cluster $|V(C_{i+j})| \geq k + 2$ cost at least $k + 1$ with the minimum being $|V(C_i)| = 1$ and $|V(C_j)| = k + 1$. □

Reduction Rule 7. If $|V(C_1)| \cdot |V(C_2)| > k$ return the trivial NO-instance

4.3.4 Analysis

Theorem 4.3.13. DIFFERENCECC *has a polynomial kernel with* $\mathcal{O}(k)$ *vertices that can be constructed in* $\mathcal{O}(|V|^2)$ *time.*

Proof. As for the FACTORCC, all rules are applied only once, with a running time bounded by $\mathcal{O}(|V|^2)$.

The number of vertices in the reduced instance is at most $|V(G')| \leq 4k + 1 + \delta$. □

Theorem 4.3.14. DIFFERENCECC *has a FPT-algorithm with running time* $k^{\mathcal{O}(k)} + \mathcal{O}(|V|^2)$.

Proof. By first employing the kernelization from Theorem 4.3.13 running in time $\mathcal{O}(|V|^2)$, we have an instance with $\mathcal{O}(k)$ vertices. By then brute force joining k components and checking if the graph is a balanced cluster graph, we get an algorithm running in time $k^{\mathcal{O}(k)} + \mathcal{O}(|V|^2)$. □

4.3.5 FACTORCD

In the polynomial kernel for Edge Deletion we will use two data structures in addition to the instance (G, k). These will be a list I_v of the complete components we remove from G, sorted by size to easy find the smallest and largest components, and the critical clique graph $\mathcal{K}$ of G. The reduction rules will be applied exhaustively, by always applying the lowest numbered rule which can be applied.

Reduction Rule 1. If $k < 0$, then return the trivial NO-instance

Now we will employ a procedure first introduced by Guo et al [15]. By computing the critical clique graph $\mathcal{K}$ of G, we can use that to both get a bound of the number of vertices in G, and to find edges between critical cliques which has to be deleted.

Reduction Rule 2. Build the critical clique graph $\mathcal{K}$ of G. For each isolated vertex in $\mathcal{K}$, remove it from $\mathcal{K}$ and move the corresponding component from G to I_v.

Lemma 4.3.15. *Reduction Rule 2 is safe and can be carried out in time $\mathcal{O}(|V| + |E|)$.*

Proof. This rule does not change the instance in any way. Moving components from G to I_v preserves them for later checking for size bounds.

The graph $\mathcal{K}$ can be constructed in time $\mathcal{O}(|V| + |E|)$ by lexicographical sorting of G. The critical cliques are then the vertices in the same class. Moving a component from G to I_v is also a linear operation. □

If now G is empty, then we know that each component is a complete component, and if the size bound is upheld we have an YES-instance.

Reduction Rule 3. If $G = \emptyset$ and for all $C_i, C_j \in I_v$ it holds that $|V(C_i)| \leq \alpha|V(C_j)|$, then return the trivial YES-instance.

Lemma 4.3.16. *Reduction Rule 3 is safe and can be carried out in time $\mathcal{O}(|V| + |E|)$.*

Proof. As G is empty, each component is a complete component. Computing the size of all components can be done in $\mathcal{O}(|V| + |E|)$. By finding the smallest component and comparing all other components against this, the number of comparisons is linear. If all components already are within the size bound we can return a trivial YES-instance. □

As we now have removed all complete components from G, we know that each component in G demands at least one deletion. By this property we can give a bound on the number of components in a YES-instance. We can also give a bound on the number of vertices in $\mathcal{K}$ by the property that each vertex is part of at least one P_3.

Reduction Rule 4. If the number of components in $\mathcal{K}$ is greater than k or the number of vertices in $\mathcal{K}$ is greater than $4k$, then return the trivial NO-instance.

Lemma 4.3.17. *Reduction Rule 4 is safe and can be carried out in time $\mathcal{O}(|V| + |E|)$.*

Proof. As each component left in G after application of Rule 2 need at least 1 deletion to become a complete component, there can be at most k components for the instance to be a YES-instance.

To bound the number of vertices in $\mathcal{K}$, observe that each vertex $K_i \in \mathcal{K}$ is part of at least one P_3 by the properties of a critical clique graph. This means that for every edge $\{K_i, K_j\} \in E(\mathcal{K})$ there exist at least one vertex $K_x \in \mathcal{K}$ such that exactly one of the edges $\{K_i, K_x\}$ or $\{K_x, K_j\}$ are in $E(\mathcal{K})$. For there to be a clique of size x in $\mathcal{K}$ there is at least $x - 1$ edges in $E(\mathcal{K})$ which has to be deleted to make the component P_3-free.

To maximize the number of vertices in $\mathcal{K}$ of a YES-instance, observe that a component with only one deletion necessary can't have a clique of size greater than 2. Any way of connecting 3 or more vertices in $\mathcal{K}$ requires at least 1 deletion for the component to become complete. To maximize the ratio of vertices in G per deletion required, observe that for a clique of size x this ratio is $\frac{x}{x-1}$. This has a maximum at $x = 2$. As we can then create a component comprised of two cliques of size 2 connected by a single edge, we get a P_4, which has 4 vertices and requires 1 deletion. As we have a bound of k components, we see that $|V(\mathcal{K})| \leq 4k$.

By this we see that the highest number of critical cliques in a YES-instance can be at most $4k$. Checking for this can be done in linear time by e.g. DFS in $\mathcal{K}$. □

Two neighbouring critical cliques K_i and K_j has $|V(K_i)| \cdot |V(K_j)|$ edges between them, and therefore the cost of making them two different complete components costs $|V(K_i)| \cdot |V(K_j)|$. If an edge $\{K_i, K_j\} \in E_{\mathcal{K}}$ has a cost of more than k to delete, we know that $V(K_i)$ and $V(K_j)$ has to be in the same component. To achieve this, we have to remove all edges to vertices adjacent to one of the critical cliques, but not adjacent to the other.

Reduction Rule 5. For some edge $\{K_i, K_j\} \in E_{\mathcal{K}}$ such that $|V(K_i)| \cdot |V(K_j)| > k$, if there exist an edge $\{u, x\} \in E(G)$ where $u \in V(K_i) \cup V(K_j)$ and $x \notin N(K_i) \cap N(K_j)$, then delete $\{u, x\}$ and decrease k by 1. The new instance is then $(G - \{u, x\}, k - 1)$.

Lemma 4.3.18. *Reduction Rule 5 is safe and can be carried out in time $\mathcal{O}(|V| + |E|)$.*

Proof. In one direction, assume that after application of Rule 5 the reduced instance (G', k') is a NO-instance. As the instance (G, k) was a YES-instance, there had to be a way to make the component which the critical cliques K_i and K_j belongs to into complete components. By application of Rule 5 we made the components such that K_i and K_j is part of the same component. If this made the instance into a NO-instance, this implies that K_i and K_j should belong to different components. To do this we have to remove all $|V(K_i)| \cdot |V(K_j)|$ edges between them. But as the number of edges between them is greater than k, we don't have enough budget for this operation and it is therefore not possible.

For the other direction, if(G', k') is a YES-instance, observe that we decrease the budget k by the number of edges removed. So if (G', k') is a YES-instance, then there exist a set A of edges such that $(G' \cup A, k' + |A|) = (G, k)$.

Finding an edge in $\mathcal{K}$ were this rule applies can be done by DFS in time $\mathcal{O}(|V| + |E|)$. Then finding the critical cliques which is neighbour of only one of them can be done in time $\mathcal{O}(k)$ as we have already bounded the number of vertices in $\mathcal{K}$.

□

After application of Rule 5 each edge in $E_{\mathcal{K}}$ has a deletion cost of at most k. This means that the size of two adjacent critical cliques K_i and K_j is at most $|V(K_i)| + |V(K_j)| \leq k+1$ as this is the maximum of the function $|V(K_i)| \cdot |V(K_j)| \leq k$. As we also know that a clique of size x in $\mathcal{K}$ demands at least $x-1$ deletions, we know that in a YES-instance, there is no cliques of size greater than $k+1$ in $\mathcal{K}$. By applying the first bound on each edge of a clique of size $k+1$, we get a clique containing one critical clique of size k and k critical cliques of size 1, for a total size of the clique of $2k$. By this we know that if there exists components in I_v of size greater than $\alpha \cdot 2k$ and the instance is not already solved it is a NO-instance.

Reduction Rule 6. If there exist a component C in I_v such that $|V(C)| > \alpha \cdot 2k$, then return the trivial NO-instance.

Lemma 4.3.19. *Reduction Rule 6 is safe and can be carried out in time $\mathcal{O}(|V|)$.*

Proof. If $G = \emptyset$ we know by Rule 3 that the instance is not already solved, and since the cost of splitting complete graphs of size greater than $2k$ is more than k, we therefore have a NO-instance.

Otherwise, if G is not empty, creating a complete graph of $k+1$ critical cliques has an editing cost of at least k by the definition of a critical clique graph. By Rule 5, no critical clique has size greater than k. Assume that vertices $\{K_1, K_2, ..., K_x\} \in \mathcal{K}$ creates a clique in $\mathcal{K}$, and at least $x-1$ of the vertices has an edge to at least one vertex not adjacent to any other critical clique in the clique. To remove the vertices not in the clique from the component the number of edges which has to be removed is the size of their neighbouring critical clique. As we know that there are at least $x-1$ such vertices, x can be at most $k+1$, and $\sum_{i=1}^{x-1} |V(K_i)| \leq k$. By then having $|V(K_x)| = k$ this gives us a maximal size of the clique at $k + k = 2k$.

Application of the rule is done in linear time, as computing the sizes of complete components is bounded by $\mathcal{O}(|V|)$. □

Reduction Rule 7. If I_v has more than $k+2$ components, remove all components except C_{min} and the $k+1$ largest components.

Lemma 4.3.20. *Reduction Rule 7 is safe and can be carried out in time $\mathcal{O}(|V|)$.*

Proof. If more than k components needs a split to be within the size bound, we have a NO-instance, so by storing $k+1$ largest components, we know that at least one of them has to be within the size bound already for there to be a YES-instance. We also don't know if the smallest component of the solution is C_{min} or a part of one of the components in G, so we have to store C_{min} too. □

Reduction Rule 8. Return $(G \cup I_v, k)$

4.3.6 Analysis

Theorem 4.3.21. FACTORCD *has a polynomial kernel with $\mathcal{O}(k^2)$ vertices that can be constructed in $\mathcal{O}(|E|(|V| + |E|))$ time.*

Proof. As the reduction rules are applied exhaustively, with the number of edges bounding the number of application of each rule, and each rule can be applied in time $\mathcal{O}(|V| + |E|)$ the total running time of the kernel is then $\mathcal{O}(|E|(|V| + |E|))$.

To give a bound on the number of vertices in the reduced instance, observe that there are at most $4k$ critical cliques in G, each with a size of at most k.This gives a bound of $|V(G)| \leq 4k^2$. In addition to this we have at most $k + 2$ components in I_v, with a maximum number of vertices as $|V(I_v)| \leq (k+2) \cdot \alpha 2k = \alpha \cdot (2k^2 + 4k)$. The total number of vertices in the reduced instance is then $|V(G \cup I_v)| \leq 4k^2 + \alpha \cdot (2k^2 + 4k)$. □

Theorem 4.3.22. FACTORCD *has a FPT-algorithm with running time $k^{\mathcal{O}(k)} + \mathcal{O}(|E|(|V| + |E|))$.*

Proof. By first employing the kernelization from Theorem 4.3.21 running in time $\mathcal{O}(|E|(|V| + |E|))$, we have an instance with $\mathcal{O}(k^2)$ vertices. By then using the trivial $\mathcal{O}(2^k)$ branching algorithm by on each remaining P_3, branch in two by removing either of them. When the graph is P_3-free, brute force splitting of components which are too big and checking if the graph is a balanced cluster graph, we get an algorithm running in time $k^{\mathcal{O}(k)} + \mathcal{O}(|E|(|V| + |E|))$. □

4.3.7 DIFFERENCECD

For the problem of deleting edges to achieve a cluster graph such that the size of all complete components is C_i, C_j are within a difference of δ of each other $\forall C_i, C_j \in G : |V(C_i)| \leq |V(C_j)| + \delta$, most of the reduction rules from the kernel for FACTORCD can be used. Because of this safeness of the reduction rules which are the same is shown in the the chapter for FACTORCD.

Reduction Rule 1. If $k < 0$, then return the trivial NO-instance.

Reduction Rule 2. Build the critical clique graph $\mathcal{K}$ of G. For each isolated vertex in $\mathcal{K}$, remove it from $\mathcal{K}$ and move the corresponding component from G to I_v.

Reduction Rule 3. If $G = \emptyset$ and for each pair of components $C_i, C_j \in I_v$ we have that $|V(C_i)| \leq |V(C_j)| + \delta$, then return the trivial YES-instance.

Lemma 4.3.23. *Reduction Rule 3 is safe.*

Proof. As G is empty, each component is a complete graph. Checking the sizes of all components can be done in $\mathcal{O}(|V| + |E|)$. By finding the smallest component and comparing all components against this the number of comparisons is linear. If the sizes of all components is within the bound we have a YES-instance. □

Reduction Rule 4. If the number of components in $\mathcal{K}$ is greater than k or number of vertices in $\mathcal{K}$ is greater than $4k$, then return the trivial NO-instance.

Reduction Rule 5. If for some edge $\{K_i, K_j\} \in E_{\mathcal{K}}$ such that$|V(K_i)| \cdot |V(K_j)| > k$. If there exist an edge $\{u, x\} \in E(G)$ where $u \in V(K_i) \cup V(K_j)$ and $x \notin N(K_i) \cap N(K_j)$, then delete $\{u, x\}$ and decrease k by 1. The new instance is then $(G - \{u, x\}, k - 1)$.

Reduction Rule 6. If there exist a component C in I_v such that $|V(C)| > 2k + \delta$, then return the trivial NO-instance.

Lemma 4.3.24. *Reduction Rule 6 is safe.*

Proof. As we showed in Reduction Rule 6 of FACTORCD kernel, by deleting edges in G, the largest complete graph we can create has size $2k$. Because of this, no YES-instancecan contain a complete graph of size greater than $2k + \delta$.

□

Reduction Rule 7. If I_v has more than $k + 2$ components, remove all components except C_{min} and the $k + 1$ largest components.

4.3.8 Analysis

Theorem 4.3.25. DIFFERENCECD *has a polynomial kernel with $\mathcal{O}(k^2)$ vertices that can be constructed in $\mathcal{O}(|E|(|V| + |E|))$ time.*

Proof. As for the FACTORCD, all rules are applied exhaustively, with each rule having a running time bounded by $\mathcal{O}(|V| + |E|)$. As the number of application per rule is bounded by the number of edge deletions, we get a total running time of $\mathcal{O}(|E|(|V| + |E|))$.

The number of vertices in the reduced instance is at most $|V(G \cup I_v)| \leq 6k^2 + 4k + \delta$. □

Theorem 4.3.26. DIFFERENCECD *has a FPT-algorithm with running time $k^{\mathcal{O}(k)} + \mathcal{O}(|E|(|V| + |E|))$.*

Proof. By first employing the kernelization from Theorem 4.3.25 running in time $\mathcal{O}(|E|(|V| + |E|))$, we have an instance with $\mathcal{O}(k^2)$ vertices. By then using the trivial $\mathcal{O}(2^k)$ branching algorithm by on each remaining P_3, branch in two by removing either of them. When the graph is P_3-free, brute force splitting of components which are too big and checking if the graph is a balanced cluster graph, we get an algorithm running in time $k^{\mathcal{O}(k)} + \mathcal{O}(|E|(|V| + |E|))$. □

4.3.9 FACTORCE

The kernel for FACTOR-α BALANCED CLUSTER EDITING builds on ideas developed by Guo et al. [15] for the problem of CLUSTER EDITING. Similar to their kernel we will employ the technique of building the critical clique graph $\mathcal{K}$ of G. In addition to $\mathcal{K}$ we also need a list I_v to keep track of complete components which we remove from G so that they can be stored until we need to compare sizes of components. This kernel also heavily builds on the kernel for FACTORCD, as e.g. the technique for giving a bound on the number of vertices in $\mathcal{K}$ is equal.

The reduction rules will be applied exhaustively by always applying the lowest numbered rule which can be applied.

Reduction Rule 1. If $k < 0$, then return the trivial NO-instance.

Reduction Rule 2. Build the critical clique graph $\mathcal{K}$ of G. For each isolated vertex in $\mathcal{K}$, remove it from $\mathcal{K}$ and move the corresponding component from G to I_v.

Lemma 4.3.27. *Reduction Rule 2 is safe and can be carried out in time $\mathcal{O}(|V| + |E|)$.*

Proof. This rule does not change the instance in any way. Moving components from G to I_v preserves them for later checking against size bounds.

The graph $\mathcal{K}$ can be constructed in time $\mathcal{O}(|V| + |E|)$ by lexicographical sorting of G. The critical cliques are then the vertices in the same class. Moving a component from G to I_v is also a linear operation. □

If now G is empty, then we know that each component is a complete graph, and if the size constraint is upheld for all components in I_v, we have a YES-instance.

Reduction Rule 3. If $G = \emptyset$ and for each pair of components $C_i, C_j \in I_v$ we have that $|V(C_i)| \leq \alpha|V(C_j)|$, then return the trivial YES-instance.

Lemma 4.3.28. *Reduction Rule 3 is safe and can be carried out in time $\mathcal{O}(|V|)$.*

Proof. As G is empty, each component is a complete graph. Computing the sizes of all components can be done in $\mathcal{O}(|V|)$. By finding the smallest component and comparing all components against this the number of comparisons is linear. If the sizes of all components is within the bound we have a YES-instance. □

Reduction Rule 4. If the number of components in $\mathcal{K}$ is greater than k or number of vertices in $\mathcal{K}$ is greater than $4k$, then return the trivial NO-instance.

Lemma 4.3.29. *Reduction Rule 4 is safe and can be carried out in time $\mathcal{O}(|V| + |E|)$.*

Proof. As each component left in G after application of rule 2, need at least one edge deletion or addition to become a complete component, there can be at most k components for the instance to be a YES-instance.

To bound the number of vertices in $\mathcal{K}$, observe that each vertex $K_i \in \mathcal{K}$ is part of at least one P_3 by the properties of a critical clique graph. This means that for every edge $\{K_i, K_j\} \in E(\mathcal{K})$ there exist at least one vertex $K_x \in \mathcal{K}$ such that exactly one of the edges $\{K_i, K_x\}$ or $\{K_x, K_j\}$ are in $E(\mathcal{K})$. For each such vertex K_x, either at least one edge has to be added in, or one edge has to be deleted. For there to be a clique of size x in $\mathcal{K}$ it then has to be done at least $x - 1$ edits to make the component P_3-free.

To maximize the number of vertices in $\mathcal{K}$ of a YES-instance, observe that a component with only one deletion necessary can't have a clique of size greater than 2. Any way of connecting 3 or more vertices in $\mathcal{K}$ requires at least 1 deletion for the component to become complete. To maximize the ratio of vertices in G per deletion required, observe that for a clique of size x this ratio is $\frac{x}{x-1}$. This has a maximum at $x = 2$. As we can then create a component comprised of two cliques of size 2 connected by a single edge, we get a P_4, which has 4 vertices and requires 1 deletion. As we have a bound of k components, we see that $|V(\mathcal{K})| \leq 4k$.

By this we see that the highest number of critical cliques in a YES-instancecan be at most $4k$. Checking for this can be done in linear time by e.g. DFS in $\mathcal{K}$. □

Now if there is a critical clique K of size greater than k in G, we know that we can neither delete any edges between K and its neighbouring critical cliques or add any edges between K and critical cliques not in $N_{\mathcal{K}}[K]$. By this we then know that we have to delete any edge between a neighbour of K and a non-neighbour, and also add edges between vertices in $N_{\mathcal{K}}[K_i]$ which is not adjacent.

Reduction Rule 5. If for some critical clique $K \in \mathcal{K}$ such that $|V(K)| > k$, there exist an edge $\{u, v\} \in E(G)$ where $u \in \bigcup_{K' \in N_{\mathcal{K}}(K)} V(K')$ and $v \notin \bigcup_{K' \in N_{\mathcal{K}}[K]} V(K')$, then delete the edge $\{u, v\}$ and decrease k by 1. If there exist two vertices x, y in $\bigcup_{K' \in N_{\mathcal{K}}[K]} V(K')$ such that $\{x, y\} \notin E(G)$, then add the edge to $E(G)$ and decrease k by 1.

Lemma 4.3.30. *Reduction Rule 5 is safe is safe and can be carried out in time $\mathcal{O}(|V| + |E|)$.*

Proof. To show that if the instance (G, k) is a YES-instance, then the reduced instance (G', k') is a YES-instance we assume the opposite to show that this is not correct. If the reduced instance (G', k') is a NO-instance, then that implies that there is another way to make the big critical clique K_i part of a component which is a complete graph. For K_i to be part of the same component as a vertex $v \notin \bigcup_{K' \in N_{\mathcal{K}}[K_i]} V(K')$ the set A of edged added between vertex v and each of the vertices in K_i, is greater than k.

Likewise, if a vertex $v \in \bigcup_{K' \in N_{\mathcal{K}}[K_i]} V(K')$ is not in the same complete graph as the vertices of K_i, the set A of edges to delete is greater than k. Lastly, if a vertices $u, v \in V(K_i)$ is to end up in two different components, the cost of splitting a complete graph of size $k + 1$ into two components of size k and 1 has a cost of k. In addition to this, size $K_i \in \mathcal{K}$ this implies it has at least 1 neighbour, which has to have its edges to one of the splits of K_i removed, for a cost which then exceeds k.

To show that if the reduced instance (G', k') is a YES-instance implies that the original instance (G, k) also is a YES-instance, observe that we decrease the budget k by the number of edges removed. So if (G', k') is a YES-instance, then there exist a set A of edges such that $(G' \Delta A, k' + |A|) = (G, k)$.

Finding a big critical clique is done by traversing $\mathcal{K}$. Then finding critical cliques with distance two from a big critical clique can be done by BFS. Checking if the neighbours of a critical clique are adjacent to each other can also be done in linear time. □

After application of Rule 5 we can now give a bound on how big complete components we can create in G. This is possible as we have a maximum size of each critical clique, and also a bound on the number of critical cliques in a clique in $\mathcal{K}$.

Reduction Rule 6. If there exist a component $C \in I_v$ such that $|V(C)| > \alpha \cdot (2k + 1)$, then return the trivial NO-instance.

Lemma 4.3.31. *Reduction Rule 6 is safe and can be carried out in time $\mathcal{O}(|V| + |E|)$.*

Proof. To prove this rule we will show that by neither adding or removing edges to G can we create a complete graph of size greater than $2k+1$. If $G = \emptyset$ we know by Rule 3 that the instance is not already solved, and since the cost of splitting or merging complete graphs of size greater than $2k$ is more than k, we therefore have a NO-instance.

Otherwise, if G is not empty, creating a complete graph of $k+1$ critical cliques has an editing cost of at least k by the definition of a critical clique graph. By Rule 5, no critical clique has size greater than k. Assume that vertices $\{K_1, K_2, \ldots.., K_x\} \in \mathcal{K}$ creates a clique in $\mathcal{K}$, and at least $x-1$ of the vertices has an edge to at least one vertex not adjacent to any other critical clique in the clique. Then we either has to add an edge between the vertex not in the clique and each of the $x-1$ critical cliques in the clique it's not adjacent to, or delete all edges between it and all adjacent vertices in the clique. For the vertex to be included into the clique, it can at most require k edge additions. If it already is connected to a critical clique of size k this now gives a maximal size of the clique as $k+k+1 = 2k+1$.

By instead removing the vertex from the component the number of edges which has to be removed is the size of the neighbouring critical clique. As we know that there are at least $x-1$ such vertices, x can be at most $k+1$, and $\sum_{i=1}^{x-1} |K_i| \leq k$. By then having $|K_x| = k$ this gives us a maximal size of the complete graph at $k+k=2k$.

Each application of the rule is done in linear time, as computing the sizes of each component is bounded by $\mathcal{O}(|V|+|E|)$.

□

Reduction Rule 7. If I_v has more than $3k+2$ components, remove all components except the $2k+1$ smallest and the $k+1$ largest.

.

Lemma 4.3.32. *Reduction Rule 7 is safe and can be carried out in time $\mathcal{O}(|V|+|E|)$*

Proof. If more than k large components needs splitting to be within the size bound, we have a YES-instance, so by storing the $k+1$ largest components, we know that at least one of them has to be within the size bound already for there to be a YES-instance. As merging two small components has a cost of at least 1, keeping $2k+1$ smallest componets we know that at least one of them can't be merged, and therefore already has to be big enough.

Sorting the list by sizes of components can be done in time $\mathcal{O}(|V|)$ with bucket sort. Then removing the middle components can be done in $\mathcal{O}(|V|)$ □

Reduction Rule 8. Return $(G \cup I_v, k)$

4.3.10 Analysis

Theorem 4.3.33. FACTORCE *has a polynomial kernel with $\mathcal{O}(k^2)$ vertices that can be constructed in $\mathcal{O}(|V|^2(|V|+|E|))$ time.*

Proof. After exhaustive application of Rules 1 to 7, we have at most k components with $4k$ critical cliques in G which implies that $|V(G)| \leq 4k^2$, and at most $3k+2$ components with size at most $\alpha \cdot (2k+1)$ in I_v. The number of vertices in $V(I_v)$ is bounded the fact that each component has a size at most $\alpha \cdot (2k+1)$, and therefore the total number of vertices $|V(I_v)| \leq (3k+2) \cdot \alpha \cdot (2k+1) = \alpha \cdot (6k^2 + 7k + 2)$. The total size of the kernel is then $|V(G \cup I_v)| \leq 4k^2 + \alpha \cdot (6k^2 + 7k + 2)$.

Each reduction rule has running time at most $\mathcal{O}(|V| + |E|)$, and the number of times each rule can be applied is bounded by the number of edges either added or deleted. This is bounded by $|V|^2$, which gives a bound on the running time of $\mathcal{O}(|V|^2(|V| + |E|))$ □

Theorem 4.3.34. FACTORCE *has a FPT-algorithm with running time* $k^{\mathcal{O}(k)} + \mathcal{O}(|V|^2(|V| + |E|))$.

Proof. By first employing the kernelization from Theorem 4.3.33 running in time $\mathcal{O}(|V|^2(|V| + |E|))$, we have an instance with $\mathcal{O}(k^2)$ vertices. By then using the trivial $\mathcal{O}(3^k)$ branching algorithm by on each remaining P_3, branch in three by removing either of them or adding in the missing edge. When the graph is P_3-free, brute force splitting of big components and joining small components and checking if the graph is a balanced cluster graph. By this we get an algorithm running in time $k^{\mathcal{O}(k)} + \mathcal{O}(|V|^2(|V| + |E|))$. □

4.3.11 DIFFERENCECE

For the problem of editing to a cluster graph such that the size of all components is C_i, C_j is within a difference of δ of each other $\forall C_i, C_j \in G : |V(C_i)| \leq |V(C_j)| + \delta$, most of the reduction rules from the kernel for FACTORCE can be used. Because of this safeness of the reduction rules which are the same is shown in the the chapter for FACTORCE kernel.

Reduction Rule 1. If $k < 0$, then return the trivial NO-instance.

Reduction Rule 2. Build the critical clique graph $\mathcal{K}$ of G. For each isolated vertex in $\mathcal{K}$, remove it from $\mathcal{K}$ and move the corresponding component from G to I_v.

Reduction Rule 3. If $G = \emptyset$ and for each pair of components $C_i, C_j \in I_v$ we have that $|V(C_i)| \leq |V(C_j)| + \delta$, then return the trivial YES-instance.

Lemma 4.3.35. *Reduction Rule 3 is safe and can be carried out in time* $\mathcal{O}(|V| + |E|)$.

Proof. As G is empty, each component is a complete graph. Checking the sizes of all components can be done in $\mathcal{O}(|V| + |E|)$. By finding the smallest component and comparing all components against this the number of comparisons is linear. If the sizes of all components is within the bound we have a YES-instance. □

Reduction Rule 4. If the number of components in $\mathcal{K}$ is greater than k or number of vertices in $\mathcal{K}$ is greater than $4k$, then return the trivial NO-instance.

Reduction Rule 5. If for some critical clique $K \in \mathcal{K}$ such that $|V(K)| > k$, there exist an edge $\{u, v\} \in E(G)$ where $u \in \bigcup_{K' \in N_{\mathcal{K}}(K)} V(K')$ and $v \notin \bigcup_{K' \in N_{\mathcal{K}}[K]} V(K')$, then delete the edge $\{u, v\}$ and decrease k by 1. If there exist two vertices x, y in $\bigcup_{K' \in N_{\mathcal{K}}[K]} V(K')$ such that $\{x, y\} \notin E(G)$, then add the edge to $E(G)$ and decrease k by 1.

Reduction Rule 6. If there exist a component $C \in I_v$ such that $|V(C)| > 2k + 1 + \delta$, then return the trivial NO-instance.

Lemma 4.3.36. *Reduction Rule 6 is safe.*

Proof. As we showed in Reduction Rule 6 of FACTORCE kernel, by neither adding or deleting edges can we make a complete graph of size greater than $2k + 1$. Because of this, no YES-instance can contain a complete graph of size greater than $2k + 1 + \delta$.

□

Reduction Rule 7. If I_v has more than $3k + 2$ components, remove all components except the $2k + 1$ smallest and the $k + 1$ largest

4.3.12 Analysis

Theorem 4.3.37. DIFFERENCECE *has a polynomial kernel with $\mathcal{O}(k^2)$ vertices that can be constructed in $\mathcal{O}(|V|^2(|V| + |E|))$ time.*

Proof. After application of Rules 1 to 7 we have a kernel of size at most $|V(G \cup I_v)| \leq 10k^2 + 7k + 2 + \delta$.

Each reduction rule has running time at most $\mathcal{O}(|V| + |E|)$, and the number of times each rule can be applied is bounded by the number of edges either added or deleted. This is bounded by $|V|^2$, which gives a bound on the running time of $\mathcal{O}(|V|^2(|V| + |E|))$ □

Theorem 4.3.38. DIFFERENCECE *has a FPT-algorithm with running time $k^{\mathcal{O}(k)} + \mathcal{O}(|V|^2(|V| + |E|))$.*

Proof. By first employing the kernelization from Theorem 4.3.37 running in time $\mathcal{O}(|V|^2(|V| + |E|))$, we have an instance with $\mathcal{O}(k^2)$ vertices. By then using the trivial $\mathcal{O}(3^k)$ branching algorithm by on each remaining P_3, branch in three by removing either of them or adding in the missing edge. When the graph is P_3-free, brute force splitting of big components and joining small components and checking if the graph is a balanced cluster graph. By this we get an algorithm running in time $k^{\mathcal{O}(k)} + \mathcal{O}(|V|^2(|V| + |E|))$. □

Chapter 5

Open Problems and Future Work

In this thesis we have shown that FACTOR-α BALANCED CLUSTERING and DIFFERENCE-δ BALANCED CLUSTERING are NP-complete for all graph modification operations we have studied, and also showed that they admit polynomial kernels.

As the kernel for WEIGHTED FACTORCVD has $\mathcal{O}(k^2)$ vertices, while the current best kernel for CLUSTER VERTEX DELETION has $\mathcal{O}(k^{\frac{5}{3}})$. This raises the question of does the two problems have the same lower bound on the number of vertices. Another open problem is whether the weight bound of $\mathcal{O}(k^2)$ is optimal, or whether a more sophisticated weight substitution can give a subquadratic bound.

Open Problem 1. Does WEIGHTED FACTORCVD admit a kernel with a subquadratic bound on vertices or weights?

For FACTORCC and DIFFERENCECC we showed that the hard part of the problem is the size balancing of clusters by complete joins, while completing each component is polynomial. As size balancing is the bottleneck of both BALANCED CLUSTER COMPLETION and BALANCED CLUSTER EDITING, a single exponential algorithm for BALANCED CLUSTER COMPLETION would also give a single exponential algorithm for BALANCED CLUSTER EDITING.

Open Problem 2. Does FACTORCC and DIFFERENCECC admit single exponential time FPT-algorithms?

For the problems of BALANCED CLUSTER EDITING and BALANCED CLUSTER DELETION, we provided kernels with $\mathcal{O}(k^2)$ vertices. While our kernel for BALANCED CLUSTER DELETION is an improvement of the best kernel for the less studied problem CLUSTER DELETION, the best kernel for CLUSTER EDITING has $\mathcal{O}(2k)$ vertices. Similar to the vertex deletion problems, this raises the question of whether there is a inherent difference between the regular CLUSTER EDITING problem, and the BALANCED CLUSTER EDITING, or do they have the same lower bound on the number of vertices in the kernel.

Open Problem 3. Does FACTORCE and DIFFERENCECE admit kernels with a linear number of vertices?

An interesting extension to FACTOR-α BALANCED CLUSTERING and its absolute difference version, is the problem of FACTOR-α BALANCED d-CLUSTERING where we also give the number d of clusters in a solved instance.

FACTOR-α BALANCED d-CLUSTERING

Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$

Question: Is it possible by at most k graph modifications to get a cluster graph (i.e a disjoint union of cliques) with exactly d components, where for all components C_i, C_j in G, it holds that $\alpha \cdot |V(C_i)| \geq |V(C_j)|$?

Another related problem is the problem of instead of bounding the size of each cluster by the sizes of other clusters, we give a lower bound p and an upper bound q such that it holds that for all clusters we have $p \leq |V(C_i)| \leq q$. This problem is known as CAPACITATED CLUSTERING in the literature, and notice that when $q = \alpha \cdot p$ or $q = p + \delta$ it can be used to solve the problems studied in this thesis by iterating through all possible values of p.

CAPACITATED CLUSTERING

Input: A graph $G = (V, E)$, a budget $k \in \mathbb{N}$ and integers (p, q)

Question: Is it possible by at most k graph modifications to get a cluster graph (i.e a disjoint union of cliques) where for each component C_i in G, it holds that $p \leq |V(C_i)| \leq q$?

Bibliography

[1] N. Alon, G. Gutin, E. J. Kim, S. Szeider, and A. Yeo. Solving MAX-r-SAT above a tight lower bound. In: *Algorithmica* 61.3 (2011), pp. 638–655.

[2] S. Böcker. A golden ratio parameterized algorithm for cluster editing. In: *Journal of Discrete Algorithms* 16 (2012), pp. 79–89.

[3] S. Böcker and J. Baumbach. Cluster Editing. In: *The Nature of Computation. Logic, Algorithms, Applications*. Ed. by P. Bonizzoni, V. Brattka, and B. Löwe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 33–44. ISBN: 978-3-642-39053-1.

[4] S. Böcker and P. Damaschke. Even faster parameterized cluster deletion and cluster editing. In: *Information Processing Letters* 111.14 (2011), pp. 717–721.

[5] H. L. Bodlaender, R. G. Downey, M. R. Fellows, and D. Hermelin. On problems without polynomial kernels. In: *Journal of Computer and System Sciences* 75.8 (2009), pp. 423–434.

[6] A. Boral, M. Cygan, T. Kociumaka, and M. Pilipczuk. A fast branching algorithm for cluster vertex deletion. In: *Theory of Computing Systems* 58.2 (2016), pp. 357–376.

[7] Y. Cao and J. Chen. Cluster editing: Kernelization based on edge cuts. In: *Algorithmica* 64.1 (2012), pp. 152–169.

[8] C. Crespelle, P. G. Drange, F. V. Fomin, and P. A. Golovach. A survey of parameterized algorithms and the complexity of edge modification. In: *arXiv preprint arXiv:2001.06867* (2020).

[9] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized algorithms*. Springer, 2015.

[10] R. Diestel. *Graph Theory, 4th Edition*. Vol. 173. Graduate texts in mathematics. Springer, 2012. ISBN: 978-3-642-14278-9.

[11] R. G. Downey and M. R. Fellows. *Parameterized complexity*. Springer, 1999.

[12] F. V. Fomin, D. Lokshtanov, S. Saurabh, and M. Zehavi. *Kernelization: theory of parameterized preprocessing*. Cambridge University Press, 2019.

[13] M. R. Garey and D. S. Johnson. *Computers and intractability*. Vol. 174. Freeman San Francisco, 1979.

[14] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Graph-modeled data clustering: Fixed-parameter algorithms for clique generation. In: *Italian Conference on Algorithms and Complexity*. Springer. 2003, pp. 108–119.

[15] J. Guo. A more effective linear kernelization for cluster editing. In: *Theoretical Computer Science* 410.8-10 (2009), pp. 718–726.

[16] W.-L. Hsu and T.-H. Ma. Substitution decomposition on chordal graphs and applications. In: *ISA'91 Algorithms*. Ed. by W.-L. Hsu and R. C. T. Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 52–60. ISBN: 978-3-540-46600-0.

[17] F. Hüffner, C. Komusiewicz, H. Moser, and R. Niedermeier. Fixed-parameter algorithms for cluster vertex deletion. In: *Theory of Computing Systems* 47.1 (2010), pp. 196–217.

[18] C. Komusiewicz and J. Uhlmann. Cluster editing with locally bounded modifications. In: *Discrete Applied Mathematics* 160.15 (2012), pp. 2259–2270.

[19] T.-N. Le, D. Lokshtanov, S. Saurabh, S. Thomassé, and M. Zehavi. Subquadratic kernels for implicit 3-hitting set and 3-set packing problems. In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2018, pp. 331–342.

[20] J. M. Lewis and M. Yannakakis. The node-deletion problem for hereditary properties is NP-complete. In: *Journal of Computer and System Sciences* 20.2 (1980), pp. 219–230.

[21] G.-H. Lin, P. E. Kearney, and T. Jiang. Phylogenetic k-root and Steiner k-root. In: *International Symposium on Algorithms and Computation*. Springer. 2000, pp. 539–551.

[22] J. M. van Rooij, M. E. van Kooten Niekerk, and H. L. Bodlaender. Partition into triangles on bounded degree graphs. In: *Theory of Computing Systems* 52.4 (2013), pp. 687–718.

[23] S. E. Schaeffer. Graph clustering. In: *Computer science review* 1.1 (2007), pp. 27–64.

[24] R. Shamir, R. Sharan, and D. Tsur. Cluster graph modification problems. In: *Discrete Applied Mathematics* 144.1-2 (2004), pp. 173–182.

[25] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[26] R. Xu and D. Wunsch. *Clustering*. Vol. 10. John Wiley & Sons, 2008.